Universidad Nacional Autónoma de México
Facultad de Contaduría y Administración
Sistema Universidad Abierta y Educación a Distancia

Licenciatura en Informática

Informática VI. Programación e Implementación de Sistemas

Apunte electrónico

SUAyED

COLABORADORES

DIRECTOR DE LA FCA
Dr. Juan Alberto Adam Siade

SECRETARIO GENERAL
L.C. y E.F. Leonel Sebastián Chavarría

– – – –

COORDINACIÓN GENERAL
Mtra. Gabriela Montero Montiel
Jefe de la División SUAyED-FCA-UNAM

COORDINACIÓN ACADÉMICA
Mtro. Francisco Hernández Mendoza
FCA-UNAM

– – –

AUTOR
Lic. Edith Tapia Rangel

DISEÑO INSTRUCCIONAL
Lic. Guadalupe Montserrat Vázquez Carmona

CORRECCIÓN DE ESTILO
Lic. José Antonio Medina Carranza

DISEÑO DE PORTADAS
L.CG. Ricardo Alberto Báez Caballero
Mtra. Marlene Olga Ramírez Chavero
L.DP. Ethel Alejandra Butrón Gutiérrez

DISEÑO EDITORIAL
Mtra. Marlene Olga Ramírez Chavero

OBJETIVO GENERAL

Al finalizar el curso, el alumno aplicará el proceso de implementación y las pruebas para la construcción de sistemas de información.

TEMARIO OFICIAL
(64 horas)

	Horas
1. Introducción	10
2. Modelo de implementación	10
3. Plan de implementación	10
4. Implementación de componentes	24
5. Integración de subsistemas y sistemas	10
Total	**64**

INTRODUCCIÓN

A lo largo de este curso se abordarán diversos aspectos relacionados con la programación e implementación de sistemas informáticos, de tal forma que sea posible lograr un panorama completo de las actividades a realizar, las herramientas disponibles para ello y algunas metodologías de apoyo.

De esta forma, en la primera unidad se abordarán aquellos aspectos conceptuales y de buenas prácticas propios de la programación de sistemas, como los paradigmas de programación –los cuales atienden a una serie de patrones que modelan y definen la estructura de un programa–, y los principios de programación (que se constituyen como una serie de buenas prácticas).

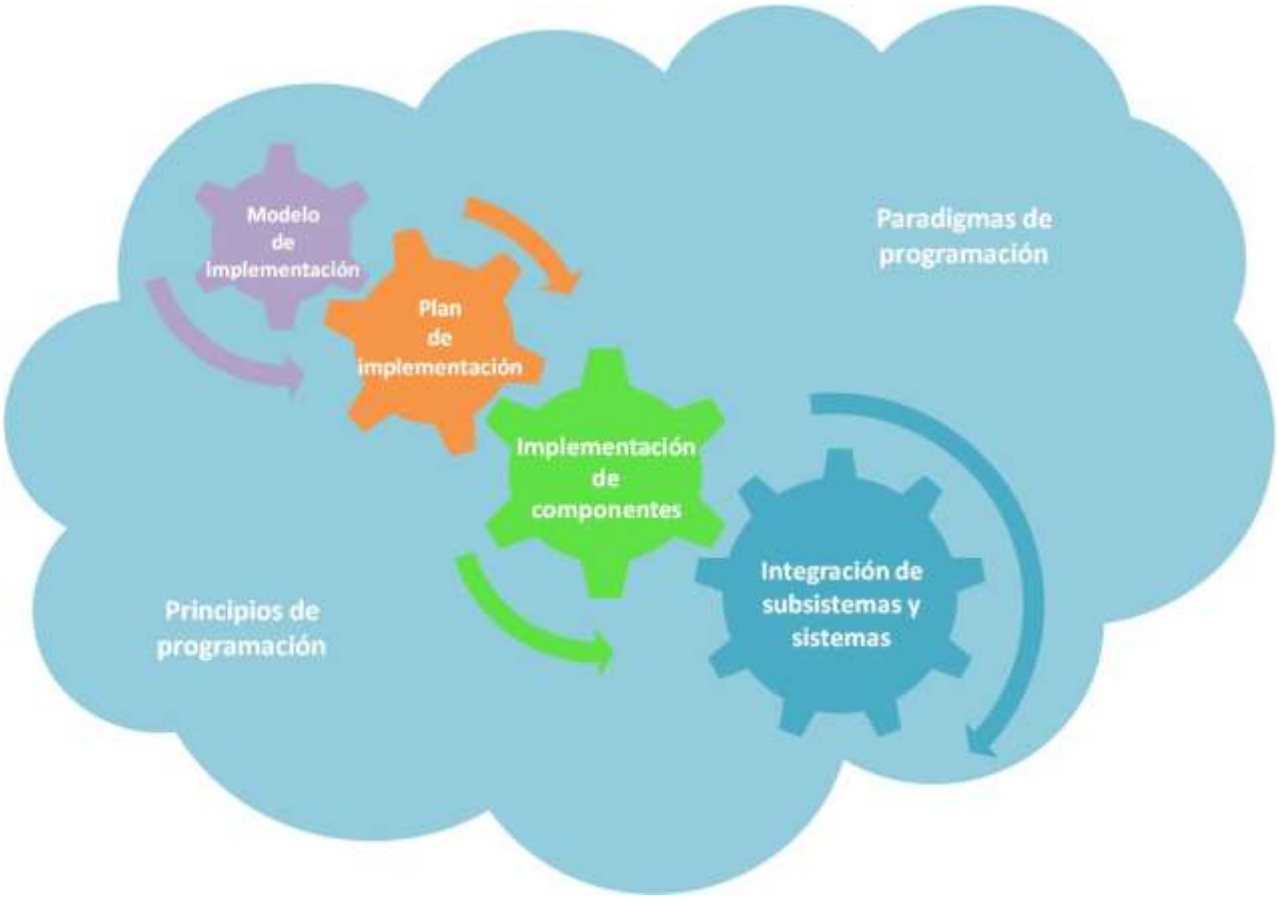
En la segunda unidad se presentarán varios elementos mediante los cuales es posible conformar el modelo de implementación de un sistema informático. Tal es el caso de los denominados patrones de diseño (que no son otra cosa que una solución probada a un problema específico) o los marcos de trabajo o conjunto de componentes que son comunes a un determinado dominio de aplicación. En este mismo apartado se caracterizarán dos metodologías de desarrollo de *software*: la Programación Extrema y el desarrollo rápido de aplicaciones (Rapid Application Development [RAD]).

En la unidad tres se describirá el “plan de implementación”, para lo cual se abordarán algunas actividades y un elemento. Dentro de las actividades mencionadas se encuentran la definición de objetivos, la estimación de tareas y tiempo, la administración de la configuración y la administración de cambios. El elemento presentado en esta unidad será el modelo de la arquitectura propuesta u organización fundamental de un sistema encarnado en sus componentes, así como sus relaciones, el ambiente y los principios que orientan su diseño y evolución.

La unidad cuatro presentará los aspectos involucrados durante la implementación de componentes, comenzando primero por la caracterización del componente para posteriormente abordar los estándares y buenas prácticas de implementación, el diseño y modelo de componentes, las técnicas de implementación y, como último tema, lo relacionado con el proceso de depuración y los métodos existentes para revisar el código de programación.

Finalmente, la quinta unidad (denominada Integración de subsistemas y sistemas) atenderá la actividad culminante dentro del desarrollo de *software*: llevarla a cabo requiere de la comprensión de los procesos de integración mismos, de sus estrategias y técnicas, de los procesos que comprenden las pruebas de integración, de las métricas para medir la calidad de los sistemas informáticos y de las actividades para elaborar la documentación del sistema.

ESTRUCTURA CONCEPTUAL



UNIDAD 1

Introducción

OBJETIVO PARTICULAR

Identificar los diferentes estilos y estándares de programación para la construcción de sistemas.

TEMARIO DETALLADO
(10 horas)

1. Introducción

1.1. Paradigmas de programación

1.2. Principios de programación

INTRODUCCIÓN

Toda vez que se ha realizado el levantamiento de requerimientos a partir de las necesidades de los usuarios y se han diseñado los principales elementos del *software*, se pasa a la etapa de implementación –que incluye diferentes actividades, como la programación.

Una manera de iniciar la programación es mediante el abordaje de los distintos paradigmas de programación que existen, los cuales atienden a una serie de patrones que modelan y definen la estructura de un programa.

En esta unidad se revisarán las tres grandes clasificaciones de paradigmas (operacionales, declarativos/definicionales y demostrativos) con sus respectivos subparadigmas.

Posteriormente se revisarán los principios de programación establecidos por Davis en su publicación “201 Principles of Software Development”, obra considerada por la Association for Computing Machinery (ACM) como uno de los 20 libros clásicos de las ciencias de la computación en 2006.

1.1. Paradigmas de programación

De acuerdo con Ambler (et ál., 1992), un paradigma de programación es una colección de patrones conceptuales que modelan el proceso de diseño para finalmente determinar la estructura de un programa.

Por su parte, Pérez y López (2007: 1) señalan que un paradigma de programación "provee (y determina) la visión y métodos que un programador utiliza en la construcción de un programa o subprograma. Diferentes paradigmas resultan en diferentes estilos de programación y en diferentes formas de pensar la solución de los problemas".

Asimismo, señalan que "un paradigma fija las reglas y propiedades, pero también ofrece herramientas para el desarrollo de aplicaciones" (2007: 2).

La categorización de paradigmas que se tratará en este documento está basada en lo establecido por Ambler (et ál., 1992), quienes agruparon los paradigmas en tres grandes categorías:

- Operacionales
- Declarativos/definicionales
- Demostrativos

Paradigmas operacionales

Se caracterizan por una organización de actividades mediante secuencias de cómputo paso a paso. Tiene como inconvenientes la dificultad para definir el resultado de las secuencias y lograr que la verificación sea idéntica a la solución esperada.

Los paradigmas operacionales se dividen en dos tipos:

- Los que modifican la representación de datos
- Los que continuamente crean nuevos datos

Paradigmas operacionales que modifican la representación de datos

En este tipo de estructuras las variables se asignan a espacios específicos de memoria, lo que modifica constantemente sus representaciones de datos. Ejemplos de este tipo de paradigmas son:

- *El paradigma imperativo.* La computadora almacena y codifica las representaciones y ejecuta una secuencia de comandos para modificar esa codificación almacenada, de tal manera que el estado final represente el resultado correcto. Esta estructura es coherente con la arquitectura de la máquina de Von Neumann. Muchos lenguajes de programación soportan este paradigma, aunque actualmente tienen mecanismos adicionados para funciones que crean datos, recursión y asignación dinámica mediante apuntadores, entre otros. Está orientado para pensar en conceptos globales.

- *El paradigma orientado a objetos*. En este paradigma los procedimientos operan sobre valores abstractos denominados “objetos”, en lugar de representaciones almacenadas. Los objetos encapsulan procesos y datos, se comunican mediante mensajes, utilizan secuencias operacionales para alterar sus representaciones internas y forman parte de clases, las cuales heredan características a los objetos que se definen dentro de ellas. Algunos lenguajes que soportan este paradigma son extensiones de lenguajes que de origen eran imperativos. Este paradigma orienta a pensar en conceptos individuales.

→ **Paradigmas que continuamente crean nuevos datos**

Denominados también “paradigmas funcionales operacionales” (diferentes a los paradigmas funcionales en el ámbito declarativo/definicional), estos paradigmas están basados en modelos matemáticos de composición funcional, donde el resultado de una operación es la entrada a la siguiente y así sucesivamente, hasta que la composición entrega el resultado deseado. En él, las funciones pueden tratarse como datos y son especificadas mediante controles de secuencia. Estos controles pueden ser de tipo secuencial (uno detrás del otro) o paralelo, el cual se maneja mediante dos propuestas: la síncrona y la asíncrona.

→ **Paradigmas definicionales**

En estos paradigmas, un programa se construye declarando hechos, reglas, restricciones, ecuaciones, transformaciones y otras propiedades del resultado para que –a partir de estos elementos– el sistema genere un esquema, incluyendo la orden para crear la solución. Puesto que no se define una secuencia de control, existen dos propuestas sobre cómo se resuelve esta situación:

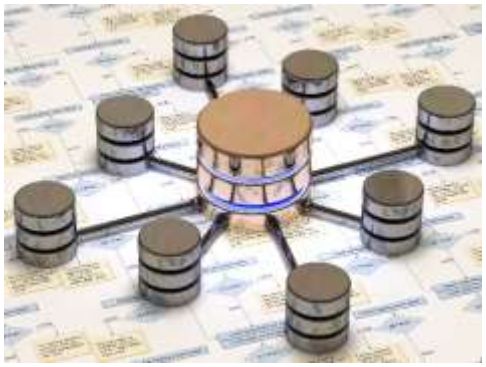
- Paradigmas declarativos
- Paradigmas pseudodeclarativos

Paradigmas declarativos. Los paradigmas declarativos establecen las características que debe presentar la solución de un problema mediante un esquema, pero no presentan una descripción de cómo alcanzarla, eliminando entonces la necesidad de probar que el valor obtenido es el valor de la solución.

Entre estos paradigmas podemos encontrar los siguientes:

- *Basados en la forma*. Define al cómputo como una colección de ecuaciones que evitan un control de secuencia, pero con una restricción en la forma de estas ecuaciones, que garantiza que el sistema desarrolle una orden de evaluación suficiente para generar una solución. En este paradigma, el programador diseña una forma que incluye fórmulas para calcular valores. Este uso de formas es análogo a los formatos e implica una sintaxis en dos dimensiones, por lo que este paradigma es soportado solamente por los lenguajes de programación visuales.

- *Flujo de datos.* En este paradigma, los datos fluyen en una red de nodos, cada uno de los cuales realiza una operación que consume dicho flujo dentro del nodo y produce nuevos datos que fluyen fuera del nodo. El programador especifica solamente las ecuaciones del nodo y soporta el paralelismo. El orden de procesamiento está determinado por las interdependencias en los datos. Es muy popular en la programación visual.



- *Programación restringida.* Su idea central es restringir lo suficiente la solución para que sólo se alcancen la o las soluciones posibles. Un programa se compone por una colección de ecuaciones restrictivas denominadas restricciones; mediante una colección de restricciones el sistema encontrará una solución que las satisfaga.

Paradigmas pseudodeclarativos. Estos paradigmas pretenden presentar el esquema de solución tal como lo hacen los declarativos; sin embargo, hacen uso de técnicas de los paradigmas operacionales para introducir secuencias de control, las cuales –al igual que los paradigmas operacionales– pueden ser seriales o paralelas. Por esa razón dejan de ser declarativos propiamente.

Entre estos paradigmas se encuentran:

- *Paradigma funcional.* Trata de asemejarse a un modelo matemático mediante la expresión de funciones como lo hacen los matemáticos. En estos paradigmas el problema se aborda como un conjunto de transformaciones disjuntas que, en conjunto, definen una función computacional. Utilizan una evaluación relajada o no estricta, ya que una función no se evalúa hasta –y a menos– que sea requerida.

- *Paradigma transformacional.* Esta representación consiste en un conjunto de reglas de transformación. Cada regla se compone de un encabezado seguido de un cuerpo. El sistema encuentra una subexpresión que coincida con el encabezado de una regla. Esa subexpresión es entonces reescrita mediante la sustitución del cuerpo de la regla, en lugar del encabezado de la subexpresión. El encabezado puede contener variables que son capaces de enlazarse con porciones de la subexpresión coincidente; tales variables, enlazadas a continuación, se pueden utilizar en la expansión del cuerpo de la regla. La expansión del cuerpo de la regla consiste en la sustitución de las variables con sus valores enlazados o, bien, en la sustitución de ellos con el resultado de algún cálculo de sus valores enlazados. El proceso se repite hasta que la expresión no contiene subexpresiones reducibles.

- *Paradigma lógico.* Asume que partimos de un conjunto de hechos y reglas conocidos. Solamente es la declaración del componente lógico de un algoritmo. El sistema desarrolla el componente de control de secuencia. En este paradigma, la evaluación asume que cuando se selecciona una regla, es porque ésta es la única posibilidad o la necesaria para resolver el problema. Es decir, se encuentra una solución si un conjunto de reglas adecuado y las sustituciones a dichas reglas producen un conjunto de reglas aterrizadas (sin variables libres), suficientes para deducir el resultado de los hechos conocidos.

→ **Paradigmas demostrativos**

Los programadores demuestran soluciones a instancias específicas de problemas similares y dejan que el sistema generalice una solución operacional de estas demostraciones. Sus propuestas pueden ser:

- *Inferenciales:* buscan generalizar mediante un razonamiento basado en conocimiento. Trata de determinar las formas mediante las cuales un grupo de objetos son similares, dibujando generalizaciones de estas similitudes.

- *No inferenciales:* buscan resolver el problema de cómo el usuario instruye al sistema para generalizar a partir de ejemplos concretos. Es una propuesta de pensamiento de “abajo hacia arriba”, por lo que toma ventaja de nuestra habilidad natural de pensamiento concreto, lo cual es opuesto a los otros paradigmas que parten del pensamiento de “arriba hacia abajo”.

1.2. Principios de programación

Podemos entender los principios de programación como directrices que los programadores deben utilizar en lo posible para lograr “buenos” programas.

Davis (1995) establece varios principios relacionados con todo el ciclo de vida del desarrollo de *software*. Entiende la programación (codificación) como un conjunto de actividades que incluyen trasladar los algoritmos especificados en la etapa de diseño, en programas escritos en un lenguaje de cómputo. De manera particular, los principios que establece con respecto a la programación son:

a) Evita trucos

Se refiere a evitar el uso de “algoritmos” o “funciones” muy ingeniosos (o difíciles de entender), pero poco claros para la generalidad de las personas. Si bien el código puede lograr los resultados esperados, puede resultar difícil de mantener. Los trucos generalmente se usan porque:

- Los programadores son muy inteligentes y quieren demostrarlo.
- Los que mantienen el sistema también demuestran que son muy inteligentes cuando “descifran” el truco.
- Brindan seguridad en el trabajo.

b) Evita variables globales

Prefiere el uso de variables locales, ya que las globales pueden generar errores lógicos sin que éstos sean intencionales, mismos que pueden ser difíciles de rastrear debido a que su carácter global implica que cualquiera puede alterar el valor de manera incorrecta.

c) Escribe el código para leerlo de arriba hacia abajo

Al organizar el código de arriba hacia abajo, éste se hace más comprensible porque se evidencia el propósito de cada pieza del código y cómo encaja en el todo; de esta manera, se puede reenviar la referencia sólo a la especificación y no a toda la aplicación. Entre las implicaciones de este principio están:

- Incluir una especificación detallada externa evidente que defina claramente el objetivo y uso del programa, así como para las rutinas que se acceden de manera externa, variables locales y algoritmos.
- Utilizar las construcciones de la programación estructuradas que son fáciles de seguir.

d) Evita efectos secundarios

El efecto secundario de un procedimiento es algo que el procedimiento hace, pero que no es su propósito principal; sin embargo, es visible (o sus resultados son perceptibles) desde afuera del procedimiento. Los efectos secundarios son el origen de muchos errores sutiles en el *software*, por lo que son difíciles de descubrir una vez que se manifiestan.

e) Usa nombres significativos

Algunos programadores insisten en denominar las variables con nombres sin significado. El argumento habitual es que los programadores son más productivos cuando se reduce lo que hay que teclear. En contraposición a lo anterior se establece que los nombres excesivamente acortados, en realidad, disminuyen la productividad por dos razones:

- Aumento de costos de pruebas y mantenimiento porque las personas pasan el tiempo tratando de descifrar los nombres
- Se requiere más inversión de tiempo cuando se utilizan nombres cortos porque se hace necesario adicionar comentarios, como vemos en el siguiente ejemplo:

```
N_FLT = P_FLT 1 %"Mira el siguiente vuelo" (34 letras)

Next_flight = Previous_flight + 1 (29 letras)
```

En este sentido, es recomendable elegir nombres que son fáciles de pronunciar y evitar abreviaturas crípticas; utilizar nombres cortos para las variables de corta duración y nombres más descriptivos para las variables que sirven a un propósito importante; así como utilizar la terminología del dominio de aplicación cuando sea posible.

f) Escribe programas primero para la gente

En los primeros días de la computación, lo más importante era el uso eficiente de cualquiera de los recursos en el sistema informático por su costo excesivo. Sin embargo, las cosas han cambiado. Hoy en día, el recurso más valioso es la mano de obra para el desarrollo del *software*, para mantener el *software* y para aumentar la capacidad. Por esta razón, los programadores deben pensar primero en las personas que luego intentarán entender y adaptar el *software*, por lo que cualquier cosa que se pueda hacer para ayudarlos se deberá hacer.

g) Haz uso de estructuras de datos óptimas

La estructura de datos y la estructura de los programas de manipulación de datos están íntimamente relacionadas entre sí. Al seleccionar las estructuras de datos adecuadas, los algoritmos (y, por lo tanto, su código) son fáciles de escribir, de leer y de mantener. Al prepararse para escribir un programa se deben desarrollar juntos

los algoritmos y estructuras de datos. Revisa dos, tres o más pares diferentes antes de seleccionar la mejor combinación y asegúrate de encapsular la estructura de datos en un solo componente, de modo que cuando –más adelante– encuentres una mejor estructura, se pueda cambiar fácilmente.

h) Hazlo bien antes de hacerlo rápido

Es más fácil adaptar un programa que funcione para hacerlo más eficiente que adaptar de manera rápida un programa para hacerlo que funcione.

No te preocupes acerca de la optimización al hacer tu codificación inicial, procura no caer en el extremo de utilizar algoritmos o estructuras de datos evidentemente ineficientes. Cada proyecto de *software* tiene fuertes presiones de tiempo. Algunos pueden iniciar muy relajados, pero incluso ésos acelerarán el ritmo conforme se acerquen las fechas finales. Esto significa que al momento de programar se debe asegurar primero que la aplicación se ejecute correctamente, antes de revisar los aspectos que se pueden optimizar, ya sea en velocidad de ejecución u otros aspectos.

i) Comenta antes de finalizar la programación

El código se comenta para que sea más fácil depurar, probar y mantener el *software*. Al comentar el código mientras programas (o antes), será más fácil depurar el *software*. Al depurar el *software* indudablemente se encuentran defectos. Si la falla se encuentra en la conversión del algoritmo a código, tendrás que cambiar sólo el código, no los comentarios. Si la falla es en el algoritmo, tendrán que cambiarse tanto los comentarios como el código; pero, ¿cómo reconocer un error algorítmico sin comentarios?

j) Documenta antes de iniciar la programación

Después de realizar el diseño detallado de un componente (es decir, la documentación de su interfaz externa y su algoritmo), escribe sus comentarios en línea. La mayoría de los comentarios en línea serán seguramente la interface y el algoritmo previamente documentado. Pon estos comentarios y pasa el código por el compilador para asegurarte de que no has cometido algún error (como la omisión de un delimitador de comentario). Al convertir cada línea de comentario en un segmento de programa correspondiente, el proceso de depuración será más fácil.

k) Ejecuta de manera aislada cada componente

Puede ser que se tarde 30 minutos para ejecutar de manera aislada un componente de *software* con un par de casos de pruebas sencillas, lo cual es preferible. ¿Por qué?

Considera que el sistema falla; entonces, tres o cuatro días o tres o cuatro personas se necesitan para tratar de aislar la causa de la falla, para al final contar con una media docena de componentes aislados como posibles candidatos. Cada uno recibe de sus desarrolladores un examen más detenido; cada candidato invierte 30 minutos en ejecutar de manera aislada el componente con unos pocos casos de prueba simples. En resumen, 30 minutos son más eficientes que tres o cuatro personas trabajando durante tres o cuatro días.

l) Inspecciona el código

Para encontrar errores en las pruebas, la inspección funciona mejor. Para ello es necesario definir los criterios para completar una inspección y llevar un registro de los tipos de errores que se encuentran en la inspección. El plan original del proyecto debe tener en cuenta el tiempo para inspeccionar (y corregir) cada componente. Se puede pensar que el proyecto no puede tolerar tales "lujos", pero no es así. Los datos han demostrado que incluso se puede reducir el tiempo de prueba de un 50-90 %.

m) Se pueden utilizar lenguajes no estructurados

Es posible escribir un código estructurado en lenguajes que no contienen estructuras de control, como los lenguajes ensambladores, al documentar el código con las sentencias de control estructurados y restringir el uso de GOTO para implementar estas estructuras. Para ello, escribe los algoritmos utilizando las estructuras de control anteriores. A continuación, deben convertirse en comentarios en línea para después traducir tale comentarios en las sentencias equivalentes del lenguaje no estructurado de programación. Aparecerá GOTO, pero será implementando mejores construcciones que faciliten y no obstaculicen, que sean legibles, sustentables y probadas.

n) El código estructurado no es necesariamente un buen código

No todos los programas estructurados son buenos. Uno puede escribir programas muy oscuros que están perfectamente estructurados. La estructura es casi una condición necesaria, pero está lejos de ser suficiente para una programación de calidad.

o) No anides muy profundo

Anidar algunas instrucciones, como las condicionales (IF-THEN-ELSE), puede simplificar la lógica del código. Sin embargo, al incrementar el nivel de anidación de estas instrucciones se disminuye en la misma proporción su comprensión, lo que conduce a que uno mismo u otra persona que revise el código se confunda.

p) Utiliza lenguajes de programación apropiados

Los lenguajes de programación varían mucho en su capacidad para ayudar a hacer el trabajo. El proyecto en específico u objetivos del producto frecuentemente dictarán el lenguaje apropiado. Para ello, te puedes apoyar en las siguientes directrices:

- Si tu objetivo principal es la portabilidad, utiliza un lenguaje que haya demostrado ser muy portátil (por ejemplo, C, FORTRAN o COBOL).
- Si pretendes lograr un desarrollo rápido, es recomendable utilizar un lenguaje que ayude a ese desarrollo rápido (4GL's, Basic, APL, C, C ++, o SNOBOL).
- Si buscas aplicaciones que requieran poco mantenimiento, utiliza un lenguaje con muchas características de calidad incorporadas (como Ada de Eiffel).
- Si tu aplicación requiere mucho uso de las cadenas de caracteres o estructuras de datos complejas, selecciona un lenguaje que los soporte.
- Si al programa le debe dar mantenimiento un grupo de expertos en el lenguaje X, entonces utiliza el lenguaje X.
- Por último, si el cliente señala que quiere el *software* en la aplicación Y, se hará en el lenguaje Y.

q) El lenguaje de programación no es excusa

Algunos proyectos se ven obligados a utilizar lenguajes de programación menos que ideales. Esto puede ser causado por: el deseo de reducir los costos de mantenimiento ("todos nuestros mantenedores saben COBOL"); para programar rápido ("tenemos la más alta productividad con C"); para garantizar una alta

fiabilidad ("los programas de Ada son los más tolerantes a fallas "), o para lograr una alta velocidad de ejecución ("nuestras aplicaciones son de tiempos críticos, por lo que se debe usar el lenguaje ensamblador").

Es posible escribir programas de calidad en cualquier idioma. De hecho, un buen programador debe ser un buen programador en cualquier lenguaje, inclusive en un lenguaje menos que ideal, en el que tendrá que trabajar más.

r) El conocimiento del lenguaje no es tan importante

Los buenos programadores son buenos independientemente del lenguaje que utilicen. Los malos programadores son malos independientemente del lenguaje que utilicen. Un buen programador debe ser capaz de aprender cualquier nuevo lenguaje de programación fácilmente. Esto se debe a que un buen programador entiende y aprecia los conceptos de una programación de calidad, no sólo la idiosincrasia sintáctica y semántica de un lenguaje de programación.

Por lo anterior, el principal impulsor de la selección de idioma para un proyecto debe ser lo apropiado que resulta para él mismo, no el aumento de quejas de los programadores. En ese sentido, si algunos abandonan el proyecto porque se seleccionó un lenguaje diferente, probablemente el proyecto estará mejor.

s) Da formato a tus programas

La comprensibilidad de un programa se ha mejorado en gran medida mediante el uso de protocolos de indentación estándar, como el siguiente:

- Se debe utilizar un único modelo de indentación a lo largo de todo el programa.
- Los bloques de código (por ejemplo, dentro de un bucle o el cuerpo de una función) deberán ir indentados.
- Si un bloque de código está anidado dentro de otro bloque de código, el bloque más interno deberá ir indentado respecto al externo.
- Para evitar que cuando haya múltiples bloques anidados (unos dentro de otros) el código sobrepase la anchura de la página, se recomienda que el número de espacios de la indentación no sea excesivo (3 o 4).

t) No programes muy rápido

La programación es análoga a la construcción de un edificio: requiere mucho trabajo preliminar. No cedas a codificar antes de tiempo (porque la administración quiere ver progresos); por el contrario, asegúrate de que los requisitos y el diseño son correctos y apropiadas antes de la definición de la aplicación y, ciertamente, antes de programar el producto final.

RESUMEN

A lo largo de la unidad se abordaron las características principales de los grandes paradigmas. La primera categoría de ellos, denominada “operacionales”, se refiere a que organizan los programas mediante secuencias, con sus subparadigmas: imperativo, orientado a objetos, y funcionales operacionales.

Una segunda categoría de paradigmas, denominados “declarativos/definicionales”, se construye con base en hechos y reglas lógicas, con los subparadigmas: declarativos basados en la forma, declarativos por flujo de datos, y declarativos de programación restringida; además de los subparadigmas pseudodeclarativos (tales como el paradigma funcional, el paradigma transformacional y el paradigma lógico).

Finalmente, una tercera categoría de paradigmas, denominada “demostrativos”, se resuelve atendiendo problemas similares, contando entre sus subparadigmas al paradigma inferencial y al paradigma no inferencial.

Posteriormente se revisaron los 19 principios de programación (codificación) establecidos por Davis, que brindan las directrices para orientarnos a lograr buenas prácticas en la actividad de programación, a saber: evita trucos; evita variables globales; escribe el código para leerlo de arriba hacia abajo; evita efectos secundarios; usa nombres significativos; escribe programas primero para la gente; haz uso de estructuras de datos óptimos; hazlo bien antes que hacerlo rápido; comenta antes de finalizar la programación; documenta antes de iniciar la programación; ejecuta de manera aislada cada componente; inspecciona el código; se pueden utilizar lenguajes no estructurados; no anides muy profundo; utiliza lenguajes de programación apropiados; el lenguaje de programación no es excusa; el conocimiento del lenguaje no es tan importante; da formato a sus programas, y no programes muy rápido.

BIBLIOGRAFÍA

Bibliografía básica

Ambler, A. L., Burnett, M. M. y Zimmerman, B. A. (1992). Operational versus definitional: A perspective on programming paradigms. *Computer, 25* (9): 28-43. Consultado el 02 de septiembre de 2013, disponible en: ftp://ftp.engr.orst.edu/pub/burnett/Computer-paradigms-1992.pdf

Davis, A. (1995). *201 principles of software development.* Estados Unidos: McGraw-Hill.

Referencias bibliográficas

Ambler, A. L., Burnett, M. M. y Zimmerman, B. A. (1992). Operational versus definitional: A perspective on programming paradigms. *Computer, 25* (9): 28-43. Consultado el 02 de septiembre de 2013, disponible en: ftp://ftp.engr.orst.edu/pub/burnett/Computer-paradigms-1992.pdf

Pérez, P. y López, M. (s. f.). *Multiparadigma en la enseñanza de la programación.* Buenos Aires: Universidad Nacional del Comahue. Consultado el 02 de septiembre de 2013, disponible en: http://sedici.unlp.edu.ar/bitstream/handle/10915/20499/Documento_completo.pdf?sequence=1

Sitios electrónicos

Sitio	Descripción
http://see.stanford.edu/see/lecturelist.aspx?coll=2d712634-2bf1-4b55-9a3a-ca9d470755ee	Curso abierto de Stanford “Introduction to computer science. Programming paradigms”.
https://www.udacity.com/course/cs212	Curso en UDACITY “Design of Computer Programs. Programming Principles”.

UNIDAD 2

Modelo de implementación

OBJETIVO PARTICULAR

Diseñar un modelo de implementación de los paquetes de subsistemas que conforman el sistema.

TEMARIO DETALLADO
(10 horas)

2. Modelo de implementación

2.1. Patrones de diseño
2.2. Marcos de trabajo
2.3. Programación extrema
2.4. Frases del desarrollo rápido de aplicaciones (RAD)

INTRODUCCIÓN

A lo largo de esta unidad se revisarán los diversos aspectos que permiten conformar el modelo de implementación de un sistema informático. En primer lugar se abordará el término *patrón de diseño*, entendido como una solución probada para un problema de diseño específico que puede aplicarse en casos similares. Del mismo modo, se identificarán los elementos esenciales que lo conforman (nombre, problema, solución y consecuencias), la manera en que debe seleccionar, cómo utilizarlo y los principales tipos de patrones de diseño que existen, conociendo de cada uno su nombre e intención.

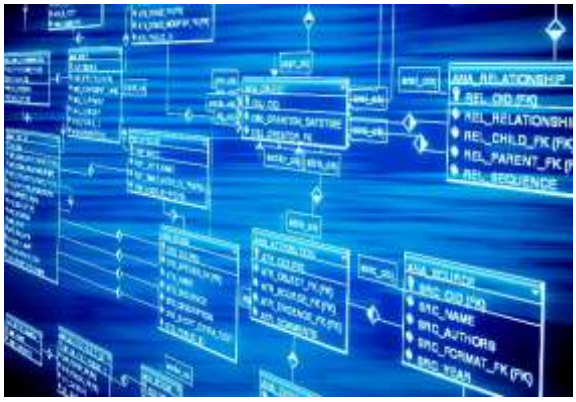
Enseguida se revisará el tema de los marcos de trabajo, entendidos como un conjunto de componentes que son comunes a un determinado dominio de aplicación. De éstos conoceremos su clasificación, las dimensiones que los caracterizan y cuáles se consideran más populares dentro del ámbito de los desarrolladores de *software*.

Posteriormente se abordará la metodología de desarrollo de *software* conocida como “programación extrema”, identificando los conceptos básicos que la sustentan, el proceso mediante el cual se desarrolla y las reglas y prácticas de dicha metodología.

Por último, se estudiará el desarrollo rápido de aplicaciones (Rapid Application Development [RAD]), entendido como un ciclo de vida de desarrollo de sistemas, identificando las etapas que lo componen y las tareas que se desarrollan en cada una de esas fases.

2.1. Patrones de diseño

Evolución histórica de los patrones de diseño



La primera referencia que se tiene sobre los patrones de diseño es la que presentó Christopher Alexander en su obra “The timeless way of building”, donde propuso una nueva teoría arquitectónica basada en la comprensión y configuración de **patrones de diseño**.

Con este antecedente, Beck y Cunningham establecieron en su publicación “Using pattern languages for object-oriented programs” (1987) que un patrón de lenguaje guía al diseñador, proveyéndolo de soluciones probadas.

A partir de los años noventa los patrones de diseño cobraron relevancia, siendo muestra de ello la publicación “Design patterns: elements of reusable object-oriented software”, escrito por el Gang of Four (GoF) –grupo conformado por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. Esta publicación, con sus 23 patrones de diseño expuestos, sigue siendo el referente básico sobre este tema.

Definición de patrones de diseño

Alexander, citado por los GoF (1995: 12), menciona que “cada patrón describe un problema que se presenta continuamente en nuestro entorno, así como la solución esencial a ese problema, de manera tal que se pueda usar esa solución un millón de veces sin tener que hacerlo de la misma manera dos veces”.

Para el GoF (1995), los patrones de diseño son descripciones de objetos y clases comunicándose entre sí, adaptados para resolver un problema de diseño general en un contexto particular.

Asimismo, el GoF establece que los patrones de diseño cuentan con cuatro elementos esenciales:

1. El nombre: elemento que podemos utilizar para describir en una o dos palabras un problema de diseño, sus soluciones y consecuencias. Al asignar el nombre a un patrón de diseño, de inmediato incrementamos nuestro vocabulario de diseño. También nos permite diseñar a niveles de abstracción más elevados.

2. El problema: describe cuándo aplicar el patrón y explica el problema y su contexto. Puede describir problemas de diseño (representación de algoritmos y objetos) o estructuras de clases y objetos, así como incluir, algunas veces, una lista de condiciones necesarias para aplicar el patrón.

3. La solución: describe los elementos que integran el diseño, sus relaciones, responsabilidades y colaboración, no así su aplicación en particular. Es una descripción abstracta de un problema de diseño y de cómo una disposición general de elementos lo resuelve.

4. Las consecuencias: son los resultados, ventajas y desventajas de aplicar el patrón. Están referidas principalmente en función de las variables de espacio y tiempo. Las consecuencias permiten entender y evaluar el patrón de forma explícita.

Cuando se realizan modificaciones a los sistemas, éstas pueden implicar procesos tales como redefinición de clases, reimplementación, modificaciones de los usuarios y nuevas pruebas. Tal rediseño afecta muchas partes del sistema informático, por lo que los cambios no anticipados son generalmente muy costosos.

En este sentido, los patrones de diseño ayudan a evitar tales problemas, asegurando que el sistema se pueda modificar de maneras específicas. Cada patrón de diseño permite que algún aspecto de la estructura del sistema varíe, independientemente de los demás aspectos que lo conforman, lo que genera un sistema más robusto a un tipo particular de cambio.

Cómo seleccionar un patrón de diseño

Existen varias propuestas para seleccionar el patrón de diseño correcto para cada problema, a saber:

- Considerar cómo los patrones resuelven problemas de diseño.
- Explorar las secciones de propósitos del patrón.
- Estudiar cómo se interrelacionan los patrones.
- Estudiar comparativamente los propósitos de los patrones.
- Examinar la causa de rediseño.
- Considerar que debería ser variable en su diseño.

Cómo utilizar un patrón de diseño

- Revisar los aspectos generales que forman parte del patrón, prestando especial atención a las secciones de aplicabilidad y consecuencias, para asegurarse de que el patrón es el indicado para resolver el problema.
- Revisar con atención y estudiar las secciones de estructura, participantes y colaboraciones, para asegurar la comprensión de las clases y objetos del patrón, así como la forma de relacionarse entre sí.

- Revisar la sección de ejemplos de código para ver un ejemplo concreto del patrón en código. Estudiar el código permitirá aprender cómo implementar el patrón.
- Elegir nombres para los patrones participantes que sean significativos en el contexto de la aplicación, lo que hace que el patrón sea más explícito en la misma.
- Definir las clases y sus interfaces para establecer sus relaciones de herencia y definir las variables de instancia que representan los datos y las referencias a objetos. Identificar las clases existentes en la aplicación que el patrón afectará y modificará.
- Definir los nombres específicos de la aplicación (coherentes con las convenciones de la nomenclatura), para las operaciones en el patrón que dependerán de la misma, apoyándose en las responsabilidades y colaboraciones asociadas para cada operación a manera de guía.
- Implementar las operaciones para llevar a cabo las responsabilidades y colaboración en el patrón. Puedes apoyarte en las secciones de implementación y código.

Catálogo de patrones

De acuerdo con Gamma, Helm, Johnson y Vlissides (1980:18-19), éste es el catálogo de los diferentes patrones de diseño:

- *Método de fábrica (Factory method)*
  Intención: definir una interfaz para crear un objeto, permitiendo a las subclases decidir qué clase instanciar. Permitir que una clase difiera la instanciación a subclases.

- *Adaptador – Adapter (ambos ámbitos: clase y objeto)*

  Intención: convertir la interfaz de una clase en otra interfaz que los clientes esperan. Permitir a clases con interfaces incompatibles trabajar juntas, ya que de otro modo no podrían hacerlo.

- *Intérprete - Interpreter*

  Intención: dado un lenguaje de programación, definir una representación para la gramática de dicho lenguaje, junto con un intérprete que utilice la representación para interpretar las sentencias en el lenguaje.

- *Método de plantilla - Template method*

  Intención: definir el esqueleto de un algoritmo en una operación, aplazando algunos pasos a las subclases. Permitir a las subclases redefinir algunos pasos de un algoritmo sin cambiar la estructura del mismo.

- *Fábrica abstracta – Abstract factory*

  Intención: proporcionar una interfaz para crear familias de objetos relacionados o dependientes, sin especificar sus clases concretas.

- *Constructor - Builder*

  Intención: separar la construcción de un objeto complejo de su representación de manera que el mismo proceso de construcción pueda crear diferentes representaciones.

- *Prototipo - Prototype*

  Intención: especificar los tipos de objetos que se crearán usando una instancia prototípica, y crear nuevos objetos copiando dicho prototipo.

- *Uno solo - Singleton*

  Intención: asegurarse de que una clase tiene solamente una instancia y proporcionar un punto de acceso global a ella.

- *Puente - Bridge*

  Intención: desasociar una abstracción de su implementación, de manera que las dos puedan variar de independientemente.

- *Compositor - Composite*

  Intención: componer objetos en estructuras de árbol para representar jerarquías parte-todo. Permite a los clientes tratar objetos individuales y composiciones de objetos de manera uniforme.

- *Decorador - Decorator*

  Intención: fijar dinámicamente responsabilidades adicionales a un objeto. Proporcionar una alternativa flexible para crear subclases con funcionalidades extendidas.

- *Fachada - Facade*

  Intención: proporcionar una interfaz unificada para un conjunto de interfaces de un subsitema. Definir una interfaz de alto nivel que haga al subsistema más fácil de usar.

- *Peso mosca - Flyweight*

  Intención: hacer uso de la compartición para soportar eficientemente un gran número de objetos de grano fino.

- *Proxy - Proxy*

  Intención: proporcionar un sustituto o marcador de posición para otro objeto y controlar el acceso al mismo.

- *Cadena de responsabilidades - Chain of resposability*

  Intención: evitar el acoplamiento del remitente de una solicitud a su receptor, dándole la oportunidad de manejar la petición a más de un objeto. Encadenar los objetos receptores y pasar la petición a lo largo de la cadena hasta que un objeto lo resuelva.

- *Comando - Command*

  Intención: encapsular una petición como un objeto, permitiendo parametrizar clientes con diferentes peticiones, encolar o iniciar peticiones y apoyar las operaciones que se pueden deshacer.

- *Iterador - Iterator*

  Intención: proporcionar una forma para acceder a los elementos de un objeto agregado secuencialmente, sin exponer su representación subyacente.

- *Mediador - Mediator*

  Intención: definir un objeto que encapsula como un conjunto de objetos interactuantes. Promover la articulación flexible manteniendo los objetos referenciados entre sí, de manera explícita, variando así su interacción de forma independiente.

- *Momento - Memento*

  Intención: sin violar la encapsulación, capturar y exteriorizar el estado interno de un objeto, de modo que dicho objeto pueda ser restaurado a ese estado posteriormente.

- *Observador - Observer*

  Intención: definir una dependencia uno-a-muchos entre objetos, de modo que cuando cambie el estado de un objeto, todos sus dependientes sean notificados y actualizados automáticamente.

- *Estado - State*

  Intención: permitir que un objeto modifique su comportamiento cuando cambia su estado interno. El objeto aparecerá para cambiar su clase.

- *Estrategia - Strategy*

  Intención: definir una familia de algoritmos, cada uno encapsulado, y hacerlos intercambiables. Permitir que los algoritmos varíen independientemente de cómo los utilicen los clientes.

- *Visitante - Visitor*

  Intención: representar una operación a realizar sobre los elementos de una estructura de objeto. Permitir la definición de una nueva operación sin cambiar las clases de los elementos con los que opera.

Al organizar este catálogo por propósito y ámbito se presentará la siguiente estructura:

Tabla 1. Catálogo por porpósito y ámbito

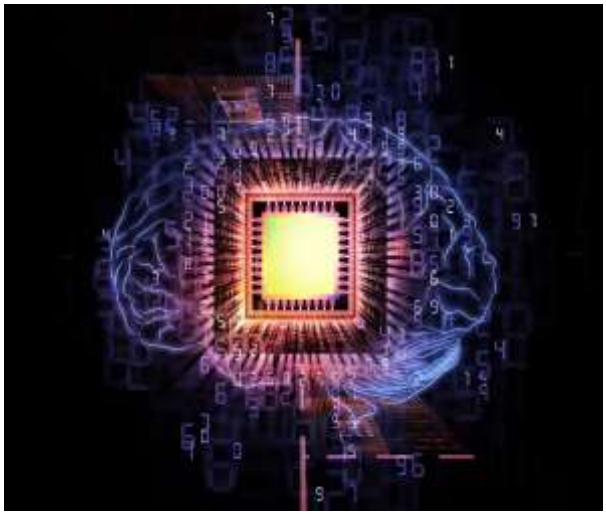
		Propósito (lo que hace el patrón)		
		De creación (proceso de creación del objeto)	**De estructura** (composición de clases y objetos)	**De comportamiento** (forma en que las clases y objetos interactúan y distribuyen responsabilidad)
Ámbito (si aplica a objetos o clases)	Clase	Método de fábrica	Adaptador	Interprete Método de plantilla
	Objeto	Fábrica abstracta Constructor Prototipo Uno solo	Adaptador Puente Compositor Decorador Fachada Peso mosca Proxy	Cadena de responsabilidades Comando Iterador Mediador Momento Observador Estado Estrategia Visitante

Elaborado con base en Gamma, Helm, Johnson y Vlissides (1980: 21)

2.2. Marcos de trabajo

El GoF (1995) estableció que un *marco de trabajo* era un conjunto de clases cooperantes que conforman un diseño reutilizable para una clase específica de *software*, esto significa que se pueden construir marcos de trabajo para la construcción de editores gráficos de distintos dominios, como el musical o el arquitectónico, o para construir aplicaciones de modelado financiero. Se puede personalizar un marco de trabajo para una aplicación en particular, creando subclases específicas para la aplicación a partir de clases abstractas especificadas en el mismo marco de trabajo.

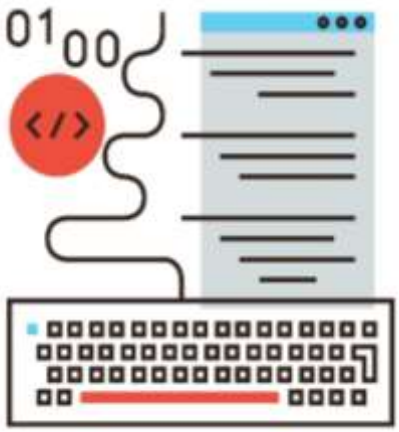
El marco de trabajo determina la arquitectura de una aplicación. Define toda la estructura general, su organización en clases y objeto, y las responsabilidades clave de los mismos, la forma en la que colaboran las clases y objetos, y los elementos de control.



Al definir el marco de trabajo, estos parámetros de diseño permiten al programador concentrarse en los detalles de la aplicación, puesto que captura las decisiones de diseño comunes al dominio de la aplicación. Ésta es la razón por la que los marcos de trabajo apoyan la reutilización del diseño sobre la reutilización del código, aunque esto no los exente de incluir subclases concretas que pueden utilizarse de manera inmediata.

El reúso a este nivel requiere controlar la aplicación y el *software* en el que se basa. Cuando se hace uso de un *toolkit* (librería de subrutinas convencional), se escribe el cuerpo principal de la aplicación y se hace una llamada al código que se desea reutilizar. Cuando se hace uso de un marco de trabajo, se reúsa el cuerpo principal y se escribe el código que se desea llamar.

Los marcos de trabajo no sólo permiten construir aplicaciones más rápido, sino también generar aplicaciones con estructuras similares, las cuales serán más fáciles de mantener y parecerán consistentes para los usuarios; por otro lado, limitan en parte la libertad creativa, puesto que varias decisiones de diseño ya están hechas.



Los marcos de trabajo son más difíciles de diseñar que las aplicaciones o que un *toolkit*. El diseñador de un marco de trabajo apuesta a que su arquitectura será funcional para todas las aplicaciones relacionadas con un dominio en específico. Un cambio sustancial en el diseño del marco de trabajo reduciría sus beneficios considerablemente, puesto que la principal contribución del marco de trabajo a la aplicación es la arquitectura que define. Por lo tanto, es necesario diseñar el marco de trabajo para que sea lo más flexible y extensible.

Por otro lado, puesto que el diseño de las aplicaciones es tan dependiente del marco de trabajo, éstas últimas son muy sensibles a los cambios en las interfaces del marco de trabajo, por lo que conforme evoluciona el marco de trabajo, evolucionan también las aplicaciones. Por ello, el acoplamiento “débil” cobra importancia como medio para evitar que un cambio pequeño en el marco de trabajo represente repercusiones relevantes en la aplicación.

Las cuestiones de diseño tratadas son el aspecto más crítico a considerar cuando se diseña un marco de trabajo. Un marco de trabajo que los resuelva haciendo uso de los patrones de diseño, es probable que alcance altos niveles de diseño y reúso de código que uno que no lo hace, de ahí que los marcos de trabajo denominados “maduros” suelen incorporar varios patrones de diseño, ya que los patrones de diseño permiten que la arquitectura del marco de trabajo sea adecuada para muchas aplicaciones diferentes, sin necesidad de sufrir rediseños.

Clasificación

Riehle y Gross (1998) establecieron que los marcos de trabajo se caracterizan por ser de caja negra, caja blanca o, bien, combinan ambas caracterizaciones. Estos atributos señalan el uso que se intenta dar al marco de trabajo.

- En un marco de trabajo de caja negra se trabaja fuera de la caja: un cliente (objeto) puede utilizar el marco de trabajo instanciando clases y realizando composiciones de instancias que adapta a sus necesidades.
- En un marco de trabajo de caja blanca los clientes deben primero suministrar subclases nuevas antes de que los objetos puedan crearse y organizarse en composiciones.

Por otra parte, la empresa Taligent Inc. (1994) establece que las dimensiones que caracterizan los marcos de trabajo son: el dominio del problema que resuelve el marco de trabajo, la estructura interna del marco de trabajo y cómo se utilizará el marco de trabajo.

1) **Dominio del problema que resuelve el marco de trabajo**

El dominio del problema que un marco de trabajo resuelve abarca tres aspectos: aplicación, dominio y soporte.

- Los marcos de trabajo de aplicación encapsulan experiencia aplicable a una gran variedad de programas. Comprenden un eje de funcionalidad horizontal que puede aplicarse a través de los dominios del cliente. Un marco de trabajo de este tipo es el de los marcos de trabajo comerciales para interfaces gráficas de usuario (GUI), que soportan la funcionalidad estándar requerida por todas las aplicaciones GUI (como el sistema Apple MacApp y el sistema OWL de Borland).

- Los marcos de trabajo de dominio encapsulan la experiencia en un dominio de problema específico. Abarcan un eje de funcionalidad vertical de un dominio de cliente en particular. Ejemplos de estos marcos de trabajo de dominio son: un marco de trabajo de sistemas de control para el desarrollo de aplicaciones de control de manufactura, el marco de trabajo de negociación de valores, el marco de trabajo multimedia o marco de trabajo para acceso a datos.

- Los marcos de trabajo de apoyo proporcionan servicios a nivel del sistema, tales como: acceso a archivos, soporte de computación distribuida o controladores de dispositivo. Los desarrolladores utilizan estos marcos de trabajo de forma directa o, bien, modificaciones producidas por los proveedores de sistemas. Sin embargo, estos marcos de trabajo de apoyo también pueden ser personalizados; por ejemplo, cuando se desarrolla un nuevo sistema de archivos o controlador de dispositivo.

2) Estructura interna del marco de trabajo

Identificando la estructura a alto nivel del marco de trabajo se facilita la descripción del comportamiento del mismo, y aporta un punto de inicio para el diseño de la interacción del marco de trabajo.

Así, algunos marcos de trabajo pueden describirse como gestionados – cuando una sola función de control despliega la mayoría de las acciones del marco. El desarrollador llama entonces a dicha función de control para iniciar el marco de trabajo y éste creará los objetos y llamará a las funciones apropiadas para realizar una tarea en específico.

Un marco de trabajo de aplicación, por ejemplo, generalmente usa un marco de trabajo gestionado para tomar los eventos que ingresan del usuario y los distribuye a otros objetos dentro del mismo marco.

3) Cómo se utilizará el marco de trabajo

Esta clasificación se fundamenta en cómo se utilizará el marco: guiado por arquitectura o guiado por datos; o, bien, como: dominio de la herencia contra dominio de la composición.

Los marcos de trabajo guiados por arquitectura se basan en la herencia para lograr la personalización de las aplicaciones. Los clientes personalizan el comportamiento del marco de trabajo, generando nuevas clases a partir del marco de trabajo y anulando funciones integradas a él. Estos marcos son difíciles de usar porque requieren que se escriba gran cantidad de código para lograr el comportamiento deseado.

Los marcos de trabajo guiados por datos se basan principalmente en la composición de objetos para lograr la personalización. Los clientes personalizan el comportamiento del marco de trabajo utilizando diferentes combinaciones de

objetos. Los objetos que los clientes pasan por el marco de trabajo afectan lo que hace el mismo marco, pero el marco define cómo pueden ser combinados los objetos. Estos marcos son generalmente fáciles de utilizar, pero sus funcionalidades se encuentran limitadas.

Un enfoque para la construcción de los marcos de trabajo fáciles de usar y flexibles consiste en una combinación del marco de trabajo base guiado por arquitectura, con una capa de un marco guiado por datos. Gran parte de los marcos de trabajo cuenta con elementos para que los clientes usen tanto la funcionalidad original o, bien, la modifiquen o extiendan, de modo que los clientes utilizan la funcionalidad incorporada de un marco de trabajo, creando instancias de clases y llamando a sus funciones miembro. Los clientes extienden y modifican la funcionalidad de un marco, derivando en nuevas clases y reemplazando las funciones miembro.

Finalmente, Paradkar (2011) propone una lista de los marcos de trabajo más populares en el mundo del desarrollo de *software*, a saber:

- Microsoft Foundation Classes - MFC
- Java's Abstract Window Toolkit – AWT
- ACE, un marco de trabajo orientado en objetos
- Reusable Objects (ORO), un marco de trabajo de *software* libre
- Webridge Private Exchange, un marco de trabajo diseñado para construir aplicaciones B2B (Bussiness to Bussiness)

2.3. Programación extrema

Dudziak (1999) estableció que la programación extrema es tanto un proceso de desarrollo de *software* como una metodología. A continuación se revisará la acepción de esta metodología.

La programación extrema es un marco de proceso (define quién está haciendo qué, cuándo y cómo, por lo que proporciona principios, técnicas y prácticas para producir *software* eficiente, predecible y repetible) que puede adaptarse a las necesidades específicas de equipos, proyectos, organizaciones, etcétera. El proceso de programación extrema, en especial, está diseñado para que se pueda cambiar la dirección del proyecto en cualquier etapa.

La programación extrema es también una metodología de la categoría denominada "Cristal", a la que se le atribuyen características como alta productividad y tolerancia. La comunicación es fuerte en caminos cortos e informales (no documentados), y con frecuencia se entregan pequeños nuevos productos mediante procesos donde se establecen roles y actividades.

Conceptos básicos

Las cuatro variables

La programación extrema establece que un proyecto de desarrollo de aplicaciones es un sistema determinado por cuatro variables de control:

- *Costo*. La cantidad de dinero que se gasta para que estén disponibles los recursos.
- *Tiempo*. Determina cuándo debe hacerse el sistema.
- *Calidad*. Exactitud que requiere el sistema (definida por el cliente) y cómo se evaluará.
- *Alcance*. Describe qué y cuánto se hará (funcionalidad).

Los cuatro valores

La programación extrema define cuatro valores que pueden apreciarse en todas sus reglas y prácticas, a saber:

- *Comunicación*. Una buena comunicación es un factor necesario para lograr un proyecto de *software* exitoso. Los clientes necesitan comunicar sus requerimientos a los desarrolladores y éstos, a su vez, requieren comunicar sus ideas y diseños entre ellos mismos. La programación extrema trata de mantener la comunicación fluida mediante diversas formas. Casi todas sus prácticas se sustentan en la comunicación y la enfatizan al mismo tiempo.

- *Simplicidad*. La programación extrema se esfuerza por lograr programas simples que funcionen, así como para hacer la metodología simple. Reduce la cantidad de artefactos a un mínimo absoluto (requerimientos-historias de usuario, planes-plan de Juego y producto-código), y sus técnicas y prácticas se pueden aprender en cuestión de horas (aunque no su dominio). Esta simplicidad tiene como origen el hacer frente a los cambios y riesgos que

implica un proyecto de desarrollo de sistemas, de tal manera que apuesta por realizar las cosas lo más simple hoy para mañana cambiarlas con un poco más. Juntas, tanto la simplicidad como la comunicación trabajan mejor, puesto que entre más simple sea el sistema, más fácil será comunicarlo y viceversa: cuanto más se comunique, más fácilmente conocerá la información de la estructura del sistema. De esa manera, el sistema se torna simple porque se sabe más de él.

- *Retroalimentación*. La programación extrema es un proceso conducido por la retroalimentación. Se requiere retroalimentar en todas las escalas, ya sea la del cliente, el gerente o el desarrollador. Mientras se codifica, se recibe de inmediato la retroalimentación de las pruebas de caja blanca (pruebas de unidad). El cliente define las pruebas de caja negra (pruebas funcionales) y el equipo realiza entregas frecuentemente. Mediante estas prácticas, tanto clientes como desarrolladores reciben retroalimentación del estado del sistema. Esta retroalimentación cuenta con dos características importantes: primero la calidad, por ejemplo, no sólo se debe decir que algo está mal, sino también por qué y por qué no; la segunda característica es el tiempo, puesto que cuanto más temprana sea la retroalimentación, mejor.

- *Valentía*. Éste es un valor un tanto vago. Incluye coraje y cierta cantidad de agresividad. El primero es necesario porque muchas normas y prácticas de la programación extrema van contra la “tradición” de la ingeniería de *software*, incluido el papel del cliente. La segunda es una actitud necesaria para lograr la implementación del sistema.

El proceso de la programación extrema (XP)

XP es un proceso iterativo e incremental. El proyecto se divide en miniproyectos, lo cual se traduce en un incremento de la funcionalidad: la tan mencionada liberación. Una liberación es una versión del sistema planeado que tiene un sentido comercial. Todas las características que son parte de la liberación se implementan por completo. Un proyecto XP crea liberaciones frecuentes (cada uno o tres meses), a fin de obtener retroalimentación pronto y frecuente. Estas liberaciones de manera incremental construyen la deseada funcionalidad (pues el sistema crece con el tiempo).

Las liberaciones son negociadas en el plan de juego, ya sea que el cliente defina qué debe ser parte de las liberaciones y los desarrolladores establezcan el tiempo que tomará implementar dichas liberaciones o, bien, que los clientes establezcan las fechas y los desarrolladores establezcan qué cantidad de trabajo se logrará en ese tiempo. Cada ciclo de liberación se constituye de un par de iteraciones, cada una de las cuales no excede las tres semanas.

Plan de juego

Es el documento en el que se organizan de manera cronológica el orden y la entrega de las historias de usuario. El plan de juego consiste en los siguientes pasos:

- **Exploración.** El cliente escribe historias de usuario en tarjetas indizadas que definirán lo que quiere que haga el sistema. Las historias deben contar con un nombre corto y uno o dos párrafos de texto, evitando en lo posible los detalles técnicos. Puesto que para cada historia de usuario se calcula el tiempo necesario para desarrollarla y los riesgos que implica, es necesario que sean lo suficientemente detalladas para poder estimar dicha información fácilmente. Cada historia debe estar lista idealmente en máximo tres semanas de tiempo de ingeniería, pues de no ser así, debe descomponerse

en historias más cortas; pero si fuera lo contrario, deberá combinarse con otras historias cortas. Si los desarrolladores no entienden lo suficientemente bien, entonces de manera discreta investigarán más, creando una o dos soluciones de punta para ganar más conocimiento. Estas soluciones de punta son pequeños prototipos que solamente inspeccionan el problema; tales prototipos se utilizan también para experimentar otras cuestiones (como desempeño) o para escoger diferentes estrategias de solución.

- **Planeación.** Una vez que se tiene el conjunto de historias estimadas, se está en posición de realizar una reunión para agendar compromisos, donde se negociará qué conjunto de historias estimadas se implementarán en la liberación próxima. Para ello, los clientes entregan por su parte tres pilas de historias: las necesarias para liberar la función, las historias valiosas para la empresa y las historias que les gustaría desarrollar. Por su parte, el equipo de desarrollo establece el “factor de peso” o “factor unidad XP”, que describe la relación entre el número real de días del calendario necesarios para la tarea y el número de días de ingeniería real estimados para la tarea. Con ambos elementos, las pilas de historias y el factor de peso, se determina la agenda de compromisos. El alcance de la liberación se elige por el cliente de dos posibles maneras: 1) la liberación deberá hacerse en una fecha específica o, bien, 2) el cliente solicita que se liberen algunas tarjetas específicas.

- **Dirección.** La dirección significa la influencia en el proceso mediante pequeños movimientos (como mantener el carro en la dirección que uno quiere). Se puede dar a través de cuatro posibles movimientos:
    - Iteración. Las iteraciones constituyen el desarrollo.
    - Recuperación. Cuando el equipo de desarrollo se da cuenta de que no puede cumplir con la agenda, tiene el derecho (y la responsabilidad) de pedir al cliente una renegociación de la agenda de compromisos, ya sea modificando la fecha de liberación o la disminución del alcance (menos historias).
    - Nueva historia. El cliente tiene el derecho de adicionar nuevas historias, las cuales serán estimadas y –si el cliente decide que deben ser parte de la liberación– renegociadas en la agenda de compromisos.
    - Re-estimación. Cuando el equipo de desarrollo considera que la agenda de compromisos no es exacta, entonces se re-estiman las historias restantes y se renegocia la agenda de compromisos.

- **Desarrollo.** Una vez que se determina liberar el plan de juego, éste debe implementarse. Esto se realiza mediante un ciclo interno. Para ello, primero la agenda de compromisos se divide en iteraciones de 2 a 3 semanas (una semana de tiempo ideal de ingeniería), de tal manera que se conforma un plan de juego de iteración. Es importante considerar que primero se hacen las historias más valiosas para no generar conflictos entre las partes, puesto que esas historias son las más valiosas para el negocio.

Plan de juego de iteración

Cada iteración comienza con un plan de juego de iteración. Cabe precisar que la planeación de una iteración en particular es generada al inicio de la misma, por lo que ninguna iteración se planea por adelantado. Si algo es importante para una iteración posterior, se señala en una lista de tareas pendientes, de manera que esté

disponible cuando se planifique dicha iteración. Generalmente, cada tres iteraciones la agenda de compromisos se actualiza para reflejar el estado del proyecto, donde son especialmente importantes los datos de nuevos riesgos y la refinación del factor de peso, por lo que es posible que sea necesaria una nueva planificación para renegociar la agenda de compromisos.

De manera similar al plan de juego, el plan de juego de iteración consiste en tres fases:

- **Exploración.** Desde la parte más destacada de la agenda de compromisos el cliente elige las historias a realizar en la iteración. Generalmente selecciona las historias más valiosas (y dejando probablemente las más riesgosas), revisando además aquellas historias que no han superado las pruebas funcionales. Posteriormente los desarrolladores “separan” las historias en tareas de ingeniería, las cuales son generalmente más pequeñas que las historias. Algunas veces una tarea resulta de más de una historia o, bien, no se relaciona con una historia en particular. Es una buena práctica escribir las tareas en tarjetas indexadas, similares a las que se utilizan para las historias.

- **Planeación.** Un desarrollador acepta la responsabilidad de una tarea. Las tareas no son asignadas, sino elegidas de manera voluntaria. Una vez seleccionada la tarea, el desarrollador responsable estima el tiempo ideal de ingeniería para la tarea. Sólo el desarrollador que es responsable de la tarea, realiza la estimación. Una tarea típica puede tomar entre medio día y tres días de tiempo ideal de ingeniería. Si la tarea es corta, se combina con otras tareas cortas (sólo en términos de determinar la agenda, no el contenido de la tarea). Si la tarea es

más larga, puede descomponerse en tareas más pequeñas de ser posible. Hay que considerar que las áreas derivan solamente de las historias asignadas para la iteración, por lo que no hay posibilidad de codificar por adelantado otras historias. Las tareas, junto con el nombre de los desarrolladores responsables y su estimación, conforman la agenda de iteración. Con el factor de peso actual (y refinado de la iteración previa), el equipo determina si la iteración está sobreestimada o subestimada: si fue sobreestimada, el cliente tiene que mover historias de la actual iteración a una posterior; si fue subestimada, el cliente puede seleccionar más historias para adicionar a la iteración. Finalmente, el factor de peso se balancea, por lo que si un desarrollador está sobrecargado, deberá renunciar a algunas de sus tareas.

- **Dirección.** En esta fase se realiza la codificación. Como parte de ella se pueden llevar a cabo los siguientes movimientos:
    - Implementación. Se implementa una tarjeta de tarea.
    - Registro del progreso. Mediante el *tracker* se visualiza quién tiene asignada dicha función.
    - Recuperación. Cuando un desarrollador se encuentra sobrecargado se debe realizar una o más de las siguientes acciones:
        - Obtener ayuda de otro compañero o hacer una sesión CRC (descrita adelante) para lograr mayor comprensión
        - Reducir la cantidad de trabajo, ya sea transfiriendo tareas a otros desarrolladores o reprogramándolas para una siguiente iteración).
        - Solicitar al cliente una reducción del alcance de la iteración (pasando historias a la siguiente iteración).
    - Verificación. Cada historia tiene asociadas pruebas funcionales. Tan pronto como las pruebas estén listas y todas las áreas necesarias sean implementadas, se corren las pruebas para verificar que las historias funcionan.

Al final de cada iteración, todos los casos de pruebas funcionales hechos para las historias de una iteración son revisados por el cliente. Si alguna prueba falla, entonces la historia asociada con dicha prueba será parte de la siguiente iteración también.

La metodología de programación extrema. Reglas y prácticas

Todas las reglas y prácticas trabajan mano a mano y la fortaleza de XP radica en la combinación de ellas. Éstas se agrupan en:

Reglas y prácticas de gestión

Métricas. XP define dos métricas utilizadas para describir el progreso del proyecto: el factor de peso, que describe qué tan "bien" está trabajando el equipo; y el resultado del conjunto de pruebas funcionales. Esta métrica describe (dentro de los límites) el progreso del proyecto.

Roles. XP define los siguientes roles:

- *Cliente.* Define qué hacer (historias de usuario) y en qué orden (plan de juego). Existe un rol especial denominado "cliente en sitio", conocido como un representante del cliente (parte del sitio de desarrollo) disponible para refinar las historias de usuario, preparar las pruebas funcionales, etcétera.

- *Programador*. Elemento que realiza el trabajo de desarrollo del *software*. Entre las habilidades que debe poseer están:
    - Comunicación. Especialmente cara a cara.
    - Codificación. Ser un buen programador, no un genio.
    - Habilidad para trabajar en equipos. Básica en esta metodología que es orientada al equipo, sobre todo por la propiedad colectiva del código y la programación en parejas.

- *Coach*. La gestión de un proyecto de XP se realiza por dos roles: el *coach* y el *tracker*. El primero es responsable de la ejecución técnica y evolución del proyecto y participa en las reuniones de gestión del mismo. Debe de saber establecer buena comunicación, ser firme, hábil a nivel técnico y seguro. Su trabajo es lograr que todos tomen las mejores decisiones y sus deberes son:

  - Explicar el proceso a la administración y a los clientes.
  - Aportar habilidades técnicas para probar, formatear y refactorizar.
  - Tener una visión general del sistema –en particular de las metas de refactorización a largo plazo.
  - Estar disponible como un socio de desarrollo (de la programación en pares), especialmente para los nuevos miembros del equipo.

- *Tracker*. Su trabajo es reunir las métricas para dar seguimiento al proyecto, sobre todo de los factores de peso y resultados de las pruebas funcionales. Solicita a cada desarrollador cada dos o tres días la cantidad de tiempo que ha pasado en cada una de sus tareas y cuánto tiempo le falta para finalizar, de tal modo que pueda contar con las mediciones que le permitan determinar si se puede cumplir o no la agenda de compromisos, comunicándolo en las reuniones de gestión. Debido a que éste no es un trabajo de tiempo completo, puede desempeñarlo el *coach* o un desarrollador.

- *Probador*. Ayuda al cliente a elegir y escribir las pruebas funcionales y a ejecutarlas. Un desarrollador o el *tracker* puede realizar esta función.

- *Consultor*. Es contratado para aportar sus conocimientos, compartiéndolos con los demás miembros del equipo –sobre todo en el aspecto técnico–, con el fin de resolver problemas.

Área de trabajo y herramientas. El área de trabajo se constituye por una habitación grande donde puedan trabajar todos juntos. La habitación contará con pequeños cubículos a los lados (para las reuniones CRC) y computadoras al centro (una de ellas, dedicada a la integración). Dentro de las herramientas se requieren aplicaciones para gestionar las continuas integraciones y configuraciones. Asimismo, se requiere una aplicación que facilite las pruebas continuas.

Reunión parada. Se realiza diariamente, dura un par de minutos y participan todos los miembros del equipo, describiendo lo que están trabajando, cómo van y cosas interesantes. Sirve para anunciar descubrimientos interesantes o encontrar socios para la programación en pareja.

Cuarenta horas a la semana. El proceso hace especial énfasis en este aspecto, a fin de evitar que el equipo trabaje horas extras, sobre todo por dos cosas: porque los proyectos que requieren horas extras ya van atrasados de cualquier manera y porque se afecta la motivación de los desarrolladores.

Reglas y prácticas de desarrollo

Ciclo de desarrollo. La implementación ocurre en pequeños pasos (tareas), para los cuales XP define el siguiente procedimiento –que toma un par de horas o máximo un día para completarse:

- Analizar lo que se debe hacer, ya sea que implique analizar las áreas de ingeniería y/o las historias de usuario, y realizar una reunión CRC si es necesario.

- Escribir pruebas de unidad que ayuden a encontrar interfaces y determinar si la tarea está completa.
- Implementar sólo el código suficiente para que la prueba sea superada.
- Simplificar el código si es necesario.
- Integrar los cambios en la base de código. Si hay problemas al integrar, deben resolverse o, de otro modo, habrá que empezar de nuevo.

Integración continua. Se refiere a la actividad en la que los cambios en el sistema se combinan con el código base. Cuando una tarea está terminada (adicionar una característica, corregir un error, crear una unidad de prueba) y las pruebas unitarias son realizadas al 100%, los cambios son integrados inmediatamente. Puesto que las tareas toman generalmente un día, la integración se produce constantemente, formando parte del ciclo de desarrollo. Los beneficios que reporta esta actividad son:

- La integración es más fácil porque los cambios son pequeños.
- Las pruebas de unidad funcionan correctamente antes de la integración.
- La base de código representa siempre el estado actual del sistema.

Código de propiedad colectiva. El código (clase, etcétera) creado en un proyecto XP es propiedad de todo el equipo, no de los desarrolladores en lo individual, lo cual permite que cualquier persona sea capaz de modificar cualquier cosa que sea necesaria, por lo que se vuelve un elemento fundamental del ciclo de desarrollo, especialmente para la refactorización implacable.

Programación en pares. Cualquier producción de código en un proyecto XP es realizada por un par de desarrolladores trabajando juntos en una computadora. Uno codifica y el otro revisa constantemente el código, roles que constantemente intercambian (generalmente al finalizar una tarea).

Estándares de codificación. Los proyectos de XP utilizan reglas y lineamientos (estándares de código) para nombrar y formatear unidades de código, ya sean externos o definidos por el equipo de desarrollo. Los estándares hacen más consistente al sistema, de manera que sea más fácil leerlo, comprenderlo, trabajarlo y aprenderlo.

Cliente en sitio. Puesto que en todas las fases de XP la comunicación con el cliente es necesaria, también es fundamental contar con un cliente en el equipo de desarrollo, de tal manera que siempre esté disponible para proporcionar la información necesaria para implementar la historia de usuario o su refinamiento.

Evaluación implacable. Todas las prácticas de XP descansan en la red segura creada por las pruebas frecuentes, las cuales aseguran que el sistema permanece intacto después de los cambios y que se mueve en la dirección que el cliente desea. Esta evaluación en XP se realiza en dos niveles: las pruebas de unidad a las que corresponden las pruebas de caja blanca y caja gris –que evalúan los métodos y clases que pueden descomponerse– y las pruebas funcionales a las que corresponden las pruebas de caja negra –que verifican que el sistema en producción cumple con los requerimientos del cliente.

No lo necesitarás (You Are Not Gonna Need It [YAGNI]). YAGNI es una regla de diseño importante de XP: consistente en evitar adicionar a las áreas características que el desarrollador considera que “podrían” ser necesarias en el futuro, y cumplir específicamente con la tarea asignada.

Refactorización implacable. Una refactorización es una transformación al sistema preservando su comportamiento (externo), que generalmente implica cambios en los nombres, en el orden de las unidades de código (métodos y clases) y sus dependencias. Estas transformaciones se encuentran respaldadas por las pruebas de unidad. La refactorización creará más clases con menos responsabilidades (menos campos y métodos), distribuidas de una manera más comprensible. El término “implacable” hace referencia a que si existe una oportunidad de simplificar la estructura, se debe aplicar la refactorización, por lo que ésta sólo trabaja bien cuando se combina con la evaluación implacable, que asegura que los cambios no afectarán el comportamiento.

Sesión CRC (Class Responsibility Collaborator) [Colaboración de Responsabilidad de Clase]. Es un método para visualizar y determinar el diseño de un sistema que se basa principalmente en el uso de tarjetas indexadas, que representan una clase dentro del sistema. Las tarjetas se organizan para determinar la interacción entre clases (un par a la vez) y no constituyen una técnica de documentación.

Metáfora del sistema. Los sistemas se construyen alrededor de una metáfora o un conjunto de ellas que cooperan entre sí, las cuales se utilizan como base para el diseño del sistema y como una guía esquemática para nombrar clases, métodos, etcétera. Las metáforas deben seleccionarse cuidadosamente, ya que si hay una situación en la que la metáfora no encaja, deberá ser mejorada o sustituida. Si fuera esto último, requerirá probablemente de una refactorización o incluso un nuevo plan de juego.

Optimización “vaga”. La optimización se realiza lo más tarde posible, puesto que esta metodología no busca identificar los cuellos de botella de los sistemas, sino medirlos.

2.4. Fases del desarrollo rápido de aplicaciones (RAD)

De acuerdo con la “Introducción al desarrollo rápido de aplicaciones”, elaborada por la oficina del Jefe de Información del Gobierno de Hong Kong (2009), el desarrollo rápido de aplicaciones (Rapid Application Development [RAD]) se refiere a un ciclo de vida de desarrollo de sistemas diseñado para hacer el desarrollo más rápido y lograr sistemas de mayor calidad. Está diseñado para tomar ventaja de los programas de desarrollo de sistemas más poderosos, como las herramientas de Computer Aided Software Engineering (ingeniería asistida por computadora), CASE, las herramientas para prototipos y los generadores de código. Los objetivos clave de RAD son: mayor velocidad, mayor calidad y bajo costo.

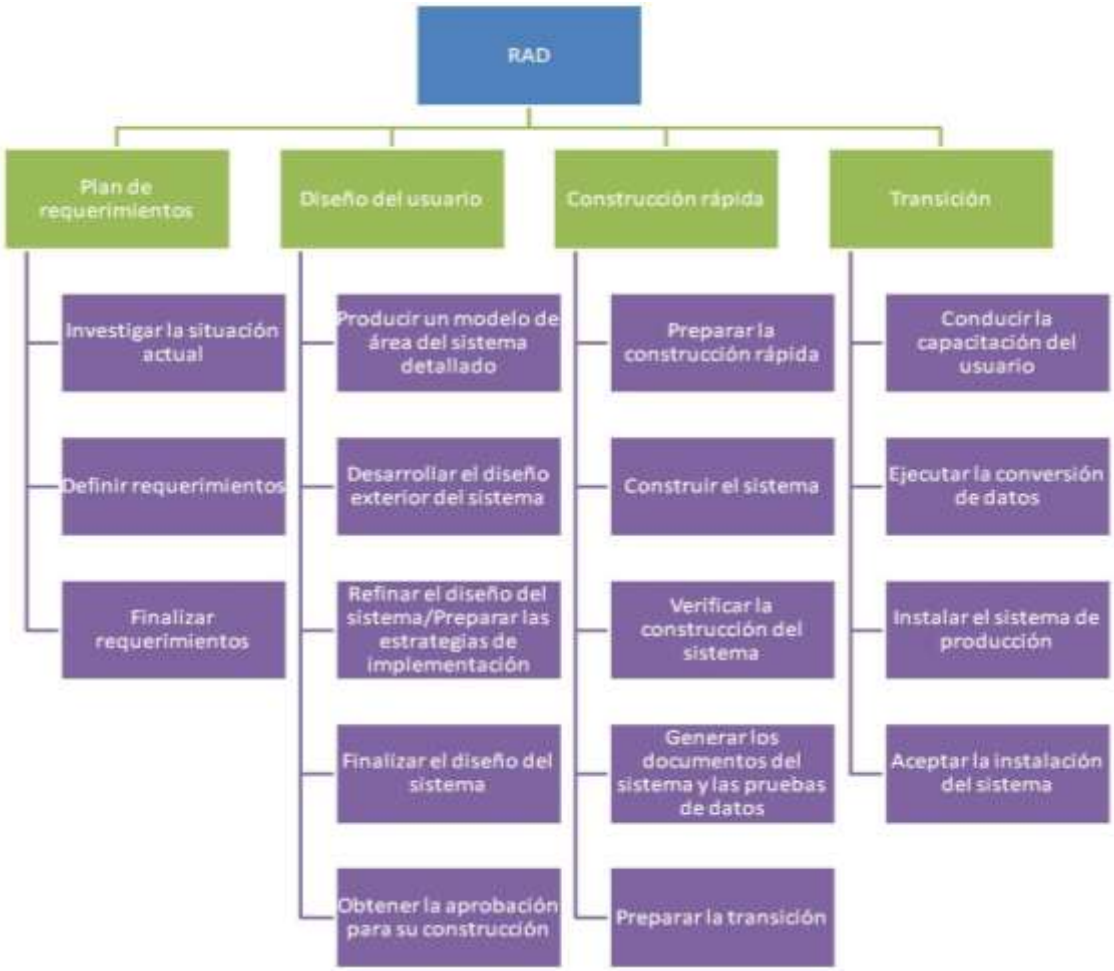
Asimismo, también se establece que RAD es una propuesta de desarrollo incremental centrada en las personas, donde es imprescindible la participación activa de los usuarios, así como la colaboración y cooperación entre todas las partes. Las pruebas son integradas a lo largo del ciclo de vida del desarrollo, de manera que el sistema es probado y revisado incrementalmente por desarrolladores y usuarios.

La estructura de RAD se divide en etapas y cada etapa, en tareas. Para cada tarea se definen claramente los objetivos, las entradas, salidas, técnicas a utilizar, roles involucrados y una lista de las subtareas que se deben realizar.

El ciclo de vida típico de RAD se compone de las siguientes etapas:

- Plan de requerimientos
- Diseño del usuario
- Construcción rápida
- Transición

Ilustración 1



Elaborada con base en Office of the Government Chief Information Officer (2009).

Plan de requerimientos

Consiste en una revisión de las áreas más relacionadas con el sistema propuesto. Las opiniones recabadas producirán una definición amplia de los requisitos del sistema en término de las funcionalidades que debe soportar. Se desarrolla también un esquema del área del sistema y su alcance. La junta del proyecto (ejecutivos del negocio, usuarios finales y personal de sistemas) participa en talleres que avanzan a través de una serie de pasos estructurados, cuyos resultados son registrados en una herramienta CASE.

Los entregables de esta fase incluyen un modelo de área del sistema delimitado (del área bajo estudio), una definición del alcance del sistema y una justificación del costo del nuevo sistema.

Las tareas de esta etapa son:

- Investigar la situación actual. Se lleva a cabo una investigación del entorno actual para preparar la definición de los requisitos del sistema propuesto. Se crea en el repositorio CASE un modelo del área del sistema inicial.

- Definir requerimientos. En esta tarea se definen tanto el modelo del área del sistema como el ámbito del mismo. La funcionalidad del sistema se expresa en términos de los procesos del negocio y los datos que el sistema soportará. Asimismo, se identifican beneficios potenciales, costos y riesgos asociados con el sistema propuesto. También se documentan cuestiones de gestión, tales como restricciones y limitaciones que afectarán el desarrollo subsecuente.

- Finalizar requerimientos. Se documenta formalmente el alcance del sistema propuesto. Se prepara un estimado de los recursos y la duración para implementar el sistema, y se obtiene la aprobación para proceder con el desarrollo de la aplicación.

Diseño del usuario

Consiste en un análisis detallado de las actividades y datos del negocio, relacionados con el sistema propuesto. Los usuarios “clave”, reunidos en talleres, descomponen las funciones del negocio y definen los datos asociados con el sistema. Completan el análisis definiendo las interacciones entre procesos y datos. Los resultados de los talleres son registrados en la herramienta CASE.

Después del análisis se delimita el diseño del sistema. Los procedimientos del sistema son diseñados y desarrollados mediante los diseños preliminares de pantallas e informes. Se construyen y revisan los prototipos de procedimientos críticos. Se selecciona una propuesta de construcción apropiada para el sistema y se prepara el plan de implementación del sistema.

Las tareas en esta etapa son:

- Producir un modelo de área del sistema detallado. Se conducen los talleres para el diseño conjunto de la aplicación (JAD: Joint Application Design Workshops) para completar el análisis de las actividades del negocio y los datos asociados con el sistema propuesto, a fin de producir un modelo de área del sistema detallado. Se revisa y refina el alcance del sistema a desarrollar, con el fin de asegurar que las funciones críticas del sistema se entregarán en el periodo solicitado. Se revisan y exploran los detalles de las definiciones de actividades del negocio y sus datos asociados, se establece cómo se aplicarán las reglas en cada actividad del negocio y los atributos de cada entidad.

- Desarrollar el diseño exterior del sistema. Se identifican los procedimientos requeridos por el sistema y se desarrollan esquemas tentativos para pantallas y reportes. Se desarrolla el diseño exterior del sistema, identificando las interacciones entre los procedimientos del sistema y los datos, es decir, se establece el uso de datos por cada función del sistema.
- Refinar el diseño del sistema. Se revisa y verifica la integridad del modelo de área del sistema detallado y del diseño exterior del sistema. Se confirma la consistencia del análisis mediante un análisis a la interacción del prototipo. Se analizan mediante un análisis de interacción todas las interacciones entre las funciones del sistema y los datos, para identificar funciones perdidas/extrañas. Una vez resueltas las inconsistencias, se desarrollan los prototipos de pantallas y diálogos para que los usuarios las revisen. También se hacen los ajustes a la lista de cuestiones abiertas.
- Preparar las estrategias de implementación. La propuesta de implementación se selecciona después de revisar el diseño del sistema. Después se prepara un plan de implementación, enlistando todas las áreas que deben ejecutarse para desarrollar el sistema y lograr su uso operacional. También se hace un estimado del esfuerzo necesario para completar cada tarea y se suma todo en un estimado global del costo del proyecto.
- Finalizar el diseño del sistema. En esta etapa deben estar finalizados el diseño exterior del sistema y el plan de implementación. Se discuten y resuelven cuestiones de diseño pendientes, excepto aquellas que no impactan en el diseño del sistema, tales como las cuestiones culturales. Se presenta, discute y confirma el plan de transición para el sistema.
- Obtener la aprobación para su construcción. Se incorporan los resultados del taller JAD final al diseño del sistema y al plan de transición. Se solicita la autorización para proceder con la siguiente etapa: construcción rápida.

Construcción rápida

En esta etapa, un pequeño equipo de desarrolladores conocido como equipo SWAT (Skilled With Advanced Tools [expertos con herramientas avanzadas]) trabaja directamente con los usuarios, finaliza el diseño y construye el sistema. El proceso de construcción de *software* consiste en una serie de pasos “diseña y construye”, mediante los cuales los usuarios tienen la oportunidad de afinar los requerimientos y revisar la implementación del *software* resultante. Se prepara también el plan de transición para la producción. Los entregables de esta etapa incluyen la documentación e instrucciones necesarias para operar la nueva aplicación, retinas y procedimientos necesarios para poner al sistema en operación.

Las tareas de esta etapa de construcción rápida son:

- Preparar la construcción rápida. El ambiente que se desarrolla ha finalizado. La base de datos es diseñada con base en la estructura de datos preliminar desarrollada en la etapa de diseño de usuario. Se conforman los equipos de informáticos y usuarios, quienes construirán el sistema. Se revisan los estándares que se aplicarán en el diseño final y en la construcción del sistema.
- Construir el sistema. Es decir, el *software* para implementar el sistema es desarrollado y completado por el equipo SWAT con un límite de tiempo establecido. Un equipo de asistencia para la construcción trabaja junto con el equipo SWAT para desarrollar el *software*, utilizando una técnica de prototipos iterativos. El código resultante es probado en el sistema y a través de todos los componentes del sistema asignados al equipo SWAT.

- Verificar la construcción del sistema. Se verifica y confirma cada componente del sistema y las actividades del sistema por completo, de acuerdo con los requerimientos del usuario.
- Generar los documentos del sistema y las pruebas de datos. Este paso consiste en desarrollar las pruebas de datos necesarias para verificar la capacidad operacional del sistema. Estos datos serán utilizados durante las pruebas de integración, del sistema y de aprobación. Se produce la documentación que explica cómo debe operarse el sistema por los usuarios y por el personal operativo de informática.
- Preparar la transición. Se prepara un plan de trabajo detallado para las actividades de transición y un plan de contingencia para atender cualquier falla en la aplicación. Es desarrollado el *software* necesario para convertir los datos existentes, y se desarrollan los procedimientos requeridos para ejecutar la transición al sistema que se está desarrollando. Se resuelve cualquier aspecto organizacional relacionado con el despliegue de la nueva aplicación. El trabajo de preparación para la transición es llevado a cabo durante la etapa de construcción rápida.

Transición

Implica la implementación del nuevo sistema y la gestión del cambio del entorno del viejo sistema al entorno del nuevo sistema, por lo tanto, puede incluir la implementación de puentes entre clientes nuevos y existentes, así como la conversión de datos para el nuevo sistema, el entrenamiento de usuarios para operar la nueva aplicación y proporcionar soporte para resolver cualquier problema que surja de manera inmediata, después de que la aplicación se vuelve operacional. La aceptación del usuario es el punto final de la etapa de transición.

Las tareas para esta etapa son:

- Conducir la capacitación del usuario. Se llevan a cabo sesiones de capacitación para instruir a los futuros usuarios del nuevo sistema y la forma en que opera. El entrenamiento se completa antes de que la aplicación se coloque en el ambiente de producción.
- Ejecutar la conversión de datos. La información necesaria para la operación del nuevo sistema se convierte a partir de fuentes de datos existentes en un formato accesible para el nuevo sistema. Los datos convertidos son “cargados” en las estructuras de datos asociadas con el sistema.
- Instalar el sistema de producción. Consiste en los pasos necesarios para iniciar el funcionamiento del sistema en el entorno de producción. Se completan los ajustes necesarios en el *hardware* y en la configuración del *software*, las instrucciones dadas al personal operativo que operará el sistema y las librerías de *software* cargadas con las versiones de producción de la aplicación.
- Aceptar la instalación del sistema (Accept System Installation). Se considera aceptada la nueva instalación del sistema cuando éste opera por un periodo determinado, dentro de las tolerancias definidas por el rendimiento, la tasa de error y la usabilidad. Esta aceptación se basa en acuerdos entre los usuarios, el personal de producción/operaciones, el personal de soporte, así como la gestión de la información para la organización relacionada con agendas, procedimientos, políticas de precios, acuerdos de garantía y documentación de sistemas y *software*.

RESUMEN

En esta unidad se abordaron los diversos elementos que permiten la conformación de un modelo de implementación para las aplicaciones informáticas. Dentro de estos elementos se encuentran los patrones de diseño, que aportan soluciones ya probadas a problemas de diseño de *software* específicos. También se estableció que los patrones de diseño constan de cuatro elementos esenciales: nombre, problema, solución y problema, y se revisaron los pasos para seleccionar y utilizar un patrón de diseño, finalizando con una presentación genérica de los tipos de patrones de diseño existentes y la intención de cada uno.

Otro elemento revisado fue el de los marcos de trabajo, considerados como un conjunto de componentes que son comunes a un determinado dominio de aplicación. De éstos se presentó su clasificación (caja negra, caja blanca y ambos tipos de cajas), las dimensiones que los caracterizan (dominio del problema, estructura interna y cómo se utilizará) y una lista de los marcos de trabajo más populares.

Asimismo se presentó la programación extrema, identificándose los conceptos básicos que la sustentan (las variables y valores), el proceso mediante el cual se desarrolla (plan de juego, iteración), y las reglas y prácticas de la misma programación extrema. Por último se abordó el denominado “desarrollo rápido de aplicaciones” (Rapid Application Development [RAD]), presentándose todas y cada una de las etapas que lo componen, así como las tareas que se desarrollan en cada fase.

BIBLIOGRAFÍA

Bibliografía básica

Dudziak, T. (1999). Extreme programming. An overview. *Methods and tools for software production WS, 2000*: 1-28. Consultado el 29 de mayo de 2015, disponible en: https://www.google.com.mx/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0CCEQFjAA&url=http%3A%2F%2Fcsis.pace.edu%2F~marchese%2FCS616%2FAgile%2FXP%2FXP_Overview.pdf&ei=j_oiVJjZJ8WeyATx7ILACA&usg=AFQjCNHxDhn-dhquADAnHjGQSB7xkBr7_A&sig2=tQLvgojH_DCDacRpu4g9mQ

Gamma, E., Helm, R., Johnson, R. y Vlissides, J. Gang of Four. (1995). *Design Patterns. Elements of Reusable Object-Oriented Software*. Estados Unidos: Adisson-Wesley. Consultado el 29 de mayo de 2015, disponible en: http://www.uml.org.cn/c++/pdf/DesignPatterns.pdf

Office of the Government Chief Information Officer (2009). *An Introduction to Rapid Application Development.* Consultado el 29 de mayo de 2015, disponible en: http://www.ogcio.gov.hk/en/infrastructure/methodology/rad/doc/g47a_pub.pdf

Taligent Inc. (1994). *Building Object-Oriented Frameworks*. *A Taligent White Paper.* Consultado el 29 de mayo de 2015, disponible en: http://lhcb-comp.web.cern.ch/lhcb-comp/Components/postscript/buildingoo.pdf

Referencias bibliográficas

Cáceres, J. (2009). Patrones de diseño: ejemplo de aplicación en los Generative Learning Object. *Revista de Educación a Distancia* [número especial dedicado a Patrones de eLearning y Objetos de Aprendizaje Generativos]. Consultado el 29 de mayo de 2015, disponible en: http://www.um.es/ead/red/M10/caceres.pdf

Paradkar, S. (2011). *The Anatomy of Software Frameworks. BPT Trends, Abril*. Consultado el 29 de mayo de 2015, disponible en: http://www.bptrends.com/publicationfiles/04-05-2011-ART-Anatomy%20of%20Software%20Frameworks-Paradkar-final.pdf

Riehle, D. y Gross, T. (1998, octubre). Role model based framework design and integration. *ACM SIGPLAN Notices*, *33* (10): 117-133. Association for Computing Machinery. Consultado el 29 de mayo de 2015, disponible en: http://dirkriehle.com/computer-science/research/1998/oopsla-1998.pdf

Sitios electrónicos

Sitio	Descripción
http://www.oodesign.com/	Patrones de diseño. Sitio que explica a profundidad los patrones de diseño.
http://xprogramming.com/index.php	Xprogramming. Sitio que contiene recursos para XP.
http://softwaresolution.informer.com/Free-RAD-Tools/	Free RAD (Rapid Application Development) Tools Download. Sitio que presenta diversas herramientas gratuitas para implementar RAD.
http://users.dsic.upv.es/asignaturas/facultad/lsi/ejemploxp/Gestion_Proyecto.html#planificacion	Ejemplo de desarrollo de *software* utilizando la metodología Extreme Programming (programación extrema).
http://goo.gl/Tnvaoy	Ejercicio guiado de análisis y diseño orientado a objetos.

UNIDAD 3

Plan de implementación

OBJETIVO PARTICULAR

Planear la implementación de subsistemas.

TEMARIO DETALLADO
(10 horas)

3. Plan de implementación

3.1. Definición de objetivos
3.2. Estimación de tareas y tiempos
3.3. administración de la configuración
3.4. Administración de cambios
3.5. Modelo de la arquitectura propuesta

INTRODUCCIÓN

El primer tema de la unidad, denominado “Definición de objetivos”, aborda la importancia que reportan los objetivos para los proyectos, así como las herramientas para lograr una buena definición de los mismos objetivos.

Posteriormente, el tema “Estimación de tareas y tiempos”, establece que la estimación es una tarea importante que permite lograr una planificación del proyecto adecuada, al predecir el esfuerzo requerido para el desarrollo de un sistema, y de la cual se presentarán algunos métodos para desarrollarla y los factores generales que la determinan. En el tema “Administración de la configuración” se revisará qué es el conjunto de actividades que se desarrollan para gestionar los cambios a lo largo del ciclo de vida del desarrollo de un sistema, abordándose también los elementos conceptuales que la soportan (como la línea base y los elementos de configuración del software), así como el proceso mismo de la administración de configuración. El tema “Administración de cambios” será más específico en aspectos concretos de la actividad. Finalmente, en el tema “Modelo de la arquitectura propuesta”, la cual se entiende como la organización fundamental de un sistema encarnada en sus componentes, las relaciones entre ellos y el ambiente, y los principios que orientan su diseño y evolución, se abordarán los diversos estilos arquitectónicos que se han desarrollado y los beneficios que reporta la arquitectura de software.

3.1. Definición de objetivos

De acuerdo con Guerra y Bedini (2005: 5), “Un proyecto es una asociación de esfuerzos, limitado en el tiempo, **con un objetivo definido**, que requiere del acuerdo de un conjunto de especialidades y recursos. Cuando los objetivos de un proyecto son alcanzados se entiende que el proyecto está completo”.

Reconociendo entonces, de acuerdo con la definición anterior, que los proyectos tienen un objetivo definido, se establece qué es un objetivo. De acuerdo con la Universitat Oberta de Catalunya, un objetivo es la expresión de un fin “que se quiere conseguir y que debe permitir la articulación de una serie de acciones encaminadas a su consecución” (UOC, 2003).

Los objetivos sirven, entre otras cosas, para formular los resultados deseados de manera concreta y objetiva; para planear las acciones a realizar, para dirigir los procedimientos y para cualificar o cuantificar los resultados.

El blog de la herramienta para gestión de proyectos en línea Dolphy (2009), señala que es importante establecer objetivos porque…

- “Ayudan a acotar el alcance real del proyecto y son la base de la toma de decisiones durante la ejecución del proyecto.”
- “Su validación con el cliente (ya sea interno o externo) incrementa la capacidad de satisfacción de sus expectativas.”

- “Permite involucrar a todas las partes interesadas y miembros del proyecto en base a unas metas claras, por lo que estos objetivos deben ser comunicados.”
- “Es una base fundamental para la planificación del proyecto, ya que permite definir con acierto las tareas a realizar, en cuánto tiempo y por parte de quién. Además, una vez que los objetivos están priorizados, es posible ordenar y distribuir los esfuerzos a realizar en base a dichas prioridades.”

- “Sin objetivos previos claros es imposible monitorizar la evolución del proyecto ni medir su éxito final” (Blog Dolphy, 2010: 1).”

Futrell, et ál. (2002), establecen que para que los objetivos de un proyecto sean claros deben ser:

- Enfocados en los procesos y en los resultados (que son los importantes para el cliente).
- Medibles y comprobables (en costo, porcentaje de mercado ganado, fechas).
- Orientados a la acción, es decir, que impliquen acciones a lograr.
- Fáciles de trasmitir, es decir, que puedan ser platicados y explicados en pocos segundos.
- Factibles, es decir, razonables, que no impliquen alcanzar una utopía.
- Comunicables; el equipo los conoce y están publicados en la documentación del proyecto.

Para desarrollar objetivos con estas características, el equipo de desarrollo debe comprender primero el problema a resolver. Para lograr ello, se puede apoyar de la herramienta “Es/No es”, la cual permite describir el problema de la mejor manera.

Esta herramienta es especialmente útil cuando hay comprensiones “parciales” del problema entre los miembros del equipo.

El uso de la herramienta “Es/No es” lleva a cabo los siguientes pasos:

1. Declarar el problema en términos claros y objetivos.
2. Describir todo lo que se sepa acerca del problema:
    a. ¿Qué es? / ¿Qué no es?
    b. ¿Dónde está? / ¿Dónde no está?
    c. ¿Cuándo es un problema? / ¿Cuándo no lo es?
    d. ¿Quién está involucrado? / ¿Quién no lo está?
    e. ¿Qué tan frecuente sucede el problema?
    f. ¿Qué tan grande es el problema?
3. Resolver las siguientes preguntas:
    a. ¿Cómo sabremos que hemos tenido éxito en solucionar el problema?
    b. ¿Cuáles son los indicadores que nos lo dirán?
4. Definir los límites del proceso, entradas y salidas.
    a. ¿Qué personas, departamentos, equipos están involucrados?
    b. ¿Cuáles son los límites del proceso?
    c. ¿Cuáles son las entradas y quiénes las suministran?
    d. ¿Cuáles son las salidas y quiénes las reciben?
5. Descubrir lo que los clientes esperan del proceso y su opinión respecto a la calidad, tiempo de respuesta, entrega a tiempo y costo.
6. Registrar las ideas del equipo en una hoja de trabajo que idealmente lleva el siguiente formato:

Declaración del problema			
¿Qué necesita mejorarse? ¿Qué está mal?			
	Es	No es	Información requerida
Qué (proceso, producto, servicio, objeto, falla o desviación)			
Dónde (localizable en productos, objetos o geográficamente)			
Cuándo (primer avistamiento, otras ocurrencias, etapa del proceso)			
Quién (personas involucradas en el proceso, clientes, proveedores)			
¿Qué tan frecuente ocurre el problema?			
¿Qué tan grande es el problema?			
Medidas de éxito. ¿Cómo sabremos que el problema ha sido resuelto?			

La hoja de trabajo tiene como finalidad clarificar el problema para que posteriormente se puedan establecer objetivos claros que permitan iniciar un proyecto de software exitoso.

La importancia de definir los objetivos de manera clara radica en tres aspectos principales. El primero es que el software desarrollado solucione efectivamente el problema y atienda las necesidades del usuario. El segundo es que evite retrasos durante las siguientes etapas. Finalmente, el tercero es que el producto sea susceptible de actualizarse o mejorarse.

Por lo tanto, una técnica útil para definir objetivos claros es el método SMART, sigla de *Specific* (específicos), *Measurable* (medibles), *Achievable* (alcanzables), *Realistic* (realistas) y *Time-Bound* (con límite de tiempo).

Para cualquier objetivo se deben determinar los resultados específicos y metas de desempeño necesarias para cumplir con las expectativas, para lo cual es válido apoyarse de los siguientes cuestionamientos:

- ¿Los objetivos son específicos?
- ¿El equipo de desarrollo y los clientes están de acuerdo con los resultados que deriven de los objetivos del proyecto?
- ¿Los objetivos son medibles?
- ¿Cómo saber que han alcanzado los resultados?
- ¿Qué significa para el equipo de desarrollo y para el cliente la frase “cumplir con las expectativas” respecto a cada uno de los objetivos?
- ¿Los objetivos son realistas y relevantes?
- ¿Los objetivos están alineados con los factores clave de éxito de la organización, las metas del negocio y las estrategias?
- ¿Los objetivos están limitados en tiempo?
- ¿Están declaradas fechas específicas en las cuales los objetivos deben alcanzarse?
- ¿Existe una razón comprensible y clara para definir cada una de las fechas?
- ¿Qué o quién está detrás de las fechas (cliente, la necesidad del producto, etc.)?

3.2. Estimación de tareas y tiempos

Como parte de las actividades de gestión del proyecto de software se encuentra la estimación. De acuerdo con Moreno (s. f.: 7).

> La estimación se define como el proceso que proporciona un valor a un conjunto de variables para la realización de un trabajo, dentro de un rango aceptable de tolerancia. Podemos definirlo también como la predicción de personal, del esfuerzo, de los costes y de la planificación que se requerirá para realizar todas las actividades y construir todos los productos asociados con el proyecto (Moreno, s. f.: 7).

Estimar el software es importante porque proporciona unión entre los conceptos generales del análisis económico y el mundo de la ingeniería de software.

Esteban y Dolado (2008) señalan que una estimación lo suficientemente exacta permite a la compañía desarrolladora de software una mejor planificación de los proyectos que maneja, así como una mejor asignación de los recursos necesarios para cada proyecto; además, permite valorar el impacto ante cambios en el plan de desarrollo del proyecto.

Los modelos de estimación del costo de desarrollo de software se basan en un conjunto de variables. Estas variables se denominan factores, como el costo, el esfuerzo, el tamaño, entre otros. Cada modelo basa sus estimaciones en un conjunto propio de factores que pueden estar relacionados entre sí. La naturaleza de las relaciones causales entre los factores implica cierta incertidumbre: no son deterministas. Esto quiere decir que, suponiendo que exista una relación entre el esfuerzo de desarrollo y la calidad del software, no es necesariamente cierto que aumentando el esfuerzo se consiga una mejoría en la calidad, aunque sí es probable que suceda.

La estimación del costo del software es el proceso de predecir el esfuerzo requerido para el desarrollo de un sistema de software. La mayor parte del costo de desarrollo se origina en el esfuerzo humano, por lo que la mayoría de los métodos de estimación proporcionan sus estimaciones de esfuerzo en términos de persona-mes de los programadores, analistas y gestores de proyecto. La estimación del esfuerzo puede ir acompañada de otras unidades, como son la duración del proyecto (en días en el calendario) y el costo (basado en el costo medio en unidades de moneda de la plantilla implicada en el desarrollo).

Históricamente, la estimación del costo de desarrollo se considera la etapa más compleja del desarrollo de software, debido entre otras cosas a lo siguiente:

- Ausencia de bases de datos históricas adecuadas de medidas de costo.
- Factores implicados en el desarrollo de software que están interrelacionados y sus relaciones no están bien entendidas.
- Falta de práctica y habilidad del personal encargado de realizar las estimaciones.

Factores propios de la aplicación a desarrollar

A la hora de estimar el esfuerzo empleado en el desarrollo de software, hay que estimar el esfuerzo de una aplicación determinada, con un equipo y un método de desarrollo concreto. Todos estos elementos cuentan con características propias, denominadas *factores*. Entre los factores comunes se encuentran:

- *Tamaño.* Este factor se suele medir en líneas de código (LOC), en miles de líneas de código (KLOC) o en líneas de código fuente (SLOC). También se puede medir mediante puntos de función o considerando el espacio de almacenamiento (bytes).

  Comúnmente se ha considerado el factor del tamaño como el más relacionado con el esfuerzo de software, aunque la contabilización de líneas de código no es un proceso claro, ya que se pueden encontrar líneas en blanco o comentarios, que no están relacionados directamente con el esfuerzo de desarrollo, aunque sí influyen en la legibilidad del código, útiles a la hora de modificar una aplicación existente. Hay que tener en cuenta que no todas las líneas de código que se escriben, y que por tanto suponen un esfuerzo, llegan a ser entregadas, y que la utilización de generadores automáticos (pantallas, formularios o interfaces de usuario) crea miles de líneas de código con un esfuerzo mucho menor.

  En una aplicación orientada a objetos, establecer su tamaño en líneas de código no es apropiado, por lo que el tamaño en relación con el esfuerzo es significativo, pero no determinante.

- *Lenguaje de desarrollo*. Cada lenguaje permite desarrollar un tipo de software de manera óptima, de acuerdo con su nivel de abstracción, forma de ejecución o paradigma de programación al que apoya. Unos son más adecuados que otros a la hora de realizar ciertas aplicaciones. Adicionalmente, en este factor se debe tener en cuenta la complejidad intrínseca de cada lenguaje. Mientras que algunos métodos de estimación consideran este factor de forma directa (SLIM), otros lo combinan en conjunto con la experiencia y conocimiento de los desarrolladores del equipo.

- *Herramientas de desarrollo*. Desarrollar un sistema complejo con una herramienta elemental, multiplica el esfuerzo dedicado o lo convierte en una tarea casi imposible. Por el contrario, desarrollar una aplicación muy simple con una herramienta muy compleja y potente, requiere de un esfuerzo extra dedicado al manejo de la herramienta. Por ello, emplear la herramienta adecuada implica mejorar la productividad y ajustar el esfuerzo de desarrollo. Cada herramienta lleva implícito un periodo de adaptación que se duplica cuando se decide cambiarla.

- *Fiabilidad del software*. Es la probabilidad de ejecutar una aplicación y que no se presente ninguna falla en el sistema durante un tiempo y condiciones específicas. Con la prevención de fallos se intentan corregir las fallas durante la etapa de desarrollo. Se puede ver que este factor está estrechamente relacionado con la fase de pruebas dentro del ciclo de vida de desarrollo del software; por lo tanto, cuanto mayor sea la necesidad de fiabilidad del software a desarrollar, más concisa será la fase de pruebas y, consecuentemente, el esfuerzo necesario para el desarrollo de software se verá incrementado.

- *Almacenamiento del software*. Las necesidades de almacenamiento requeridas por el software, principalmente el tamaño de la base de datos, pueden suponer esfuerzo extra en el desarrollo del software. Hoy en día el espacio necesario para un sistema o una base de datos no muy grande no supone un gran inconveniente, no siendo así con una base de datos grande, donde el esfuerzo se incrementa por la necesidad de gestionar la infraestructura de almacenaje necesaria. En los proyectos medianos, el espacio de almacenamiento no suele suponer ningún esfuerzo extra.

- *Complejidad del software*. La complejidad o dificultad asociada a un software depende del dominio del problema, de los impedimentos en la gestión del proceso de desarrollo, el nivel de detalle exigido o los límites de la capacidad humana. La complejidad del desarrollo se reduce si el equipo de programación asociado al proyecto cuenta con la experiencia suficiente en el desarrollo de aplicaciones similares.

- *Factores de plataforma*. Son aquellos que capturan el esfuerzo extra de ajustar la aplicación a desarrollar el entorno en donde se va a implementar o ejecutar, el cual puede verse afectado por el acceso individual o en red, los sistemas operativos, el tiempo de respuesta, la capacidad de cómputo de la máquina, etc.

- *Requerimientos no funcionales*. Son aquellas características que se exigen a la aplicación a desarrollar, como pueden ser la reusabilidad o una documentación exhaustiva.

Métodos de estimación

Los métodos de estimación que se presentan están clasificados en dos grupos: métodos no algorítmicos y métodos algorítmicos.

- *Métodos de estimación no algorítmicos*

Son los que no se basan en modelos matemáticos; por lo tanto, se comprenden en esta categoría, el método de estimación experta, el uso de analogías y otros métodos menos conocidos.

- *Método de estimación experta*

Es el más extendido en uso y está basado en la experiencia del experto (aquella persona que cuenta con experiencia como supervisor o gestor, o bien, que tiene experiencia haciendo estimaciones), y su conocimiento de las normas dominantes de la industria como base para la estimación del costo del software.

A pesar de ser considerado como el método más extendido, no está libre de problemas, ya que este enfoque no es ni repetible ni explícito y los factores que maneja no son muy conocidos, puesto que dependen de cada experto. Adicional a lo anterior, contar con un experto experimentado para un nuevo proyecto es muy complicado, y sus estimaciones están sujetas a un alto grado de inconsistencia.

Un intento de minimizar la inconsistencia de las estimaciones de los expertos, consiste en consultar a varios expertos aplicando técnicas de consenso, como Delphi y PERT (*Project Evaluation and Review Techniques*, Técnicas de Revisión y Evaluación de Proyectos).

Por otro lado, puesto que los gestores de los proyectos intentan maximizar la productividad y minimizar el costo, para hacer coincidir las estimaciones con el objetivo fijado con el cliente se deja en entredicho su experiencia, puesto que basarán sus estimaciones en estas manipulaciones de los presupuestos de proyectos anteriores de la compañía.

- *Uso de analogías*

Consiste en comparar el estimado del nuevo proyecto con el costo real de proyectos anteriores de la organización, ya sea en su totalidad o por partes. Comparar los proyectos por partes tiene la ventaja de ser un proceso más detallado que si se comparan proyectos en su totalidad, aunque comparar los proyectos en su totalidad lleva implícita la ventaja de incluir todos los componentes relativos al costo, incluso los componentes de los que no se tiene constancia.

La ventaja de este método radica en que las estimaciones se hacen con base en el costo real de los proyectos, lo que implica que se dispone de una cantidad de datos suficientes de proyectos anteriores. Aun así, no queda muy claro qué proyectos anteriores son representativos de los nuevos proyectos.

- *Otros métodos*

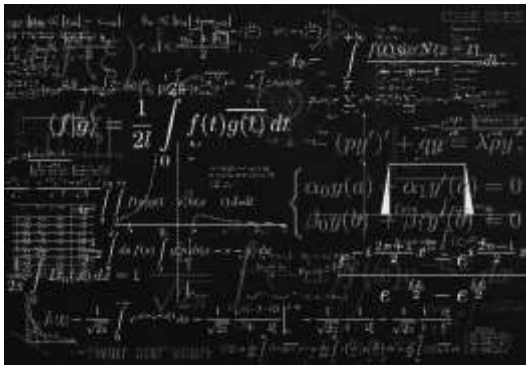
a) *El principio de Parkinson*. Determina el costo de desarrollo basándose en la máxima de que el trabajo se expande hasta completar todo el volumen disponible.
b) *Price-to-win*. Determina el costo del proyecto en función del precio que está dispuesto a pagar el cliente. Con este método es habitual que el encargado de realizar la estimación se vea obligado a ajustar la misma en función de lo que quiere pagar el cliente, y así conseguir que le asignen el presupuesto a la compañía.

La aplicación de este método puede derivar en retrasos en el desarrollo o en obligar al equipo a hacer horas extra.

c) *Enfoques bottom-up* y *top-down*. Hacen estimaciones basados en la descomposición del software en componentes. En el enfoque *bottom-up* se estima por separado cada componente del sistema. Posteriormente se unen todas las estimaciones para generar la estimación del proyecto en su totalidad. El único requerimiento para este enfoque radica en conocer perfectamente cómo se descompone el sistema, algo complicado en las etapas iniciales del desarrollo. Por otro lado, el enfoque *top-down* primero realiza una estimación global del proyecto, haciendo uso del método de estimación que prefiera, y a continuación se divide en componentes. Este enfoque resulta más útil en fases tempranas del desarrollo. Ambos enfoques no son métodos de estimación como tales, sino estrategias para facilitar la tarea de estimar el esfuerzo de desarrollo de software.

- *Métodos de estimación algorítmicos*

Realizan sus estimaciones de costo en función de un conjunto de variables y parámetros considerados como los principales factores de costo. Los diferentes métodos de estimación algorítmicos se diferencian tanto por los factores que emplean en su modelo, como por la forma de la función que utilizan (modelos lineales, multiplicativos o con función de rendimientos). Los métodos de estimación algorítmicos se clasifican en métodos empíricos y métodos analíticos.



- *Métodos de estimación algorítmicos empíricos*

Se basan en la experiencia a la hora de realizar una formulación matemática que modele el esfuerzo de desarrollo de software. COCOMO es un ejemplo de este tipo de métodos.

COCOMO (*Constructive cost model*) es un método de estimación ideado por Boehm a principios de 1980. Está basado en el método Walston-Felix. Se trata de un método de estimación fundamentado en una función de rendimiento expresado como:

$$Esfuerzo=a*S^b*m(X)$$

en donde *S* es el tamaño del código en miles de líneas (KLOC), *a* y *b* son coeficientes dependientes de la complejidad del software y *m(X)* representa un multiplicado dependiente de 15 factores.

En COCOMO los factores de costo se dividen en cuatro grupos: factores de producto (fiabilidad requerida, tamaño de la base de datos y complejidad del producto), factores de computación (restricciones de tiempo de ejecución y de almacenaje, inestabilidad de la máquina virtual y de tiempo de respuesta del equipo), factores del equipo (capacidad del analista, capacidad del programador, experiencia con la aplicación a desarrollar, con el lenguaje de desarrollo y la máquina virtual) y factores del proyecto (plan de desarrollo requerido, herramienta de desarrollo y la utilización de prácticas modernas de programación). Este método basa la calibración de sus coeficientes en 63 proyectos finalizados, para los que consigue resultados moderados, aunque no se puede concluir que sea válido para todos los casos.

El modelo COCOMO reflejaba las prácticas de desarrollo de software de la época de los 80. En la década y media siguiente estas técnicas cambiaron radicalmente, por lo que la aplicación del modelo original empezó a resultar problemática y se acabó por desarrollar un nuevo modelo: COCOMO II.

- *Métodos de estimación algorítmicos analíticos*

Se basan en una comprensión del problema mediante la descomposición, para así entender mejor su comportamiento. A partir de ello desarrollan ecuaciones matemáticas que modelen el problema de estimar el esfuerzo de desarrollo del software. Ejemplo de estos métodos son SOFTCOST y SLIM.

SOFTCOST. Este modelo asume una relación lineal entre el esfuerzo y el tamaño de la aplicación mediante la siguiente fórmula:

$$\textbf{\textit{Esfuerzo}}=S/P_1 \text{ considerando } P_1 = P_0A_1A_2,$$

Donde:

S = es el tamaño estimado en KLOC.

P_1 = es la productividad media en KLOC/Personas-mes. Para definir P_1 se utilizan varios factores.

P_0 = es una constante.

A_1 = engloba seis factores de ajuste de productividad combinados mediante una fórmula matemática.

A_2 = al igual que A_1, es una combinación de otros 29 factores.

El método SOFTCOST engloba un total de 68 parámetros. La implementación del SOFTCOST es interactiva y recopila la información de una serie de 47 preguntas con las que deduce los valores de los factores. Como salida ofrece el esfuerzo y la duración del desarrollo de software descompuesto en una estructura estándar. También ofrece una estimación del nivel de la plantilla, tamaño en páginas de la documentación y requerimientos del procesador.

SLIM es un método de estimación que se apoya en el modelo teórico de Norden-Rayleigh sobre la forma de la curva de los desarrollos de los proyectos, y en teoría es aplicable a proyectos con un tamaño superior a 70 000 líneas de código. Se basa

en el punto de vista de que la cantidad de trabajo asociada a cualquier producto se puede ver como una proporción del producto entre el esfuerzo realizado y el tiempo necesario para conseguirlo, expresado en la fórmula:

$$\textbf{Producto} = \textbf{Parámetro de productividad} * (\textbf{Esfuerzo}/B)^{1/3} * \textbf{Tiempo}^{4/3}$$

Donde:

Producto = representa cierta medida sobre la funcionalidad del mismo y se suele medir en LOC (*Lines of code*, líneas de código).

El *esfuerzo* = viene medido en personas-año o personas-mes.

El *tiempo* = representa la cantidad de tiempo empleada en el desarrollo y se mide en meses o años.

El *parámetro de productividad* = es una constante que engloba los factores del entorno y expresa la productividad de la compañía. Esta constante permite igualar las otras tres variables. Siendo así la cantidad de *producto* con el mismo *esfuerzo* y *tiempo* dedicado, puede variar dependiendo del entorno de trabajo.

El parámetro *B* = representa la destreza en función del tamaño del sistema. Las potencias de la fórmula corresponden a sucesivas validaciones del método.

3.3. Administración de la configuración

La administración de la configuración es un conjunto de actividades que se llevan a cabo en el desarrollo de un proyecto de ingeniería de software que tiene como finalidad identificar, organizar y controlar los cambios que sufre el software a lo largo de su ciclo de vida.

Estos cambios generalmente tienen como origen nuevas reglas de negocio, nuevas necesidades demandadas por los clientes, cambios en la organización del negocio y cambios en la planificación del negocio.

3.3.1. Líneas base

Una línea base es una especificación o producto que se ha revisado de manera formal por los administradores y que sirve como base para continuar en el desarrollo, y por lo tanto sólo se puede modificar mediante los procedimientos formales de control de cambios.

Los objetivos de las líneas base son:

- Identificar de forma clara, concisa y precisa, cualquier tipo de cambio que se desee realizar al producto.
- Controlar la modificación para evitar que se modifique algún otro elemento no previsto.
- Asegurar que se cumpla adecuadamente con lo especificado anteriormente.
- Informar a todas las personas involucradas con el proyecto los cambios que se han realizado.

Las ventajas que representa establecer las líneas base en un proyecto son:

- Capacidad de evaluar el desempeño, haciendo comparativos con datos históricos de planificaciones anteriores, identificando si el proyecto transita de manera correcta o incorrecta.
- Calcular el valor devengado en este proyecto contra costos y tiempos de proyectos anteriores, consiguiendo tendencias de desempeño y rutas probables.
- Mayor exactitud en estimaciones futuras, al contar con un histórico de estimaciones previas comparadas con las actuales.

La gestión de la configuración del software administra todas las entidades del producto (denominadas por algunos Elementos de Configuración del Software, ECS), así como sus diversas representaciones en documentación. Ejemplos de entidades gestionadas en el proceso de ingeniería de software incluyen:

1. Planes de administración.
2. Especificaciones (requerimientos y diseño).
3. Documentación del usuario.
4. Diseño, casos y especificación de procedimientos de pruebas.
5. Pruebas de datos y procedimientos de generación de pruebas.
6. Software de soporte.
7. Diccionarios de datos y diversas referencias cruzadas.
8. Código fuente (el medio que pueda leerse por el hardware).
9. Código ejecutable (el ejecutable).
10. Librerías.
11. Bases de datos.
    a. Datos a procesar.
    b. Datos que son parte de la programación.
12. Documentación de mantenimiento (listados, descripciones detalladas de diseño, etc.).

3.3.2. El proceso de administración de la configuración

Para que la administración de la configuración sirva a garantizar la calidad del software, es necesario que se desarrollen cinco tareas principales.

Identificación de objetos en la configuración del software

A fin de estar en condiciones de controlar y gestionar los ECS, es necesario que éstos primero se identifiquen de manera única, para posteriormente organizarlos mediante un enfoque de objetos. Se pueden identificar objetos básicos (una sección de una especificación de requisitos, un listado fuente de un módulo, un conjunto de casos prueba) y objetos compuestos (colección de objetos básicos y de otros objetos compuestos, tal como la especificación de diseño completa).

La identificación de objetos debe considerar entre los siguientes elementos:

- Un nombre. Cadena de caracteres que identifican al objeto sin ambigüedad.
- Una descripción. Se conforma por los siguientes datos:
    - Tipo de ECS (documento, programa).
    - Identificador del proyecto al que pertenece el objeto
    - Información de la versión.
- Una lista de recursos que el objeto proporciona, procesa, referencia o requiere, ya sean tipos de datos, funciones específicas o nombres de variables de paso de parámetros.

- Las relaciones existentes entre los objetos, ya sea relación de tipo “parte-de”, o bien, las relaciones entre objetos como modelo de datos-diagramas de flujo de datos, o clase-conjunto de casos de prueba. Estas relaciones se pueden representar mediante un lenguaje de interconexión de módulos, el cual describe las interdependencias entre los objetos, con lo cual se construyen las versiones de un software.

Control de versiones

Esta actividad es el resultado de combinar procedimientos y herramientas para administrar las versiones de los objetos de configuración establecidos, de tal suerte que se puedan lograr configuraciones alternativas a través de una elección de las versiones adecuadas. Para gestionar las versiones es necesario asociarlas con un atributo, de tal forma que se pueda construir una configuración a partir de un conjunto de atributos. Los atributos generalmente son números específicos y progresivos. Bajo este concepto, cada versión se compone de una serie de ECS con el mismo nivel de revisión. Otra forma de configurarlos es mediante variantes, las cuales coleccionan objetos con diferente nivel de revisión. Se considera que el software cambia de versión cuando se realizan cambios significativos en uno o más objetos de la configuración.

Control de cambios

Esta actividad requiere de procedimientos humanos y herramientas automáticas, e inicia con una petición de cambio, la cual se evalúa a fin de establecer el esfuerzo necesario para realizarlo, el impacto que tendrá sobre otras funciones y objetos del software para imaginar posibles “efectos secundarios”. Esta evaluación es informada a la autoridad de control de cambios, la cual puede recaer en una persona o en un grupo, y quiénes serán los responsables de tomar la decisión respecto al cambio y la prioridad que se le asigna. Por cada cambio aprobado por esta autoridad

se genera una “orden de cambio”, la cual es un documento que describe el cambio, las restricciones del mismo y los criterios para revisarlo y auditarlo. Al ejecutarse esta orden, el objeto es separado de la configuración, modificado, revisado y nuevamente integrado en la configuración, previo manejo del control de versiones. La siguiente figura nos muestra el proceso de forma gráfica.

Ilustración 2. Proceso de control de cambios. Elaboración propia.

Auditoría de la configuración

Para asegurar la calidad del software es necesario verificar que el cambio fue implementado correctamente. En apoyo a ello se realizan las revisiones técnicas formales, las cuales evalúan el ECS para establecer la consistencia del mismo con respecto a otros ECS, así como las omisiones o posibles efectos secundarios. Esta revisión se debe realizar para cualquier cambio importante.

La auditoría de configuración del software complementa la revisión técnica formal y comprueba otras características del cambio, relacionado con aspectos como:

- Verificación que el cambio atienda a la orden de cambio generada para que no incorpore modificaciones adicionales.
- Verificación que se ha realizado la revisión técnica formal.
- Evaluación respecto al proceso seguido y a los estándares del proyecto que debieron aplicarse.
- Verificación de que los cambios se han aplicado también en los atributos del objeto en la configuración del software mediante los procedimientos establecidos.
- Verificación de que se han actualizado todos los ECS relacionados con el ECS modificado.

Informes de estado

En esta tarea se realizan las actividades necesarias para llevar el registro de cada tarea de gestión de la configuración del software realizada, atendiendo a los parámetros: qué se realizó, quién lo realizó, cuándo se realizó y qué se afecta con

lo realizado. Conforme se van registrando estos eventos, se van presentando de manera regular y constante a todos los miembros del proyecto, para que todos estén enterados en tiempo y forma de lo que sucede con los ECS, sobre todo con aquellos que les afecten directamente.

3.4. Administración de cambios

En el tema anterior se estableció que uno de los procesos de la administración de la configuración es el control de cambios, también conocido como control de la configuración del software. A continuación veremos los aspectos básicos que rodean esta actividad.

Esta actividad atiende la gestión de los cambios durante el ciclo de vida del software. Comprende el proceso para determinar los cambios a realizar, la autoridad que los aprobará, el apoyo para implementarlos y las desviaciones formales y exenciones a los requisitos del proyecto. La información que resulte de estas actividades será útil para conocer los grados y frecuencia de cambios y los grados y frecuencia de los retrabajos.

Solicitud, evaluación y aprobación de cambios en el software

El primer paso en esta actividad es determinar los cambios que se realizarán a determinados elementos. El proceso de solicitud de cambio al software establece procedimientos formales para el envío y registro de las solicitudes de cambio, evaluando el costo e impacto potencial del cambio propuesto, y aceptando, modificando o rechazando el cambio propuesto.

Las solicitudes de cambio en los elementos de configuración del software se originan por cualquier persona en cualquier momento del ciclo de vida del software y pueden incluir una propuesta de solución y la prioridad del mismo. El origen de estas solicitudes es el inicio de una acción correctiva en respuesta a problemas reportados. Independientemente del origen, el tipo de cambio (defecto o mejora) por lo general es registrado en la Solicitud de Cambio del Software (SCS), la cual brinda la oportunidad de dar seguimiento a los defectos, así como de recolectar evidencias de las actividades por cada tipo de cambio solicitado.

Una vez que se recibe un SCS, se realiza una evaluación técnica para establecer el alcance de las modificaciones que serían necesarias para que el cambio se acepte. Para esta tarea es importante una buena comprensión de las relaciones entre los elementos de software (y probablemente del hardware). Finalmente, una autoridad establecida considerará la línea base afectada, el elemento de configuración de software involucrado y la naturaleza del cambio para evaluar los aspectos técnicos y de gestión de la solicitud de cambio, y en su caso aceptarla, modificarla, rechazarla o aplazar el cambio propuesto.

Comité de control de la configuración del software

La autoridad para aceptar o rechazar las solicitudes de cambio propuestas reside en una agrupación generalmente denominada Comité de control de la configuración del software. En proyectos pequeños, esta autoridad puede residir en el líder responsable o en una persona asignada para ello, en lugar de un comité. Puede haber varios niveles de autoridad de cambio, dependiendo de diversos criterios, tales como lo crítico que sea el elemento involucrado, la naturaleza del cambio (si impacta en los presupuestos será muy importante), o el estado actual en el ciclo de vida del

software. Esto significa que la composición del comité dependerá de dichos factores, siempre considerando que todos los involucrados deben estar representados. También será necesario que la actividad de este comité sea auditada o revisada por el área de aseguramiento de la calidad.

Soporte al proceso de solicitud de cambio de software

Un proceso eficiente de control de cambios requiere del uso de herramientas y procedimientos de soporte, que comprende formularios impresos, procedimientos documentados e inclusive herramientas digitales para generar las solicitudes de cambio, organizando el flujo del proceso, capturando las decisiones del comité y reportando información del proceso de cambios. Un enlace entre esta herramienta y el sistema de reporte de problemas facilitaría el seguimiento de solución a problemas reportados. Generalmente las herramientas para la administración del cambio se adaptan a los procesos locales y las *suites* de herramientas se desarrollan a nivel local, generando lo que se conoce como un ambiente para la ingeniería de software.

Implementando los cambios en el software

Las solicitudes de cambio aprobadas son implementadas utilizando procedimientos definidos y coherentes de software con los requisitos de programación aplicables. Dado que es posible implementar varias solicitudes de cambio de manera simultánea, es necesario contar con mecanismos para “rastrear” cuáles solicitudes de cambio son incorporadas en versiones o líneas base en particular. Como parte del cierre del proceso de cambio, estos cambios deberán someterse a auditorías de configuración y verificación por parte del área de calidad, a fin de garantizar que se hicieron los cambios aprobados.

La implementación actual de un cambio está soportada por las capacidades de la herramienta de librería del sistema de gestión de versiones y por el repositorio de código. Como mínimo, estas herramientas proporcionan la entrada/salida y las capacidades de control de versiones asociadas. Algunas herramientas más poderosas pueden soportar el desarrollo paralelo y los ambientes geográficamente distribuidos.

Estas herramientas pueden manifestarse como aplicaciones especializadas, separadas bajo el control de un grupo de gestión de cambios independiente. También pueden aparecer como parte integrada de un ambiente de desarrollo de software. Finalmente, un sistema de control de cambios rudimentario y elemental proporcionado por el sistema operativo.

Desviaciones y renuncias

Las restricciones impuestas a los esfuerzos de desarrollo de software o las especificaciones producidas durante las actividades de desarrollo pueden contener provisiones que no pueden satisfacerse en un punto específico del ciclo de vida.

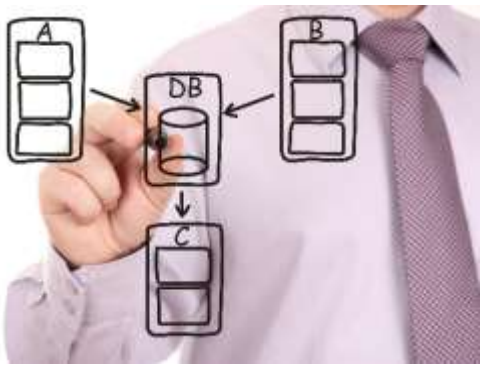
Una desviación es una autorización para abandonar una provisión previa establecida para el desarrollo del elemento.

Una renuncia es una autorización para utilizar un elemento, continuando su desarrollo, que lo aparta de la provisión de alguna forma. En estos casos, se utiliza un proceso formal para ganar la aprobación para las desviaciones o renuncia de las provisiones.

3.5. Modelo de la arquitectura propuesta

Arquitectura de software (AS), de acuerdo con el **IEEE Std 1471-2000**, es “la organización fundamental de un sistema encarnada en sus componentes, las relaciones entre ellos y el ambiente y los principios que orientan su diseño y evolución” (IEEE, 2000: 14). Para mayor información consultar el documento *IEEE Recommended practice for architectural description of software-intensive systems*, recuperado de http://cabibbo.dia.uniroma3.it/ids/altrui/ieee1471.pdf

La arquitectura de software es la representación de las estructuras del sistema, la cual comprende los componentes del software, las propiedades de esos componentes visibles externamente y las relaciones entre ellos.



Establecer la arquitectura de software es importante porque facilita la comunicación entre los miembros del equipo de desarrollo, permite tomar decisiones a tiempo que impactarán todo el proyecto y se constituye como un modelo de cómo se conforma el sistema y cómo trabaja.

Lo más rescatable de estas estructuras arquitectónicas (estilos) es que su abstracción permite que puedan aplicarse cuando se diseñan otros sistemas.

Por lo tanto, cada estilo arquitectónico comprende los siguientes elementos:

- Conjunto de componentes que realizan una función específica para el software.
- Conjunto de conectores mediante los cuales se realiza la comunicación, coordinación y cooperación entre componentes.
- Restricciones respecto a la integración de los componentes.
- Modelos semánticos que permiten la comprensión de las propiedades globales, para analizar las propiedades conocidas de los componentes que la integran.

De acuerdo con ello, y con todo el software que se ha desarrollado en los últimos años, se pueden identificar los siguientes estilos arquitectónicos:

- *Arquitectura basada en los datos.* En el centro de esta arquitectura se encuentra un almacén de datos (un documento o una base de datos) al que otros componentes acceden con frecuencia para actualizar, añadir, borrar, o bien, modificar los datos del almacén. En algunos casos, el almacén de datos es pasivo. Esto significa que el software de cliente accede a los datos, independientemente de cualquier cambio en los datos o de las acciones de otro software de cliente. Una variación en este acceso transforma el almacén en una pizarra que envía notificaciones al software de cliente cuando los datos de interés del cliente cambian. Estas arquitecturas promueven la capacidad de integración. Por consiguiente, los componentes existentes pueden cambiarse o los componentes del nuevo cliente pueden añadirse a la arquitectura sin involucrar a otros clientes (porque los componentes del cliente operan independientemente). Además, los datos pueden ser transferidos entre los clientes, utilizando un mecanismo de pizarra. Los componentes son procesos ejecutados independientemente.

- *Arquitectura de flujo de datos.* Esta arquitectura se aplica cuando los datos de entrada son transformados a través de una serie de componentes computacionales o manipulativos en los datos de salida. Un patrón tubería y filtro tiene un grupo de componentes, llamados filtros, conectados por tuberías que transmiten datos de un componente al siguiente. Cada filtro trabaja independientemente de aquellos componentes que se encuentran en el flujo de entrada o de salida; está diseñado para recibir la entrada de datos de una cierta forma y producir una salida de datos (hacia el siguiente filtro) de una forma específica. Sin embargo, el filtro no necesita conocer el trabajo de los filtros vecinos. Si el flujo de datos degenera en una simple línea de transformadores se le denomina secuencial por lotes. Este patrón aplica una serie de componentes secuenciales (filtros) para transformarlos.

- *Arquitecturas de llamada y retorno*. Este estilo arquitectónico permite al diseñador del software (arquitecto del sistema) construir una estructura de programa relativamente fácil de modificar y ajustar a escala. Existen dos subestilos dentro de esta categoría:

  - Arquitecturas de programa principal/subprograma: esta estructura clásica de programación descompone las funciones en una jerarquía de control, donde un programa «principal» llama a un número de componentes del programa, los cuales, en respuesta, pueden también llamar a otros componentes
  - Arquitecturas de llamada de procedimiento remoto: los componentes de una arquitectura de programa principal/subprograma, están distribuidos entre varias computadoras en una red.

- *Arquitecturas orientadas a objetos*. Los componentes de un sistema encapsulan los datos y las operaciones que se deben realizar para manipular los datos. La comunicación y la coordinación entre componentes se lleva a cabo a través del paso de mensajes.

- *Arquitecturas estratificadas*. La estructura básica de una arquitectura estratificada genera diferentes capas y cada una realiza operaciones que progresivamente se aproximan más al cuadro de instrucciones de la máquina. En la capa externa, los componentes sirven a las operaciones de interfaz de usuario. En la capa interna, los componentes realizan operaciones de interfaz del sistema. Las capas intermedias proporcionan servicios de utilidad y funciones del software de aplicaciones.

Beneficios

Entre los beneficios que reporta el hacer uso de la AS están los siguientes:

1. Comunicación mutua. La AS representa un alto nivel de abstracción común que la mayoría de los participantes, si no todos, pueden usar como base para crear entendimiento mutuo, formar consenso y comunicarse entre sí. En sus mejores expresiones, la descripción arquitectónica expone las restricciones de alto nivel sobre el diseño del sistema, así como la justificación de decisiones arquitectónicas fundamentales.
2. Decisiones tempranas de diseño. La AS representa la encarnación de las decisiones de diseño más tempranas sobre un sistema, y esos vínculos tempranos tienen un peso fuera de toda proporción en su gravedad individual con respecto al desarrollo restante del sistema, su servicio en el despliegue y su vida de mantenimiento. La arquitectura representa lo que el método SAAM, una puerta de peaje: el desarrollo no puede proseguir hasta que los participantes involucrados aprueben su diseño.

3. Restricciones constructivas. Una descripción arquitectónica que proporciona *blueprints* parciales para el desarrollo, indicando los componentes y las dependencias entre ellos.
4. Reutilización o abstracción transferible de un sistema. La AS encarna un modelo relativamente pequeño, intelectualmente tratable, de la forma en que un sistema se estructura y sus componentes se entienden entre sí; este modelo es transferible a través de sistemas; en particular, se puede aplicar a otros sistemas que exhiben requerimientos parecidos y puede promover reutilización en gran escala. El diseño arquitectónico soporta reutilización de grandes componentes o incluso de *frameworks*, en donde se pueden integrar componentes.
5. Evolución. La AS puede exponer las dimensiones, a lo largo de las cuales puede esperarse que evolucione un sistema. Haciendo explícitas estas “paredes” perdurables, quienes mantienen un sistema pueden comprender mejor las ramificaciones de los cambios y estimar con mayor precisión los costos de las modificaciones. Esas delimitaciones ayudan también a establecer mecanismos de conexión que permiten manejar requerimientos cambiantes de interoperabilidad, prototización y reutilización.
6. Análisis. Las descripciones arquitectónicas aportan nuevas oportunidades para el análisis, incluyendo verificaciones de consistencia del sistema, conformidad con las restricciones impuestas por un estilo, conformidad con atributos de calidad, análisis de dependencias y análisis específicos de dominio y negocios.
7. Administración. La experiencia demuestra que los proyectos exitosos consideran una arquitectura viable como un logro clave del proceso de desarrollo industrial. La evaluación crítica de una arquitectura conduce típicamente a una comprensión más clara de los requerimientos, las estrategias de implementación y los riegos potenciales.

RESUMEN

A lo largo de esta unidad se estableció que los objetivos son elemento fundamental de los proyectos, por lo que se abordó su importancia, las características para otorgarle el atributo de claro, y las técnicas “Es/No es” y SMART, que apoyan su generación de manera adecuada y exitosa.

Asimismo, se abordó la temática de estimación de tareas y tiempos, iniciando con una definición de estimación en general y su importancia, así como los elementos que comprende la estimación, algunos métodos de estimación desarrollados y practicados y los factores comunes a las aplicaciones que consideran dichos métodos.

Posteriormente se abordó el tema de administración de la configuración, tratándose temas como la línea base, los elementos de configuración del software, el proceso de configuración y la administración de cambios, en donde se profundizó en aspectos relacionados con esta actividad.

Finalmente, se revisaron los temas de arquitectura de software, los estilos de arquitectura de software y los beneficios que reporta su uso.

MESOGRAFÍA

Bibliografía básica

Benítez, J. (2011). *Enfoque más comercial de la fase de definición de un proyecto informático*. Universitat Oberta de Catalunya. Disponible en http://openaccess.uoc.edu/webapps/o2/bitstream/10609/6102/1/fbenitezcTFC0111memoria.pdf

Billy, C. (2004). *Introducción a la arquitectura de software*. Universidad de Buenos Aires. Disponible en http://carlosreynoso.com.ar/arquitectura-de-software/

Pressman, R. (2002). *Ingeniería del software. Un enfoque práctico* (5ª ed.). España: McGraw-Hill.

Bibliografía complementaria

Bass, L., Clements, P. y Kazman, R. (1998). *Software architecture in practice*. USA: Addison-Wesley.

Benítez, J. (2011). *Enfoque más comercial de la fase de definición de un proyecto informático*. Universitat Oberta de Catalunya. Disponible en http://openaccess.uoc.edu/webapps/o2/bitstream/10609/6102/1/fbenitezcTFC0111memoria.pdf

Billy, C. (2004). *Introducción a la arquitectura de software*. Universidad de Buenos Aires. Disponible en http://carlosreynoso.com.ar/arquitectura-de-software/

Blog Dolphy. (2010). *Establece los objetivos de tus proyectos*. Disponible en http://blog.doolphy.com/es/2010/08/10/establece-los-objetivos-de-tus-proyectos/

Esteban, J. y Dolado, J. (2008). Estimación del esfuerzo software: factores vinculados a la aplicación a desarrollar. *Actas de los Talleres de las Jornadas de Ingeniería del Software y Bases de Datos* (vol. 2-1).

Futrell, R., Shafer, D. y Shafer, L. (2002). *Quality software project management*. USA: Prentice Hall.

Guerra, L. y Bedini, A. (2005). *Gestión de proyectos de software*. Chile: UTFSM. Disponible en http://www.inf.utfsm.cl/~guerra/publicaciones/Gestion%20de%20Proyectos%20de%20Software.pdf

Heriot-Watt University (HWU). (2008). *How to write SMART objectives*. Disponible en http://www.hw.ac.uk/hr/htm/pdr/06b%20SMART%20Objectives.pdf

Institute of Electrical and Electronics Engineers, INC (IEEE). (2000). *IEEE Recommended practice for architectural description of software-intensive systems*. Software Engineering Standards Committee. USA. Disponible en http://cabibbo.dia.uniroma3.it/ids/altrui/ieee1471.pdf

Moreno, A. (s. f.). *Estimación de proyectos de software. Titulación: Ingeniería Superior en Informática. Asignatura de Planificación y Gestión de Proyectos Informáticos*. Universidad de Alcalá. Disponible en http://www.cc.uah.es/jlcastillo/PP/media/EstimacionProyectosSoftware.pdf

Scott, J. y Nisse, D. (2001). Software configuration management. En *Software Engineering Body of Knowledge* (SWEBOK). IEEE. USA.

Universitat Oberta de Catalunya (UOC). (2003). *Formación de formadores. Las fases.* Disponible en http://cv.uoc.edu/UOC/a/moduls/90/90_156/programa/

Sitios electrónicos

Sitio	Descripción
http://www.sei.cmu.edu/architecture/	*Software Architecture. Software Engineering Institute*. Espacio del Instituto de Ingeniería de Software, desde el cual se puede acceder a la investigación, herramientas y métodos, servicios de consulta y casos de estudio relacionados con la arquitectura de software.
http://www.infosysblogs.com/softwaretools/2012/06/software_configuration_managem.html	*Software Configuration Management Tools. Infosys*. Espacio comercial que hace un interesante recuento de las herramientas disponibles en el mercado para la administración de la configuración del software.
http://www.codingthearchitecture.com/pages/about.html	*Coding the architecture*. Comunidad de arquitectos de software.

UNIDAD 4

Implementación de componentes

OBJETIVO PARTICULAR

Codificar en un lenguaje de programación los componentes del sistema.

TEMARIO DETALLADO
(24 horas)

4. Implementación de componentes

- 4.1. Definición de componentes
- 4.2. Estándares y buenas prácticas de implementación
- 4.3. Diseño y modelo de componentes
- 4.4. Técnicas de implementación
- 4.5. Depuración y métodos para revisar el código

INTRODUCCIÓN

A lo largo de esta unidad se revisarán aspectos relacionados con los componentes de software, abordando su definición, sus características clave, los ámbitos necesarios para identificarlos y las variables para clasificarlos.

El tema dos denominado “Estándares y buenas prácticas de implementación”, abordará los términos de estándar y buena práctica de codificación, para posteriormente detallar estándares generales de codificación y hacer una revisión del estándar 1074-1997, para revisar de manera especial las actividades de implementación que pertenecen al grupo de actividades de desarrollo.

El tercer tema, “Diseño y modelo de componentes”, revisará los conceptos de modelo de componentes y *framework* de componentes, profundizando también en el tema de interfaz de los componentes.

El tema cuarto denominado “Técnicas de implementación”, revisará las características principales de las programaciones modulares, estructuradas y orientadas a objetos.

Finalmente, el tema “Depuración y métodos para revisar el código” detallará los aspectos relacionados con las pruebas, sus objetivos, principios y tipos principales, para finalizar con una revisión del proceso de depuración.

z4.1. Definición de componente

Brown y Wallnau, citados por Pressman (2002: 475), nos acercan algunas definiciones de componente, cuando señalan que…

- Un componente es una parte no trivial, casi independiente y reemplazable de un sistema que llena claramente una funcionalidad dentro de un contexto en una arquitectura bien definida. Un componente se conforma y acciona por medio de un conjunto de interfaces.
- Un componente de software es un paquete dinámicamente vinculado con uno o varios programas manejados como una unidad, que son accedidos mediante interfaces bien documentadas que pueden ser descubiertas en tiempo de ejecución.

- Un componente de software es una unidad de composición con interfaces contractualmente especificadas y explícitas con dependencias dentro de un contexto. Dicho componente puede ser desplegado independientemente y es sujeto a la composición de terceros.

Las características clave de un componente son:

- Identificable: debe tener una identificación que permita acceder fácilmente a sus servicios y que permita su clasificación.
- Autocontenido: un componente no debe requerir de la utilización de otros para finiquitar la función para la cual fue diseñado.
- Reemplazable por otro componente: se puede remplazar por nuevas versiones u otro componente que lo remplace y mejore.
- Accedido mediante su interfaz: debe asegurar que éstas no cambiarán a lo largo de su implementación.

- Servicios invariables: las funcionalidades ofrecidas en su interfaz no deben variar, pero su implementación sí.
- Bien documentado: un componente debe estar correctamente documentado para facilitar su búsqueda si se quiere actualizar, integrar con otros, adaptarlo, etc.
- Genérico: sus servicios deben servir para varias aplicaciones.
- Reutilizado dinámicamente: puede ser cargado en tiempo de ejecución en una aplicación.
- Independiente: del hardware, software, sistema operativo (Ariza y Molina, 2004: 3-4).

Para poder identificar los componentes que son “compatibles” con nuestra necesidad, se deben identificar a partir de tres ámbitos (Andrews y Ghosh, 2002):

- Dominio. Este ámbito necesita emparejar los requisitos y las capacidades de los componentes, lo que implica establecer una estructura y función a nivel de la aplicación, identificando el comportamiento límite para determinar el rango, la estabilidad y la precisión de la solución, así como otros requisitos no funcionales, como la velocidad y la tolerancia a fallas. Esto representa en términos generales todos los aspectos del problema del usuario y está relacionado de forma directa con la funcionalidad que apoya el componente.
- Situación: Este ámbito debe adquirir el conocimiento acerca de las entidades de diseño y comportamiento, así como conocimiento del flujo de información y tipos de algoritmos.

- Programa: Este ámbito es el que más varía de componente a componente, ya que muestra de forma más detallada la información técnica del componente, como la estructura de los archivos de información, definiciones de las bases de datos, la definición de la interface de datos, los tipos de parámetros, información acerca de la invocación de los métodos del componente, etcétera.

Variables para clasificar componentes

La clasificación de componentes es un proceso extenso, ya que un componente involucra en sí mismo varios atributos relevantes. Entre las variables que se podrían considerar para clasificarlos están las siguientes (Ariza y Molina, 2004):

- *Tamaño*

Tal y como en el diseño orientado a objetos, el tamaño de los componentes puede ser medido a partir de dos aspectos: el número de líneas de código (LDC) o por el número de funciones que realiza.

- *Complejidad*

Conocer el nivel de complejidad de un componente es útil para saber cuándo reutilizarlo o cómo está su nivel de calidad. Sin embargo, medir la complejidad de los componentes no es un proceso fácil, y es por ello que se han desarrollado distintas métricas, como son las métricas de Tullio Vernazza, el sistema de Salman, la de Bertoa et ál., la de Cho et ál., la de Gill y Grover, la de Ardimento y la de Sharma et ál.

- *Mantenibilidad*

Se refiere a la facilidad con la cual puede sufrir cambios el componente para adaptarse a las nuevas condiciones. Para ello, las métricas se basan en el conocimiento de los atributos, el comportamiento y el flujo de trabajo de los componentes.

- *Reusabilidad*

Se refiere a la capacidad del componente de ser susceptible de uso en otra aplicación. Para establecer el valor de este indicador, se puede apoyar en dos aspectos principalmente:

- Qué tan preparado está el componente para ser reutilizado, en razón de las variables y métodos para el paso de parámetros.
- Y el porcentaje de funcionalidad que el componente representa para el total del software.

- *Frecuencia de reúso*

Se refiere a cuántas veces se utilizó un mismo componente en distintos productos de software.

- *Confiabilidad*

El porcentaje de incertidumbre respecto al funcionamiento erróneo del componente.

4.2. Estándares y buenas prácticas de implementación

Un estándar es un conjunto de criterios explícitos que sirven para especificar y determinar qué tan adecuada es una acción u objeto. El estándar de codificación especifica las características de esta labor, y es necesario porque permite que el código sea más legible, y por lo tanto sea más fácil de leer, entender, revisar, probar y modificar.

Ningún conjunto de lineamientos satisface a todo el mundo. El objetivo de un estándar es lograr la eficiencia en una comunidad de desarrolladores. Aplicar un conjunto de estándares de codificación bien definidos redundará en un código con menos *bugs* y mejor mantenibilidad. Adoptar un estándar no familiar puede ser complejo inicialmente, pero rápidamente se aprenderá y se verán los beneficios de su uso, sobre todo cuando el código se entrega para su mantenimiento a otro equipo (Ge, 2014).

Por su parte, las buenas prácticas de codificación son experiencias consideradas ejemplares, que orientan la acción del proceso de codificación apoyada en acciones ya realizadas.

Se consideran buenas prácticas de codificación las que cumplen con las siguientes características:

1. Comprensible. El código debe ser claramente legible y sencillo. Debe mostrar los elementos clave para lo que fue diseñado. Sus partes relevantes deben ser fáciles de reusar; no deben contener código innecesario y deben incluir la documentación apropiada.
2. Correcto. Debe demostrar de manera adecuada cómo ejecuta los aspectos clave para los cuales fue diseñado. Debe compilar limpiamente, correr y documentarse correctamente y evaluarse.
3. Consistente. Se debe seguir un estilo y organización de codificación que haga al código más fácil de leer. Debe ser coherente entre sí para facilitar su interacción.
4. Moderno. Debe demostrar prácticas actualizadas, como Unicode, manejo de errores, programación defensiva, portabilidad, recomendaciones del uso de librerías y uso de marcos estándares de proyectos y de construcción.
5. Confiable. Debe cumplir con los estándares legales, de privacidad y de políticas. No debe demostrar prácticas de programación laxas o de piratería. Todos los pasos de instalación y ejecución deben ser reversibles.
6. Seguro. Se deben ejercer prácticas de programación segura, tales como privilegios mínimos, versiones aseguradas de funciones de librerías en tiempo de ejecución y marcos de proyecto recomendados en el ciclo de vida del desarrollo.

El uso apropiado de prácticas de codificación, diseño y características del lenguaje determina lo bien que se logra el código; de ahí que los estándares de codificación están diseñados para servir como “mejores prácticas” que los desarrolladores pueden imitar.

Estándares generales de codificación

Los siguientes estándares generales de codificación se pueden aplicar a todos los lenguajes y proporcionan guías de alto nivel para el estilo, formateo y estructura del código fuente.

Claridad y consistencia

- Con el fin de garantizar que el código resultante sea fácil de entender y mantener, se debe generar un código claro, conciso y autodocumentado que asegure que la claridad, la legibilidad y la transparencia son primordiales.
- Cuando se apliquen los estándares de codificación debe haber certeza de que sea de manera consistente.

Formato y estilo

- No use tabuladores en los archivos fuente, porque causan confusión en el formato. Todo el código debe escribirse utilizando espacios para indentar.
- Limite la longitud de las líneas de código, ya que las líneas de código excesivamente largas inhiben la legibilidad del código. “Rompa” la línea de código cuando su longitud sobrepase las columnas 78, 86 o 90 de acuerdo al caso.
- Haga uso de una tipografía de ancho fijo, como la Courier New en el editor del código fuente.

Utilizando librerías

- No haga referencia a librerías, encabezados o ensamblajes en los archivos innecesarios. Si pone atención a estos pequeños detalles puede mejorar sus tiempos de construcción, reduciendo al mínimo la posibilidad de errores, dando a los lectores de su código una buena impresión.

Variables globales

- No minimice las variables globales. El uso apropiado de variables globales es con paso de parámetros entre las funciones. No se refiera a ellas directamente dentro de las funciones o clases, porque crearía un efecto lateral que alteraría el estado del programa en general, sin conocimiento de la función que la ha llamado. Lo mismo aplica para las variables estáticas. Si necesita modificar una variable global, se puede hacer como un parámetro de salida o regresando una copia al valor global.

Declaración de variables e inicializaciones

- Declare variables locales en el bloque con el alcance mínimo que pueda contenerlas; por lo general, justo antes de utilizarlas si el lenguaje lo permite, o si no, en la parte superior del bloque.
- Inicialice las variables cuando las declare.
- Declare e inicialice/asigne variables locales en una sola línea cuando el lenguaje lo permita. Esto reduce espacios verticales y asegura que la variable no existe con un estado sin inicializar o en un estado que cambie inmediatamente.
- No declare múltiples variables en una sola línea. Se recomienda una declaración por línea, ya que permite comentar y evita confusiones.

Declaración y llamada de funciones

- El nombre de la función/método, el valor de retorno y la lista de parámetros puede tomar diversas formas. Lo ideal sería que todo esto quedara en una sola línea. Si hay muchos argumentos que no caben en una línea, éstos pueden conjuntarse, muchos por línea o uno por línea.

- Ordene los parámetros, agrupando primero los parámetros de entrada y al final los parámetros de salida. Dentro del grupo, ordene los parámetros pensando en ayudar a los programadores a darles los valores apropiados. Cuando se diseñe una serie de funciones que tomen los mismos argumentos, utilice un orden consistente en todas las funciones.

Sentencias

- No coloque más de una sentencia en una sola línea, porque dificulta el recorrido del código en la depuración.

Enumeraciones

- Enumere los parámetros poco flexibles, propiedades y valores de retorno que representen conjuntos de valores.
- No utilice la numeración para los conjuntos abiertos (versión de sistema operativo, etcétera).
- Considere un valor “0” para una numeración simple.

Espacios en blanco (líneas en blanco)

- Utilice líneas en blanco para separar grupos de sentencias relacionadas. Omita líneas en blanco extra que no faciliten la lectura del código.
- Utilice dos líneas en blanco para separar la implementación de métodos de la declaración de clases.

Espacios

Los espacios mejoran la legibilidad al decrecer la densidad de código. Los lineamientos para utilizar espacios en el código son:

- No dejar espacio entre el nombre de la función y el paréntesis.
- Colocar un espacio después de una coma.
- No dejar espacio dentro de los corchetes.
- Colocar un espacio antes de las sentencias de control de flujo.
- Colocar un espacio para separar operadores.

Llaves

- Utilice las llaves solas en un renglón.
- Utilice llaves para contener condicionales de una sola línea, para facilitar la inserción de código en dichas condicionales en un futuro y evitar ambigüedades.

Comentarios

- Utilice comentarios que resuman cuál es el propósito y el porqué de una pieza de código, no para repetirlos.
- Utilice el formato de comentario lineal aun cuando el comentario requiera de múltiples líneas.
- Indente el comentario al mismo nivel que el código que describe.
- Utilice oraciones completas, iniciando con mayúsculas y terminando con punto, cumpliendo con aspectos de ortografía y gramática.

Comentarios entre líneas

- Deben incluirse en la propia línea y deben indentarse al mismo nivel del código que comentan con un espacio en blanco antes, no después. Los comentarios que describen un bloque de código deben aparecer en una línea, indentados al nivel del código que describen, con un espacio en blanco antes y después.

- Los comentarios entre líneas son permitidos en la misma línea que el código cuando aportan una breve descripción de la estructura, clase, variable, parámetro o sentencia corta. En este caso, es una buena idea alinear los comentarios para todas las variables.
- No inunde su código de comentarios. Comentar cada línea con descripciones obvias de lo que hace el código reduce su lectura y comprensión. Los comentarios de una sola línea deben utilizarse cuando el código está haciendo algo que podría no ser obvio.
- Debe colocar comentarios para hacer notar aspectos que no son intuitivos o comportamientos que no son obvios en la lectura del código.

Comentarios en la cabecera de archivos

- Debe haber un comentario en la cabecera de cada archivo de código creado por un programador.

Comentarios de clases

- Agregar comentarios en el encabezado de todas las clases y estructuras que no sean triviales. El nivel del comentario debe ser adecuado a la audiencia del código.

Comentarios en funciones

- Todas las funciones públicas y no públicas que no sean triviales se deben comentar en el encabezado, y el nivel del comentario debe ser adecuado en función de las personas que revisarán el código.
- Cualquier método o función que puede fallar con los efectos secundarios debe tener claramente comunicados dichos efectos en los comentarios de la función. Como regla general, el código debe escribirse de tal forma que no produzca efectos secundarios en casos de error o falla; la presencia de tales efectos secundarios debe tener una justificación clara cuando sea escrito el

código (tal justificación no es necesaria para rutinas que no producen salidas o para las que sobrescriban algún parámetro de sólo salida).

Regiones

- Utilice la declaración de la región en donde haya una gran cantidad de código que se beneficiará del mismo. Agrupar una gran cantidad de código por ámbito o funcionalidad mejora la lectura y estructuración del código.

Convenciones de nominación

Convenciones generales de nominación

- Use nombres significativos para diferentes tipos, funciones, variables, construcciones y estructuras de datos. Su uso debe ser claramente discernible a partir de su nombre.
- Las variables de un sólo carácter deben ser solamente utilizadas como contadores o como coordenadas. Como regla de oro considere que una variable debe contar con un nombre descriptivo a medida que se incrementa su ámbito de acción.
- No utilice nombres cortos o contracciones como partes de nombre de identificadores. Para funciones de tipo común, procedimientos de ventanas, procedimientos de dialogo, entre otros, utilice sufijos comunes como “DialogProc”, “WndProc”, etcétera.

Notación húngara

- Se puede utilizar la notación húngara en los nombres de parámetros y variables; sin embargo, dificultará la refactorización de código, como cuando cambia el tipo de una variable y requiere renombrarla en todo el código.

El estándar IEEE-1074-1997

Este estándar proporciona un procedimiento para crear un proceso de ciclo de vida del software (SLCP). Está dirigido principalmente al arquitecto de procesos de un proyecto de software, puesto que su función es crear el proceso del ciclo de vida del software.

Los grupos de actividades que propone son:

- Grupo de actividades de gestión de proyectos. Estas actividades inician, supervisan y controlan el proyecto de software a lo largo de su ciclo de vida.
- Grupo de actividades de predesarrollo. Estas actividades exploran conceptos y distribuyen los requisitos del sistema antes de que el desarrollo de software pueda comenzar.
- Grupo de actividades de desarrollo. Son las actividades que se realizan durante el desarrollo de un producto de software.
- Grupo de actividades de posdesarrollo. Son las actividades que se realizan para instalar, operar, apoyar, mantener y retirar un producto de software.
- Grupo de actividades integrales. Actividades necesarias para asegurar la culminación exitosa de un proyecto. Generalmente se les considera apoyo de las actividades, y no actividades orientadas directamente a los esfuerzos de desarrollo.

Cuando se hace uso del estándar 1074-1997, relacionado con los procesos del ciclo de vida del software, es necesario considerar para esta etapa el subgrupo actividades de implementación, que forma parte del grupo de actividades de desarrollo, como se ve a continuación.

Grupo de actividades de desarrollo

Las actividades de esta agrupación se subagrupan en…

- Actividades de requerimientos. Siguen el desarrollo de los requerimientos totales del sistema, y la distribución funcional de dichos requisitos entre el hardware, el software y las personas.
- Actividades de diseño. Desarrollan una representación coherente y bien organizada del sistema de software acorde con los requerimientos de software.
- Actividades de implementación. Resuelven la transformación de la representación detallada de diseño de un producto de software en código generado en un lenguaje de programación. Las actividades específicas de este grupo son:
    - Crear código ejecutable. Se generará el código fuente, incluyendo los comentarios adecuados, tal como se especifica en el diseño detallado del software.
    - Crear documentación operativa. Se genera la documentación operativa del proyecto de software a partir del diseño detallado del software y de los requerimientos de interfaz del software, en concordancia con la información de planificación documentada. Esta

documentación operativa se requiere para instalar, operar y dar soporte al sistema a lo largo del ciclo de vida.

- Realizar la integración. Esta actividad debe integrar la base de datos, el código fuente (si es necesario), el código ejecutable, rutinas y *drivers*, según se especifica en la información planeada para la integración del software integrado. Si el sistema incluye tanto componentes de hardware como de software, la integración del sistema puede incluirse como parte de esta actividad.

4.3. Diseño y modelo de componentes

Un componente se considera una implementación “opaca”, debido a que se distribuye predominantemente en formato binario y a que sus consumidores lo utilizan a través de su interfaz como una “caja negra”, lo que lo alinea con el principio de encapsulamiento de la programación orientada a objetos.

La composición por terceros implica que los componentes son intrínsecamente reutilizables; esto es, un sistema basado en componentes puede ser ensamblado con relativa facilidad por un integrador, empleando componentes suministrados por múltiples proveedores independientes.

Finalmente, la coordinación e interacción entre componentes exige un marco regulatorio estandarizado (modelo de componentes) que establece la infraestructura de software requerida (framework) y las convenciones y restricciones de diseño de los mismos.

Frameworks y modelos de componentes

Los sistemas basados en componentes confían en estándares y convenciones bien definidas (modelo de componentes) y en una infraestructura de soporte (*framework* de componentes).

Según Montilva, Arapé y Colmenares (2003), los modelos de componentes especifican las reglas de diseño que deben obedecer los componentes. Estas reglas de diseño mejoran la composición, aseguran que las calidades de servicio se logren en todo el sistema y que los componentes se puedan desplegar fácilmente, tanto en entornos de desarrollo como de producción.

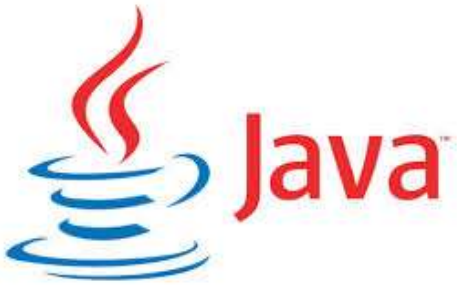
Los modelos de componentes imponen estándares y convenciones sobre…

- Tipos de componentes: un tipo de componente puede ser definido en términos de las interfaces que implementa. Tipos diferentes de componentes pueden desempeñar diferentes roles en el sistema y participar en distintos tipos de esquemas de interacción.
- Esquemas de interacción: especifican cómo los componentes son localizados, cuáles protocolos de comunicación son utilizados y cómo se satisfacen las calidades de servicio (ej. seguridad, transacciones, alta disponibilidad). El modelo de componentes puede describir cómo interactúan los componentes entre sí y cómo interactúan dichos componentes con el *framework*.
- Asociación (*bindings*) de recursos: el proceso de composición de componentes consiste en simplemente enlazar un componente a uno o más recursos. Un recurso es un servicio ofrecido por un *framework* o por

otro componente desplegado en ese *framework*. Un modelo de componentes describe cuáles recursos están disponibles a los componentes, y cómo y cuándo se asocian estos componentes a estos recursos.

Los *frameworks* de componentes proporcionan servicios que soportan y hacen cumplir el modelo asociado. El *framework* maneja recursos compartidos por los componentes y proporciona mecanismos subyacentes que permiten la comunicación (interacción) entre ellos. Los *frameworks* son activos y actúan directamente sobre componentes; por ejemplo, administrando su ciclo de vida (comenzar, suspender, reasumir o terminar la ejecución de un componente) y otros recursos.

Existen muchos ejemplos de *frameworks* de componentes, como Enterprise JavaBeans y VisualBasic Framework de Microsoft.



Un mercado robusto de componentes de software requiere modelos y *frameworks* estandarizados. Sin embargo, la experiencia ha demostrado que distintos dominios de aplicación tienen diferentes requisitos de funcionalidad y calidad de servicio. Esto sugiere la necesidad de tener una variedad de modelos y *frameworks* de componentes para cada dominio.

Tipos de componentes

Los tipos de componentes se categorizan de acuerdo con su interfaz. La interfaz define el conjunto de operaciones/servicios/responsabilidades que un componente puede realizar. Las interfaces proveen un mecanismo para interconectar componentes y controlar las dependencias entre ellos.

La naturaleza de la interfaz varía dependiendo del lenguaje de programación empleado para implementar el componente.

- Los lenguajes orientados a objetos (ej. Smalltalk-80, C++ y Java) soportan alguna forma de interfaz, que por lo general está separada de las implementaciones.
- En lenguajes procedimentales (ej. Pascal y FORTRAN) las interfaces se definen a través de declaraciones de funciones o procedimientos y el uso de variables globales.
- Algunos lenguajes neutrales de especificación de interfaces han sido desarrollados, tales como IDL (*Interface Definition Language*) de OMG (*Object Management Group*).

Algunas tecnologías (ej. Enterprise JavaBeans) exigen que sus componentes implementen dos tipos de interfaces:

- Interfaz de negocio: que refleja el rol del componente en el sistema.
- Interfaz de infraestructura: es impuesta por el modelo de componentes con el fin de permitirle al *framework* interactuar con el componente.

Esquemas de interacción

Existen tres clases principales de interacción en los sistemas basados en componentes:

1. Componente-Componente (C-C): permite la interacción entre componentes. De este tipo de interacción se obtiene la funcionalidad de la aplicación, de forma tal que los contratos que especifican este tipo de interacción pueden ser clasificados como contratos a nivel de aplicación.

2. Componente-Framework (C-F): posibilita las interacciones entre el *framework* y sus componentes. Dicha interacción permite que el *framework* administre los recursos de los componentes. Los contratos que especifican estas interacciones pueden ser clasificados como contratos a nivel de sistema.

3. Framework-Framework (F-F): posibilita las interacciones entre *framewoks* y permiten la composición de componentes desplegados en *framewoks* heterogéneos. Estos contratos pueden ser clasificados como contratos de interoperabilidad.

Asociación

La forma de materializar la composición entre componentes depende de los mecanismos especificados por su modelo de programación.

Típicamente, los modelos de componentes se basan en tecnologías orientadas a objetos; por lo tanto, los mecanismos de composición emplean algunas características, tales como relaciones entre clases (especialización, agregación, asociación y uso), polimorfismo y enlace dinámico. Adicionalmente, dichos mecanismo de composición típicamente se describen mediante el uso de patrones de diseño.

4.4. Técnicas de implementación

Programación modular

La programación modular consiste en subdividir un programa en subprogramas independientes (funciones y subrutinas). Estos subprogramas hacen que el programa principal sea más corto y fácil de leer y entender. Adicional a ello, los argumentos de paso entre el programa principal y los subprogramas muestran exactamente qué información utilizará el subprograma, lo que facilita la revisión cuando se pide un cambio en el programa.

Los subprogramas también facilitan que otras personas comprendan el funcionamiento del programa y reducen la probabilidad de errores en el programa. Puesto que los subprogramas pueden utilizar variables locales, es menos probable que un cambio en el código del subprograma interfiera con todo el programa o con otros subprogramas. También el pequeño tamaño de los subprogramas facilita la comprensión de los efectos globales del cambio de una variable.

Programación estructurada

Conjunto de técnicas que usan un número limitado de estructuras de control, que reduce la complejidad de los problemas y minimiza los errores. Esta programación es un conjunto de técnicas que incorpora:

- Diseño descendente (*top-down*). Proceso en el cual el problema se descompone en una serie de niveles o pasos sucesivos.
- Recursos abstractos. Descomposiciones de acciones complejas en acciones más simples.
- Estructuras básicas. Un programa propio puede ser escrito utilizando sólo tres tipos de estructuras de control:

Podemos definir un programa como estructurado si cumple con las siguientes características:

- Tiene un sólo punto de entrada y uno de salida o fin de control del programa.
- Existen caminos desde la entrada hasta la salida que se pueden seguir y que pasan por todas partes del programa.
- Todas las instrucciones son ejecutables y no existen lazos o bucles infinitos (sin fin).

Programación orientada a objetos

Conjunto de técnicas que considera a los objetos como los elementos principales de construcción.

Los conceptos generales más utilizados en el modelo orientado a objetos son:

- Abstracción. Descripción simplificada de un sistema que hace énfasis en algunos detalles significativos y suprime los irrelevantes.

- Encapsulamiento. La información acerca de un objeto está encapsulada por su comportamiento.
- Modularidad. División de un programa en partes, llamadas módulos, los cuales pueden trabajarse por separado.

Los conceptos generales más utilizados en la programación orientada a objetos son:

- Objeto. Conjunto de localidades en memoria con un conjunto de subprogramas (métodos) que definen su comportamiento y un identificador asociado. Un objeto tiene propiedades (un estado) y un comportamiento.
- Clase. La clase representa la esencia del objeto (entidad que existe en el tiempo y el espacio). La clase tiene dos vistas: la exterior, que hace énfasis en la abstracción y se oculta en la estructura, y la vista interior o implementación.
- Método. El código que actúa sobre los datos.
- Envío y recepción de mensajes. Petición para que un objeto se comporte de una determinada manera, ejecutando una de sus funciones miembro.

- Herencia. Define relaciones entre clases, donde una clase comparte la estructura o comportamiento definidos en una o más clases.
- Polimorfismo. Capacidad de tener métodos con el mismo nombre, pero con implementación diferente.

4.5. Depuración y métodos para revisar el código

Pruebas

Las pruebas del software representan una revisión final de las especificaciones, del diseño y de la codificación, y son un elemento crítico para la garantía de calidad del software.

Objetivos de las pruebas

- La prueba es el proceso de ejecución de un programa con la intención de descubrir un error.
- Una prueba tiene éxito si descubre un error no detectado hasta entonces.

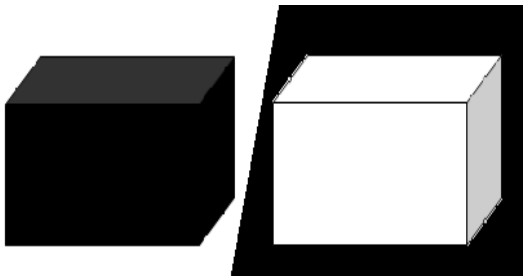
Si la prueba se lleva a cabo con éxito (de acuerdo con el objetivo anteriormente establecido), descubrirá errores en el software. Como ventaja secundaria, la prueba demostrará hasta qué punto las funciones del software parecen operar de acuerdo con las especificaciones y parecen alcanzar los requisitos de rendimiento. Adicionalmente, los datos que se van recogiendo a medida que se lleva a cabo la prueba, proporcionan una buena indicación de la fiabilidad del software, y de alguna manera, indican la calidad del software como un todo.

Principios de las pruebas

- *A todas las pruebas se les debería poder hacer un seguimiento hasta los requisitos del cliente*. Se entiende que los defectos más graves (desde el punto de vista del cliente) son aquellos que impiden al programa cumplir sus requisitos.
- *Las pruebas deberían planificarse mucho antes de que empiecen*. La definición detallada de los casos de prueba puede empezar tan pronto como el modelo de diseño esté consolidado.
- *El principio de Pareto es aplicable a la prueba del software.* Implica que al 80 por 100 de todos los errores descubiertos durante las pruebas se les puede hacer un seguimiento hasta un 20 por 100 de todos los módulos del programa.
- *Las pruebas deberían comenzar por «lo pequeño» y progresar hacia «lo grande».* A medida que avanzan las pruebas, desplazan su mirada en un intento de encontrar errores en grupos integrados de módulos y finalmente en el sistema entero.
- *No son posibles las pruebas exhaustivas.* Cubrir adecuadamente la lógica del programa y asegurarse de que se han aplicado todas las condiciones en el diseño a nivel de componente.
- *Para ser más eficaces, las pruebas deberían ser realizadas por un equipo independiente*. El ingeniero de software que creó el sistema no es el más adecuado para llevar a cabo las pruebas para el software.

Cualquier producto de ingeniería (y de muchos otros campos) puede probarse de una de estas dos formas:

- Prueba de caja negra. Conociendo la función específica para la que fue diseñado el producto, se pueden llevar a cabo pruebas que demuestren que cada función es completamente operativa y al mismo tiempo buscando errores en cada función.



- Prueba de caja blanca. Conociendo el funcionamiento del producto, se pueden desarrollar pruebas que aseguren que «todas las piezas encajan»; es decir, que la operación interna se ajusta a las especificaciones y que todos los componentes internos se han comprobado de forma adecuada.

Existen diversas pruebas de caja blanca, entre las cuales destacan las siguientes:

❖ *Prueba del camino básico*

El método del camino básico permite al diseñador de casos de prueba obtener una medida de la complejidad lógica de un diseño procedimental y usar esa medida como guía para la definición de un conjunto básico de caminos de ejecución. Los casos de prueba obtenidos del conjunto básico garantizan que durante la prueba se ejecuta por lo menos una vez cada sentencia del programa.

❖ *Prueba de condición*

Método de diseño de casos de prueba que ejercita las condiciones lógicas contenidas en el módulo de un programa. Se centra en la prueba de cada una de las condiciones del programa. El propósito de la prueba de condiciones es detectar, no sólo los errores en las condiciones de un programa, sino también otros errores en dicho programa.

Si un conjunto de pruebas de un programa es efectivo para detectar errores en las condiciones en que se encuentra éste, es probable que el conjunto de pruebas también sea efectivo para detectar otros errores en el programa.

- *Prueba de flujo de datos*

El método de prueba de flujo de datos selecciona caminos de prueba de un programa de acuerdo con la ubicación de las definiciones y los usos de las variables del programa.

- *Prueba de bucles*

La prueba de bucles es una técnica de prueba de caja blanca que se centra exclusivamente en la validez de las construcciones de ciclos. Se pueden definir cuatro clases diferentes de bucles: bucles simples, bucles concatenados, bucles anidados y bucles no estructurados.

Prueba de unidad

Inicialmente, la prueba se centra en cada módulo individualmente, asegurando que funciona adecuadamente como una unidad, por lo que se le conoce como prueba de unidad. La prueba de unidad hace un uso intensivo de las técnicas de prueba de caja blanca, ejercitando caminos específicos de la estructura de control del módulo, para asegurar un alcance completo y una detección máxima de errores.

El proceso de pruebas de unidad se conforma de las siguientes actividades:

- Planificación general. En esta fase se especifican las unidades que se van a probar, el orden en que se van a probar, las características que serán consideradas en las pruebas, también las que no serán consideradas, y por último las especificaciones de las unidades que serán empleadas.

- Diseño de casos de prueba de la clase. El objetivo es obtener un conjunto representativo de casos de prueba, que sin embargo sea el más pequeño posible. Esto con el fin de que el tiempo requerido para realizar las pruebas sea el mínimo posible.

- Planificación de la ejecución de casos de prueba. Para planificar la ejecución de los casos de prueba es conveniente agruparlos con base en la similitud de las condiciones de las pruebas para organizar sesiones de prueba de grupos de casos que requieran la misma preparación o una muy similar. Para cada sesión se deben describir explícitamente las actividades de preparación para establecer y cumplir esas condiciones de las pruebas, y para determinar y medir los resultados obtenidos de manera que se puedan evaluar los criterios de fallo. De acuerdo con los facilitadores (personal del proyecto) y los desarrolladores se planifica la ejecución de las sesiones de prueba, registrando los números (o nombres) de los casos de prueba y la fecha y hora planificadas.
- Ejecución de casos de prueba. De acuerdo con lo registrado, se preparan –la anticipación de la preparación depende de los recursos que se necesiten– y ejecutan los casos de prueba, y los detalles de los resultados obtenidos se registran junto con la especificación de caso de prueba. Se registran la fecha de ejecución, si falló S/N, quién supervisará, y si es el caso se describe el fallo o se anotan algunas observaciones. Una vez se ejecuten los casos de prueba se elabora el resumen de la ejecución de los casos de prueba, en donde se totaliza el número de casos planificados ejecutados y con fallo, y el porcentaje de fallo.

Depuración

La depuración es la corrección de errores que sólo afectan a la programación, porque no provienen de errores previos en el análisis o en el diseño. A veces la depuración se hace luego de la entrega del sistema al cliente y es parte del mantenimiento.

En realidad, en las revisiones de código y las pruebas unitarias, encontrar un error es considerablemente más sencillo, ya que se hace con el código a mano. Aun cuando se hubiera optado por una prueba unitaria de caja negra, es sencillo recorrer el módulo que revela un comportamiento erróneo por dentro para determinar el lugar exacto del error.

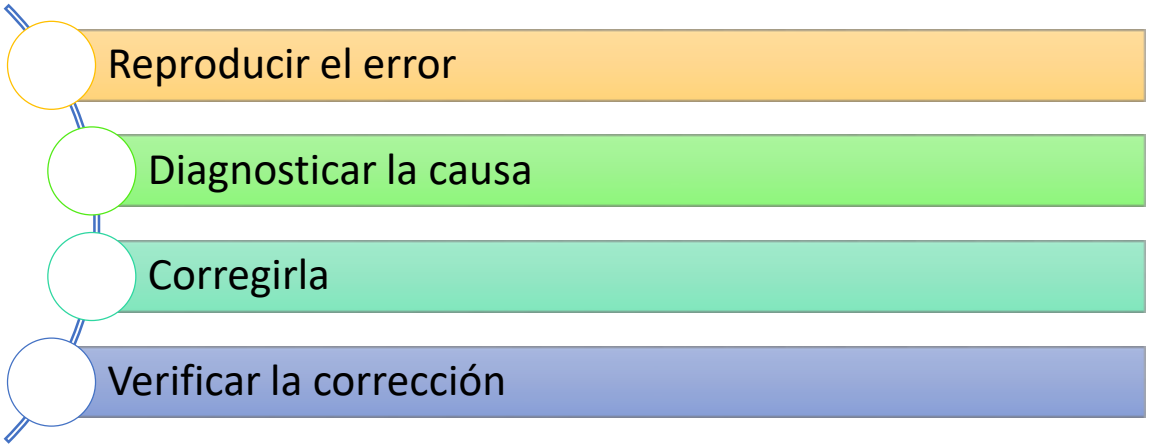
Existen incluso herramientas de depuración (en inglés, *debugging*) de los propios ambientes de desarrollo que facilitan esta tarea, que incluso proveen recorrido paso a paso y examen de valores de datos.

A fin de cuentas, lo importante es analizar correctamente si el error está donde parece estar o proviene de una falla oculta más atrás en el código. Para encontrar estos casos más complejos son útiles las herramientas de recorrida hacia atrás, que permiten partir del lugar donde se genera el error y recorrer paso a paso el programa en sentido inverso.

Las pruebas de integración, de sistema y de aceptación también pueden llevar a que sea necesaria una depuración, aunque aquí es más difícil encontrar el lugar exacto del error. Por eso, a menudo se utilizan bitácoras (*logs*, en inglés), que nos permiten evaluar las condiciones que se fueron dando antes de un error mediante

el análisis de un historial de uso del sistema que queda registrado en medios de almacenamiento permanente.

La depuración se hace en cuatro pasos:



Si el error no se repite al intentar reproducirlo es muy difícil hacer el diagnóstico. Como en casi todas las ciencias, se buscan causas y efectos, condiciones necesarias y suficientes para que se produzca el error. Luego hay que buscar el sector del código en donde se produce el error, lo que nos lleva a las consideraciones hechas recientemente. La corrección del error entraña mucho riesgo, porque a menudo se introducen nuevos errores (hay quienes hablan de tasas de 0.2 a 0.5 nuevos errores por corrección), y nunca hay que olvidarse de realizar una nueva verificación después de la corrección.

RESUMEN

A lo largo de esta unidad se presentaron definiciones de componente, sus características clave, como identificable, autocontenido, etc., los ámbitos necesarios para identificarlo (dominio, situación y programa), y las variables para clasificarlo (como tamaño, complejidad, reusabilidad, etc.).

Posteriormente se abordó el tema de estándares y buenas prácticas de implementación, en el cual se describió brevemente lo que se entendía por estándar y buena práctica de codificación, y se detallaron algunos estándares generales de codificación (claridad y consistencia, formato y estilo, etc.).

Asimismo, en este apartado se revisó el estándar 1074-1997, que proporciona un procedimiento para crear un proceso de ciclo de vida del software, analizándose en específico el grupo de actividades de desarrollo y dentro de él las actividades de implementación.

En el tercer tema se abordaron los conceptos de modelo de componentes, entendidos como el conjunto de reglas de diseño que deben obedecer los componentes, y de *framework* de componentes, que se constituye como su infraestructura de soporte.

También se profundizó en el tema de interfaz de los componentes, que es el elemento que permite categorizarlos. El cuarto tema presentó las características principales de la programación modular, la programación estructurada y la programación orientada a objetos. Finalmente, en el tema depuración y métodos para revisar el código, se hizo mención de las pruebas, sus objetivos, principios y tipos principales, y del proceso de depuración.

BIBLIOGRAFÍA

Bibliografía básica

Andrews, A., Ghosh, S. y Choi, E. M. (2002). A model for understanding software components. En *Software maintenance, 2002. Proceedings* (pp. 359-368). International Conference on. IEEE.

Ariza, M. y Molina, J. C. (2004). *Introducción y principios básicos del desarrollo de software basado en componentes.* Colombia: Pontificia Universidad Javeriana. Disponible en http://pegasus.javeriana.edu.co/~jcpymes/Docs/DSBC.pdf

Computer Science Department. (2007). *C++ Coding Standard*. Missouri University of Science and Technology. Disponible en http://web.mst.edu/~cpp/cpp_coding_standard_v1_1.pdf

Gamma, E., Helm, R., Johnson, R. y Vissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. USA: Pearson Education. Disponible en http://books.google.es/books?id=6oHuKQe3TjQC&lpg=PT2&ots=lNrLDTfMKB&dq=Design%20Patterns%20Elements%20of%20Reusable%20Object-Oriented%20Software&lr&pg=PT2#v=onepage&q=Design%20Patterns%20Elements%20of%20Reusable%20Object-Oriented%20Software&f=false

Ge, J. (2014). *All-in-one code framework coding standards*. Microsoft All-In-One Code Framework Project Team. Disponible en https://www.google.com.mx/url?sa=t&rct=j&q=&esrc=s&source=web&cd=4&ved=0CEUQFjAD&url=http%3A%2F%2Flwoodyiii.files.wordpress.com%2F2012%2F10%2Fall-in-one-code-framework-coding-standards.docx&ei=u8gMU-OiFOaayQG87IDAAw&usg=AFQjCNHdH1k2WIXNG8Mz6Ro10c-MqMrezA&sig2=NrfT8TStJ-Oks0pfGQkGMQ

Montilva, J. A., Arapé, N. y Colmenares, J. A. (2003). *Desarrollo de software basado en componentes.* Actas del IV Congreso de automatización y control. Mérida, Venezuela. Disponible en http://webdelprofesor.ula.ve/ingenieria/jonas/Productos/Publicaciones/Congresos/CAC03%20Desarrollo%20de%20componentes.pdf

Pressman, R. (2002). *Ingeniería de software. Un enfoque práctico* (5ª ed.). España: McGraw-Hill.

UNIDAD 5

Integración de subsistemas y sistemas

OBJETIVO PARTICULAR

Integrar los subsistemas de un sistema y validar su buen funcionamiento.

TEMARIO DETALLADO
(10 horas)

5. Integración de subsistemas y sistemas

- 5.1. Tipos de integración de sistemas
- 5.2. Pruebas de integración
- 5.3. Métricas para medir la calidad de sistemas
- 5.4. Generación de documentación

INTRODUCCIÓN

La integración de sistemas, es la actividad culminante dentro del desarrollo de software. Llevarla a cabo requiere de la comprensión de los procesos de integración mismos, de sus estrategias y técnicas. Asimismo, es necesario identificar los aspectos relacionados con las pruebas de integración, como elemento de verificación que permite una integración lo más estable posible, con el apoyo de las pruebas de integración ascendente o descendente, las pruebas de regresión y la prueba de humo.

Dentro de este esquema de operación es importante considerar las métricas para cuantificar de alguna manera la calidad del software, por ello se aborda el proceso de medición los principios por los que se rige, así como los factores que afectan la calidad del software y las métricas para expresar dichos factores, y finalmente identificar el estándar ISO 9126 para la evaluación de la calidad del software.

Finalmente, todo proceso de integración no está completo si no se considera la documentación, como una evidencia que el sistema sea accesible para usuarios y para las personas que le darán mantenimiento.

5.1. Tipos de integración de sistemas

Integrar significa conformar un todo a partir de sus partes. En el desarrollo de sistemas, la integración es la etapa mediante la cual todos los módulos o componentes se organizan y disponen para conformar el sistema informático.

Los cuatro tipos de integración que existen son:

- *Integración horizontal.* No considera el nivel jerárquico por lo que la información del sistema será conocida por todas las áreas. Esto es común en organizaciones cuya estructura no es muy formal.

- *Integración vertical*. Considera la estructura orgánica y niveles jerárquicos de la organización. Garantiza que la información sea única y exclusivamente para quien corresponde.

- *Integración física*. Consiste en el agrupamiento de hardware (equipos) y los componentes de comunicaciones que exigen para llevar a cabo la integración funcional. La integración funcional significa que el usuario puede tener acceso con mucha facilidad mediante una interfaz lógica, sencilla y pueda conmutar un recurso con otro. Los tres elementos de integración física son:
    - Interfaz de usuario
    - Acceso a los servicios externos
    - Interconexión física

- *Integración al medio externo o al interno*. Tiene como objetivo proporcionar a las instituciones datos del exterior, es decir, poner a la organización en comunicación con otras, con la finalidad de compartir información.

Estrategias de integración

Las estrategias de integración permiten construir la estructura del programa y realizar pruebas para detección de errores asociados con la interacción. El objetivo es tomar los módulos que ya han sido probados en unidad para construir una estructura integrada que concuerde con lo que dicta el diseño. Las estrategias de integración son:

- *Integración no incremental*. Consiste en combinar todos los módulos por anticipado y probarlos a manera de conjunto. Por lo general siempre se termina con el número de errores más grande posible, y la eliminación de los mismos resulta complicada, puesto que se tiene la vasta extensión del programa completo.

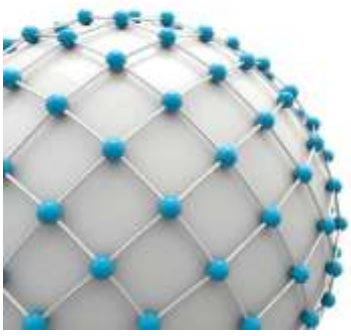
- *Integración incremental*. Consiste en tomar los módulos o componentes del sistema y probarlos por separado, a fin de facilitar el aislamiento de errores y su corrección. Es probable inclusive que se puedan probar las interfaces de manera completa y realizar pruebas sistemáticas. Esta estrategia se subdivide en dos categorías:
    - *Integración incremental descendente*. Consiste en integrar los módulos mediante la jerarquía de control, es decir, adoptando un movimiento hacia abajo iniciando con el programa principal. Este tipo de integración se la puede realizar de dos formas:
        - Primero-en–profundidad. Va integrando los módulos de un camino de control hasta llegar al módulo final, si existe un módulo que esté al mismo nivel jerárquico que otro, y este es necesario para que el módulo superior funcione adecuadamente, este debe ser integrado antes de pasar al nivel jerárquico inferior. El camino a integrar debe ser escogido de alguna forma, arbitraria y dependerá de las características específicas de la aplicación.

        - Primero-en-anchura. Integra primero todos los módulos pertenecientes a un nivel jerárquico, comenzando por los niveles superiores, es decir, en esta estrategia de integración hay que moverse en la estructura en forma horizontal.
    - Integración incremental ascendente. empieza integrando los módulos atómicos, es decir, aquellos módulos que se encuentran en el nivel más bajo de la estructura del sistema. En esta estrategia, debido a que los módulos se integran de abajo hacia arriba, el proceso requerido de los módulos subordinados siempre estará disponible por lo que se elimina la necesidad de resguardos.

- *Integración mixta*. Utiliza una combinación de las estrategias descendente y ascendente, esto depende de las características del software y del plan del proyecto en muchos casos. La integración mixta ha sido denominada también como estrategia de integración sándwich.

Adicional a lo anterior, existen tres técnicas de integración para conectar aplicaciones:

- *Base de datos maestra*. Implica que todas las aplicaciones almacenen y recuperen datos de una base de datos Maestra. Cada aplicación opera sobre la propia base de datos maestra, por lo que dicha base de datos integra completamente las aplicaciones individuales.
- *Interfaces punto a punto*. Consiste en conectar un par de aplicaciones directamente. La técnica de integración punto a punto no requiere de un modelo maestro. Las interfaces pueden ser programas separados que se ejecutan completamente por separado.



- *Integración indirecta*. En esta estrategia, la base de datos maestra puede permanentemente almacenar datos (redundante con datos de la aplicación), o puede simplemente servir como un área de traspaso para copiar datos entre aplicaciones. Este tipo de integración no requiere modificaciones en las aplicaciones.

Integración de componentes

La integración de componentes de distintos orígenes requiere de llegar a un acuerdo común en el que se establezcan los mecanismos necesarios para que esa integración se haga efectiva.

Por tanto es necesario especificar además del lenguaje de programación en el que se desarrolló el componente cuáles son sus puntos de acceso (funciones), y establecer los mecanismos de comunicación entre componentes.

En este sentido, y buscando satisfacer esa necesidad de mecanismos estándar e interfaces abiertas, los esfuerzos que más han sobresalido son:

1.- Microsoft COM y DCOM.

A principios de los años 90, Microsoft realizó un gran esfuerzo en promover OLE (Object Linking and Embedding) basado en la necesidad de los usuarios de elaborar documentos que incluyeran texto, gráficas y tablas, ya que el grado de especialización de las aplicaciones no lo permitía, por lo que debía crear mecanismo estándar de empaquetamiento de componentes si quería que OLE fuera exitoso.

Era crucial que estos componentes se pudieran desarrollar con cualquier lenguaje e incrustarse en otro componente escrito en otro lenguaje. Por ello, Microsoft creó COM (Component Object Model - modelo de objetos componentes), que es un estándar para la creación y comunicación entre componentes dentro de una máquina. A finales de 1996, Microsoft introdujo DCOM (La D es de distribuido), un mecanismo de integración de Microsoft donde se pueden utilizar los componentes creados en aplicaciones basadas en COM y trasladarlos a entornos distribuidos. Ha existido una lentitud en que aparezca COM en plataformas diferentes a Windows, por ello, se identifica a COM como una arquitectura de componentes en lugar de una arquitectura de componentes remotos o distribuidos; sin embargo, COM tiene mucho que ofrecer en este sentido, especialmente cuando se trabaja en ambientes Windows. DCOM resuelve distintos conflictos de diseño que aparecen cuando se comienza a implementar una aplicación distribuida en una red, como:

- Los componentes que interactúan más a menudo deberían estar localizados más cerca.
- Algunos componentes solo pueden ser ejecutados en máquinas específicas o lugares específicos.
- Los componentes más pequeños aumentan la flexibilidad, pero incrementan el tráfico de red.
- Los componentes grandes disminuyen el tráfico de red, pero también reducen la flexibilidad.

Con la independencia de localización de DCOM, la aplicación puede combinar componentes relacionados en máquinas "cercanas" entre sí, en una sola máquina o incluso en el mismo proceso. DCOM a su vez es independiente del lenguaje ya que virtualmente cualquier lenguaje puede ser utilizado para crear componentes

COM, y estos componentes pueden ser utilizados por muchos más lenguajes y herramientas, por lo que no se necesita un lenguaje específico para crear componentes COM.

2. CORBA.

CORBA (Common Object Request Broker Arquitecture) es un conjunto de estándares que forman un marco de referencia para establecer la interacción de los componentes. Fue elaborado por el OMG (Object Management Group), una institución formada por la mayoría de las compañías más importantes de hardware y software del mundo, que tiene por objetivo promover asuntos de orientación a objetos y la promoción de estándares abiertos para sistemas orientados a objetos.
CORBA no es un producto sino solo un estándar. Las especificaciones se escriben en un lenguaje de definición neutral (Interfase Definition Language - IDL) que define la frontera de un componente, es decir, su interface con clientes potenciales.

CORBA fue diseñado para resolver el problema de intercomunicar máquinas, por lo que es una arquitectura ad hoc para manejar componentes remotos o distribuidos.

La especificación CORBA establece una infraestructura para que exista una comunicación entre procesos dentro de la misma máquina o entre diferentes máquinas.

El empaquetamiento de componentes provee una interoperación entre lenguajes, plataformas y vendedores. Debido a que se ha instrumentado en un rango amplio de plataformas y vendedores (mainframe, Unix, Windows), CORBA es la arquitectura dominante en la distribución de objetos remotos.

Entre las características de CORBA están los servicios Middleware que son servicios de propósito general que se ubican entre plataformas y aplicaciones. Los servicios middleware se distinguen de aplicaciones finales y de servicios de plataformas específicas por cuatro importantes propiedades:

- Son independientes de las aplicaciones y de las industrias para las que éstas se desarrollan.
- Se pueden ejecutar en múltiples plataformas.
- Se encuentran distribuidos.
- Soportan interfaces y protocolos estándar.

CORBA está constituido esencialmente de tres partes: un conjunto de interfaces de invocación que son los medios por los cuales los clientes realizan la invocación sobre un objeto; el ORB (object request broker) que se constituye como la estructura por medio de la cual son invocados los métodos de los objetos y servicios; y un conjunto de adaptadores de objetos (objects adapters) que son módulos que permiten generar e interpretar las referencias a determinados objetos, invocar los métodos, asegurar la interacción entre objetos y métodos, activar y desactivar implementaciones y objetos, y relacionar referencias a objetos con sus implementaciones.

3. Common Gateway Interface (CGI)

> CGI es una interfaz al servidor Web que permite extender la funcionalidad del mismo. Con CGI se puede interactuar con los usuarios que acceden a un sitio en particular. A nivel teórico, CGI permite extender las capacidades del servidor para interpretar las entradas obtenidas del navegador y regresar la información apropiada de acuerdo a la entrada del usuario. A nivel práctico, CGI es una interfaz que facilita la escritura de programas para que se comuniquen fácilmente con el servidor.
> El protocolo CGI define una forma estándar para que los programas se comuniquen con el servidor Web (Fernández, 2006:10)

La comunicación CGI esta manejada sobre la entrada y salida estándar, lo que significa que si se conoce como imprimir en pantalla y leer datos utilizando el lenguaje de programación, se puede escribir una aplicación sobre el servidor Web.

Existen alternativas importantes para aplicaciones CGI. Ahora, muchos servidores incluyen una programación API (Application Programming Interface – Interfaz de programación de aplicaciones) que hace más fácil programar directamente extensiones al servidor en contra parte a separar aplicaciones CGI. Los APIs tienden a ser más eficientes que los programas CGI. Algunas aplicaciones se manejan por nuevas tecnologías desarrolladas en la parte del cliente (en lugar del servidor) tales como Java.

4. Java.

Es una arquitectura neutral, orientada a objetos, portable y un lenguaje de programación de alto desempeño que proporciona un ambiente de ejecución dinámica, distribuida, robusta, segura y multihilos.

La principal ventaja de Java para computación distribuida radica en la capacidad de descargar el ambiente. En términos de una arquitectura de objeto distribuido totalmente nueva, Java proporciona las siguientes opciones: Java Remote Method Invocation (RMI), Java IDL y la empresa JavaBEan. La especificación RMI (Remote Method Invocation – Invocación Remota de Métodos) es un API que nos permite crear objetos creados en lenguaje de programación Java, cuyos métodos se invocan de una Máquina Virtual Java diferente (JVM Java Virtual Machine).

RMI proporciona la capacidad para llamadas a métodos sobre objetos remotos, los cuales convierten al componente de transporte del objeto en arquitectura de objeto distribuido. También proporciona mecanismos para el registro y persistencia del objeto. Ofrece servicios distribuidos tales como Java IDL que proporciona una forma de conectar, transparentemente, a los clientes Java a los servidores de red utilizando la industria estándar: Lenguaje de Definición de Interfaces (IDL Interface Definition Language).

La tecnología Java IDL para objetos distribuidos facilita que los objetos interactúen a pesar de estar escritos en lenguaje de programación Java u otro lenguaje tal como C, C++, COBOL, entre otros. Java IDL está basado en CORBA y soporta la traducción para Java. Para soportar la interacción entre objetos en programas separados, Java IDL proporciona un ORB (Object Request Broker). El ORB es una librería clase que facilita la comunicación de bajo nivel entre las aplicaciones Java IDL y otras aplicaciones CORBA. Java RMI es una solución Java completa para objetos remotos que proporciona todas las ventajas de las capacidades de Java "una vez escrito, ejecutarlo en cualquier parte". Los servidores y clientes desarrollados con Java RMI se muestran en cualquier lugar en la red sobre cualquier plataforma que soporta el ambiente de ejecución de Java. Java IDL, en contraste, está basado en una industria estándar para solicitar, de manera remota, a objetos escritos en cualquier lenguaje de programación. Como resultado, Java IDL proporciona un medio para conectar aplicaciones "transferidas" que aún cubren las necesidades de comercio pero que fueron escritos en otros lenguajes.

5.2. Pruebas de integración

Recordemos los objetivos de las pruebas:

- La prueba es el proceso de ejecución de un programa con la intención de descubrir un error.
- Una prueba tiene éxito si descubre un error no detectado hasta entonces.

Asimismo, para estar en condiciones de revisar las pruebas de integración, recordemos los principios que orientan las pruebas:

- *Seguimiento hasta los requisitos del cliente*. Se entiende que los defectos más graves son aquellos que impiden al programa cumplir sus requisitos.
- *Planificación de pruebas*. La definición detallada de los casos de prueba puede empezar tan pronto como el modelo de diseño este consolidado.
- *El principio de Pareto.* Al 80% de todos los errores descubiertos durante las pruebas se les puede hacer un seguimiento de hasta un 20% de todos los módulos del programa.

- *Empezar de lo pequeño y progresar hacia lo grande.* A medida que avanzan las pruebas, desplazan su mirada en un intento de encontrar errores en grupos integrados de módulos y finalmente en el sistema entero.
- *Imposibilidad de pruebas exhaustivas.* Cubrir adecuadamente la lógica del programa y asegurarse de que se han aplicado todas las condiciones en el diseño a nivel de componente.
- *Pruebas independientes*. Un equipo independiente al de desarrollo debe generar y cubrir las pruebas.

A la facilidad con la que se puede probar un programa de computadora se le conoce como facilidad de prueba del software. Un conjunto de características que llevan a un software fácil de probar son:

- *Operatividad***.** Cuanto mejor funcione, más eficientemente se puede probar.
- *Observabilidad***.** Lo que ves es lo que pruebas.
- *Controlabilidad***.** Cuanto mejor podamos controlar el software, más se puede automatizar y optimizar.
- *Capacidad de descomposición*. Controlando el ámbito de las pruebas, podemos aislar más rápidamente los problemas y llevar a cabo mejores pruebas de regresión.
- *Simplicidad*. Cuanto menos haya que probar, más rápidamente podremos probarlo.
- *Estabilidad***.** Cuanto menos cambios, menos interrupciones a las pruebas.
- *Facilidad de comprensión*. Cuanta más información tengamos, más inteligentes serán las pruebas.

Teniendo estas consideraciones, se revisará lo que es la prueba de integración.

Prueba de integración

La prueba de integración es

> Una técnica sistemática para construir la estructura de un programa mientras que, al mismo tiempo, se lleva a cabo pruebas para detectar errores asociados con la interacción. El objetivo es tomar los módulos que han sido probados como unidades y construir una estructura de programa dictado por el diseño. (Visconti, Bidart y Mujica, 2008:12)

Prueba de Integración descendente

La prueba de integración descendente es un planteamiento incremental a la construcción de la estructura de programas (ver apartado de integración).

El proceso de pruebas de integración descendente se realiza en una serie de cinco pasos:

1.- Se usa el módulo de control principal como controlador de la prueba, disponiendo de resguardos para todos los módulos directamente subordinados al módulo de control principal.

2.- Dependiendo del enfoque de integración elegido (es decir, primero-en-profundidad o primero-en-anchura) se van sustituyendo uno a uno los resguardos subordinados por los módulos reales.

3.- Se llevan a cabo pruebas cada vez que se integra un nuevo módulo.

Tras terminar cada conjunto de pruebas, se reemplaza otro resguardo con el módulo real.

4.- Se hace la prueba de regresión para asegurarse de que no se han introducido errores nuevos.

5.- El proceso continúa desde el paso 2 hasta que se haya construido la estructura del programa entero.

Prueba de integración ascendente

Empieza la construcción y la prueba con los módulos atómicos (es decir, módulos de los niveles más bajos de la estructura del programa).

Se puede implementar la prueba de integración ascendente mediante los siguientes pasos:

1. Se combinan los módulos de bajo nivel en grupos que realicen una subfunción específica del software.

2. Se escribe un controlador (un programa de control de la prueba) para coordinar la entrada y la salida de los casos de prueba.

3. Se prueba el grupo.
4. Se eliminan los controladores y se combinan los grupos moviéndose hacia arriba por la estructura del programa.

Prueba de regresión

En el contexto de una estrategia de prueba de integración, la prueba de regresión es volver a ejecutar un subconjunto de pruebas ejecutadas anteriormente para asegurar que los cambios no han propagado efectos colaterales no deseados.

El conjunto de pruebas de regresión contiene tres clases diferentes de casos de prueba:

- Una muestra representativa de pruebas que ejercite todas las funciones del software
- Pruebas adicionales que se centran en las funciones del software que se van a ver probablemente afectadas por el cambio
- Pruebas que se centran en los componentes del software que han cambiado

Prueba de humo

Método de prueba de integración comúnmente utilizado cuando se ha desarrollado un producto de software empaquetado.

Es diseñado como un mecanismo para proyectos críticos en tiempo, permitiendo que el equipo de software valore su proyecto sobre una base sólida. En esencia, la prueba de humo comprende las siguientes actividades:

1.- Los componentes software que han sido traducidos a código se integran en una «construcción». Una construcción incluye ficheros de datos, librerías, módulos reutilizables y componentes de ingeniería que se requieren para implementar una o más funciones del producto.

2.- Se diseña una serie de pruebas para descubrir errores que impiden a la construcción realizar su función adecuadamente. El objetivo será descubrir errores «bloqueantes» que tengan la mayor probabilidad de impedir al proyecto de software el cumplimiento de su planificación.

Es habitual en la prueba de humo que la construcción se integre con otras construcciones y que se aplique una prueba de humo al producto completo (en su forma actual). La integración puede hacerse bien de forma descendente (top-down) o ascendente (bottom-up).

5.3 Métricas para medir la calidad de sistemas

Las métricas son medidas que se pueden emplear para valorar la calidad del producto según se va desarrollando. Estas medidas de atributos internos del producto le proporcionan al desarrollador de software una indicación en tiempo real de la eficacia del análisis, del diseño y de la estructura del código, la efectividad de los casos de prueba, y la calidad global del software a construir.

Proceso de medición

Un proceso de medición se puede caracterizar por cinco actividades:

1. *Formulación*: obtención de medidas y métricas del software apropiadas para la representación del software en cuestión.
2. *Colección*: Mecanismo empleado para acumular datos necesarios para obtener las métricas formuladas.
3. *Análisis*: Cálculo de las métricas y la aplicación de herramientas matemáticas.
4. *Interpretación*: Evaluación de los resultados de las métricas en un esfuerzo por conseguir una visión interna de la calidad de la representación.
5. *Realimentación (feedback):* Recomendaciones obtenidas de la interpretación de métricas técnicas transmitidas al equipo que construye el software.

Principios de medición

Los principios que se pueden asociar con la formulación de las métricas son los siguientes:

- Los objetivos de la medición deben establecerse antes de empezar la recogida de datos.
- Todas las técnicas sobre métricas deben definirse sin ambigüedades.
- Las métricas deben obtenerse basándose en una teoría válida para el dominio de aplicación.
- Hay que hacer las métricas a medida para acomodar mejor los productos y procesos específicos.
- Siempre que sea posible, la recogida de datos y el análisis debe automatizarse.
- Por encima de todo, intente obtener medidos técnicos simples. No se obsesione por lo métrico «perfecto» porque no existe.
- Se deben aplicar técnicas estadísticas válidas para establecer las relaciones entre los atributos internos del producto y las características externas de la calidad.
- Se deben establecer directrices de interpretación y recomendaciones para todas las métricas.

Métricas de calidad del software

Las métricas de calidad de software son un subconjunto de las métricas de software que se enfoca en los aspectos de calidad del producto. Se pueden dividir en métricas de producto, métricas del proceso y métricas del mantenimiento.

Métricas de producto.
Las métricas para medir la calidad del producto son:

- Calidad intrínseca del producto
  - Densidad de defectos
  - Tiempo entre fallos
- Calidad desde el punto de vista del usuario
  - Problemas del usuario
  - Satisfacción del usuario

La métrica de densidad de defectos se conforma por el número de defectos contra las oportunidades de error durante un periodo de tiempo específico. Como las fallas son los defectos materializados, se puede utilizar el número de casos que fallaron para dar un número aproximado de defectos del software. Para las oportunidades de error se considera el tamaño del sistema, expresado generalmente en término de líneas de código. Los períodos de tiempo pueden estar expresados en términos de lo que se denomina “vida del producto”.

Por otro lado, la métrica “tiempo entre fallos” es una medida básica de confiabilidad utilizada generalmente en sistemas críticos (como los que controlan vuelos). Se utiliza para medir el tiempo que se espera que un componente o todo el sistema se mantengan trabajando.

Otra métrica de calidad del producto utilizada por muchos desarrolladores mide la cantidad de problemas encontrados por el usuario durante el uso que le da a un producto. En este caso, todos los problemas encontrados durante el uso de un producto de software, es decir no sólo defectos válidos sino incluso errores de usuario, contra el tamaño del sistema.

La métrica de satisfacción del usuario, califica la satisfacción del usuario en el uso del sistema mediante la escala: Muy satisfecho, satisfecho, neutral, insatisfecho, muy insatisfecho.

Métricas del proceso

Las métricas en este rubro son:

- Densidad de defectos durante las pruebas. La densidad de defectos de software nunca sigue la distribución uniforme. Si una pieza de código o un producto tiene muchos defectos de pruebas, es resultado de una prueba más eficaz o de defectos mayores latentes en el código. Por tanto, la simple métrica de defectos por miles de líneas de código es un buen indicador de la calidad mientras que el software todavía está probándose. Es especialmente útil para controlar las versiones posteriores de un producto del mismo grupo de desarrollo ya que las comparaciones entre liberaciones no están contaminadas por factores externos.

- Patrón de detección de defectos durante las pruebas. Incluso con la misma tasa de defectos en las pruebas, los diferentes patrones de detección de defectos indican diferentes niveles de calidad. La meta en esta métrica es buscar detecciones de defectos que se estabilizan en un nivel muy bajo o los tiempos muy separados entre fallas, todo ello antes de terminar el periodo de prueba y liberarlo.
- Patrón para remoción de defectos basado en las fases. Complementa la métrica de densidad de defectos. Considera que se necesita dar seguimiento a los defectos en todas las fases del desarrollo, como el diseño, las inspecciones de código, etc. Puesto que muchos defectos se relacionan con

problemas de diseño, las validaciones de esta métrica mejoran la eliminación de defectos previo a las pruebas.

- Efectividad en la remoción de defectos. Es la proporción entre los defectos removidos durante la fase de desarrollo y los defectos latentes en el producto. Esta métrica puede calcularse para todo el proceso de desarrollo desde la vista de usuario y por cada fase. Cuanto más alto sea el valor de la métrica, más efectivo es el proceso de desarrollo y menos defectos llegan a la siguiente fase.

Métricas de mantenimiento

Las métricas de este rubro son:

- Listado de problemáticas no resueltas e índice de gestiones no hechas. Es un simple conteo de los problemas reportados no resueltos que permanecen en ese estado al final de cada mes o semana. Este indicador proporciona información útil para la gestión del proceso de mantenimiento pues les permite conocer sus cargas de trabajo.
- Tiempo de respuesta y capacidad de respuesta. La primera métrica se calcula como la media del tiempo de resolución de todos los problemas desde que se reportan hasta que se resuelven. Si los valores son muy extremos, entonces se utiliza la mediana en lugar de la media. Estos casos pueden presentarse cuando se trata de problemas poco relevantes por los cuales los clientes no reclamarán ni exigirán solución y por lo tanto el problema puede permanecer en estatus de pendiente por largo tiempo. Por otro lado, los elementos importantes de la capacidad de respuesta son las expectativas del cliente, el tiempo de respuesta acordado y la capacidad para cumplir con el compromiso con el cliente.
- Porcentaje de correcciones morosas. Una corrección morosa es aquella en la cual el tiempo de respuesta requerido supera en exceso la media. Esta métrica no es una medida para la gestión de correcciones morosas en tiempo

real, pues revisa solamente los problemas cerrados, de ahí que el número de correcciones morosas se comprueba al final de la semana.

- Mejorar la calidad. Esto se refleja con las correcciones de defectos del software. Esta métrica es simplemente el porcentaje de todas las correcciones de defectos en un período determinado. La corrección a los defectos puede registrarse de dos formas: cuando se descubrió o cuando se resolvió, considerando que lo más importante es evitar extender los tiempos de corrección. A partir de este conocimiento, se puede establecer el objetivo de calidad para el proceso de mantenimiento que es cero correcciones de defectos morosos.

5.4 Generación de documentación

La documentación es esencial para todo proceso de desarrollo de software, sin embargo, generalmente no se hace, no se le asigna presupuesto, y no se mantiene al día.

La documentación se clasifica en función de las personas o grupos a los cuales se dirige:

- *Documentación para los desarrolladores*. Es aquélla que se utiliza para el propio desarrollo del producto y, sobre todo, para su mantenimiento futuro. Se documenta para comunicar la estructura y comportamiento del sistema o de sus partes, para visualizar y controlar la arquitectura del sistema, para comprender mejor el mismo y para controlar el riesgo, entre otras cosas. Obviamente, cuanto más complejo es el sistema, más importante es la documentación. En este sentido, todas las fases de un desarrollo deben documentarse: requerimientos, análisis, diseño, programación, pruebas, etc. Una herramienta muy útil en este sentido es una notación estándar de modelado, de modo que mediante ciertos diagramas se puedan comunicar ideas entre grupos de trabajo.
- *Documentación para los usuarios*. La documentación para usuarios es todo aquello que necesita el usuario para la instalación, aprendizaje y uso del producto. Puede consistir en guías de instalación, guías del usuario, manuales de referencia y guías de mensajes. En el caso de los usuarios que son programadores, por ejemplo, los clientes de nuestra clase, esta documentación se debe acompañar con ejemplos de uso recomendados o de muestra y una reseña de efectos no evidentes de las bibliotecas. Buena

parte de la documentación para los usuarios puede empezar a generarse desde que comienza el estudio de requisitos del sistema. Esto está bastante en consonancia con las ideas de programación extrema (XP) y con metodologías basadas en casos de uso.

- *Documentación para los administradores o soporte técnico.* A veces llamada manual de operaciones, contiene toda la información sobre el sistema terminado que no hace al uso por un usuario final. Es necesario que tenga una descripción de los errores posibles del sistema, así como los procedimientos de recuperación. Como esto no es algo estático, pues la aparición de nuevos errores, problemas de compatibilidad y demás nunca se puede descartar, en general el manual de operaciones es un documento que va engrosándose con el tiempo.

Estructura de documentación

Un ejemplo de estructura de documentación es el que se presenta a continuación (MIT, 2001):

1. Requisitos
   - Visión general donde se especifican los fines del sistema y la forma en que los logra.
   - Especificación revisada. Son todas aquellas precisiones realizadas sobre los requisitos iniciales.
   - Manual de usuario. Documento que en forma de instrucciones muestra al usuario el uso de la aplicación.

- Funcionamiento. Hace una declaración de los recursos de hardware y software que requerirá el software así como un estimado de la disminución de capacidad de los mismos conforme se haga uso de él (como el espacio en disco duro)
- Análisis del problema. Es la presentación en forma de modelos de la cuestión que el software resolverá, sus relaciones con otras aplicaciones y datos así como las soluciones propuestas y la elegida.

2. Diseño

- Visión general del diseño. Es un panorama de cómo se diseñó la solución propuesta, sin considerar aspectos específicos. Adicionalmente puede incluir las librerías a utilizar, propias y de terceros y dilemas de diseño resueltos con sus pros y contras.
- Estructura en tiempo de ejecución. Se presenta en forma de modelo de los objetos o estructuras de programación que deberán ser utilizadas en el software.
- Estructura del módulo. Es presentada en forma de diagrama de dependencia entre módulos, de tal suerte que se especifique cuando un módulo hace uso de los servicios de otro.

3. Pruebas

La sección de pruebas de la documentación debe indicar las consideraciones que se tuvieron en cuenta para verificar y validar el sistema.

- Estrategia. En este apartado se debe describir la selección de pruebas empleadas entre la gran variedad que existe así como la justificación de la elección.
- Resultados del test. Es un concentrado de las pruebas realizadas al software respecto a estatus, defectos encontrados, defectos eliminados y defectos latentes.

4. Reflexión

Esta sección atiende a un ejercicio de autoevaluación de los pasos datos y las decisiones tomadas con el fin de que sirvan de base para futuros desarrollos.

- Evaluación. Es una relatoría de aciertos y errores durante evidenciados al cubrir las fases de desarrollo.
- Lecciones. Consiste en presentar lo aprendido de los aciertos y errores para beneficio de posteriores desarrollos.

5. Apéndice

Contiene elementos detallados del sistema que no se incluyen en la presentación general pero que sirven para verificarla.

- Formatos. Presentación de todos los formatos utilizados durante el desarrollo de todas las fases del proyecto.
- Especificaciones del módulo. Es la presentación del código fuente comentado de todo el software.
- Casos de prueba. La presentación de los casos de prueba en formato de fácil lectura/escritura.

RESUMEN

A lo largo de la presente unidad, se abordaron diversos temas relacionados con la integración de sistemas y subsistemas. En primer lugar se abordó el tema referente al proceso propio de integración, identificándose los tipos, estrategias y técnicas de integración, así como los marcos que facilitan la integración de componentes (DCOM, CORBA, CGI y Java).

En el segundo tema se abordaron los aspectos relacionados con las pruebas de integración, comenzando con un repaso de los objetivos de las pruebas y de los principios que las orientan, después se describieron los elementos que caracterizan al software fácil, los procesos de las pruebas de integración ascendente y descendente, lo que caracteriza a las pruebas de regresión y las actividades de la prueba de humo.

El tema tres describió cómo se desarrolla el proceso de medición y los principios del mismo, los factores que afectan la calidad del software y las métricas para expresar dichos factores y el estándar ISO 9126 para la evaluación de la calidad del software.

El último tema estableció los tipos de documentación que existen y una estructura a manera de ejemplo de cómo se organiza dicha documentación.

BIBLIOGRAFÍA

Referencias bibliográficas

Caraguay J., Casanova, Díaz F.A. (2013). *Análisis y diseño de integración de sistemas informáticos*. Tesis. Universidad Técnica del Norte. Ecuador. Disponible en: http://repositorio.utn.edu.ec/bitstream/123456789/1082/1/04%20ISC%20013-CAP%C3%8DTULOS.pdf.

Decker, R., & Hirshfield, S. (2001). *Programación con Java*. México, Thomson Learning.

Fernández, F. (2006) *La Red Cubana de Ciencia desde una perspectiva de su integración y componentes*. Ponencia presentada en Citmatel 2005. Cuba. Disponible en: http://www.bibliociencias.cu/gsdl/collect/eventos/index/assoc/HASH0e32.dir/doc.pdf.

Instituto Tecnológico de Massachusetts (MIT) (2001). *Curso práctico en Ingeniería de Software*. Profesores Jackson D., Devadas S. OpenCourseWare.

McCall, J. A., Richards, P. K. and Walters, G. F. (1977). Factors in Software Quality, Volumes I, II, andIII, US. Rome Air Development Center ReportsNTIS AD/A-049 014, NTIS AD/A-049 015 and NTIS AD/A-049 016, U.S. Department of Commerce. Disponible en: http://www.dtic.mil/dtic/tr/fulltext/u2/a049055.pdf

Pressman R. (2002). *Ingeniería de Software. Un enfoque práctico*. Mc. Graw Hill. 5ª ed. España.

Suarez P, Fontela C. (2003). *Documentación y pruebas*. Curso General. Facultad de Ingeniería. Universidad de Buenos Aires. http://campus.fi.uba.ar/pluginfile.php/109936/mod_resource/content/1/Documentacion_pruebas.pdf.

Visconti, M., Bidart, C., Mujica, J. (2002) Web Testing. Aspectos teóricos y prácticos. Documento para el Programa de Magister en Informática. Universidad Técnica Federico Santa María. Chile. Disponible en: http://www.inf.utfsm.cl/~visconti/testing/Documentos/WebTesting.pdf

Facultad de Contaduría y Administración
Sistema Universidad Abierta y Educación a Distancia