2019

Herramientas de productividad HDP115

UNIDAD I: ANTECEDENTES Y EVOLUCIÓN DE LAS HERRAMIENTAS INFORMÁTICAS

BLADIMIR DIAZ CAMPOS

UNIVERSIDAD DE EL SALVADOR | Facultad de Ingeniería y Arquitectura, Escuela de Sistemas Informáticos

Contenido

1. Evolución del proceso de desarrollo de software

La evolución de las herramientas de desarrollo de software, están determinadas entre otros factores, por la evolución de las capacidades del hardware; el aparecimiento de nuevas tecnologías; los cambios de paradigmas; métodos y técnicas de programación; el aparecimiento de nuevas metodologías de desarrollo de software y el aumento en la complejidad de los requerimientos del usuario.

Para comprender esta evolución es necesario revisar brevemente algunos de estos factores.

1.1. Evolución de los lenguajes de programación

Los lenguajes de programación surgen de la necesidad de reducir la complejidad y el tiempo necesario para darle instrucciones a la computadora. Veremos cómo, inicialmente las instrucciones eran dadas “en vivo” y de manera secuencial a las computadoras, lo que provocó la necesidad de crear lenguajes de programación que permitieran realizar estas tareas.

Al mismo tiempo que los lenguajes, surgen herramientas para la edición, interpretación y compilación del código fuente, y la generación de archivos ejecutables. Los lenguajes de programación, por ende, han determinado la forma en la que se construyen las herramientas informáticas para el desarrollo, y esto a su vez ha determinado la evolución de la forma en la que se desarrolla el software.

1.1.1 Primera generación: lenguaje de máquina

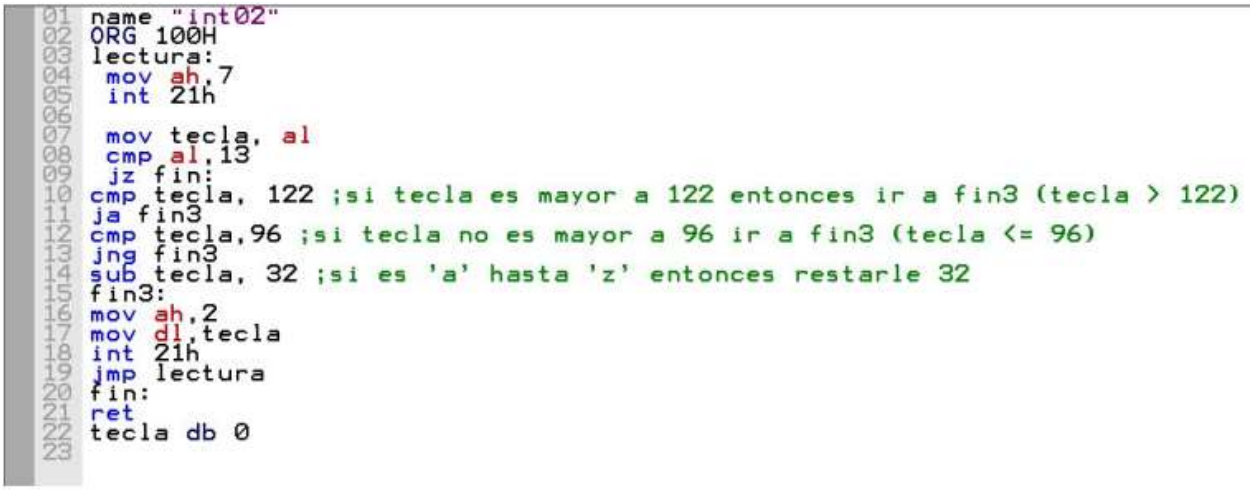
También llamado código de máquina. Tanto las instrucciones como los datos son escritos en sistema binario o hexadecimal, por lo que programar era una tarea muy compleja y lenta.

```
=======================================
[LINE]      LOC: MACHINE CODE
=======================================

[   1]         :
[   2]         :
[   3]     0100:
[   4]     0100: B4 07
[   5]     0102: CD 21
[   6]         :
[   7]     0104: A2 29 01
[   8]     0107: 3C 0D
[   9]     0109: 74 1D
[  10]     010B: 80 3E 29 01 7A
[  11]     0110: 77 0C
[  12]     0112: 80 3E 29 01 60
[  13]     0117: 7E 05
[  14]     0119: 80 2E 29 01 20
[  15]     011E:
[  16]     011E: B4 02
[  17]     0120: 8A 16 29 01
[  18]     0124: CD 21
[  19]     0126: EB D8
[  20]     0128:
[  21]     0128: C3
[  22]     0129: 00
[  23]         :

=======================================
```

Figura 1: Código de máquina en hexadecimal que convierte una letra leída del teclado en mayúscula

1.1.2 Segunda generación: Lenguajes de bajo nivel

El lenguaje de bajo nivel por excelencia es el lenguaje ensamblador. En este tipo de lenguajes, las instrucciones y las direcciones de memoria están representados por símbolos y permite la utilización de nombres simbólicos en lugar de números binarios o hexadecimales.



```
name "int02"
ORG 100H
lectura:
 mov ah,7
 int 21h

 mov tecla, al
 cmp al,13
 jz fin:
cmp tecla, 122 ;si tecla es mayor a 122 entonces ir a fin3 (tecla > 122)
ja fin3
cmp tecla,96 ;si tecla no es mayor a 96 ir a fin3 (tecla <= 96)
jng fin3
sub tecla, 32 ;si es 'a' hasta 'z' entonces restarle 32
fin3:
mov ah,2
mov dl,tecla
int 21h
jmp lectura
fin:
ret
tecla db 0

```

Figura 2: Código ensamblador que convierte una letra leída del teclado en mayúsculas

1.13 Tercera generación: Lenguajes de alto nivel

Aunque el lenguaje ensamblador reducía considerablemente la complejidad del proceso de programación, el proceso seguía siendo complejo y susceptible a errores.

En 1956, IBM lanzó el primer lenguaje de programación comercial de alto nivel, concebido para resolver problemas científicos y de ingeniería: FORTRAN. Esto marca el inicio de la tercera generación de lenguajes de programación, en la que los programas son expresados mediante secuencias de instrucciones muy parecidas al lenguaje natural.

Antes de ejecutar el programa, la computadora lo traduce a código de máquina directamente (lenguajes compiladores) o interpretando instrucción por instrucción (lenguajes intérpretes).

```
PROGRAM TRIVIAL
   INTEGER I
   I=2
   IF(I .GE. 2) CALL PRINTIT
   STOP
END
SUBROUTINE PRINTIT
   PRINT *,'Hola Mundo'
   RETURN
END
```

Figura 3: Código en lenguaje FORTRAN

1.1.4 Cuarta generación: 4gl (4th Generation Language)

Cada generación de lenguajes de programación provee un nivel más alto de abstracción y de ahí su distinción entre sí. Este nivel de abstracción determina el potencial de uso del lenguaje. Los lenguajes 4G permiten a los usuarios con poco conocimiento sobre computación y lenguajes de programación, desarrollar sus propios programas de aplicación sin la participación operativa de programadores.

Los lenguajes 4G son no procedimentales. Es decir, no pertenecen a la familia de lenguajes procedimentales, lo que provee a los usuarios especificar qué es lo que el programa debe realizar en términos del dominio del problema, en lugar de como debe hacerlo. En otras palabras, se especifica el resultado deseado, más que las acciones necesarias para obtener el resultado.

Algunos ejemplos de lenguajes de cuarta generación son: SQL, Visual FoxPro y PowerBuilder. Esta evolución de los lenguajes de programación y las herramientas utilizadas para convertir el código en programas, ha impactado en la forma en la que se desarrolla el software. El proceso de

desarrollo de software es la forma en la que se organizan las tareas necesarias para su construcción.

1.2. Evolución del proceso de desarrollo

En cada parte de la evolución de los lenguajes de programación también se ve una evolución en la forma de afrontar los problemas de desarrollo de software, en esta sección se hará un repaso sobre los más importes paradigmas de la programación.

1.2.1 Programación secuencial

Se da antes de la década de los 1960. Los primeros usuarios de la computadora fueron los mismos científicos e ingenieros que la habían diseñado y construido.

Esta etapa se caracteriza porque en el desarrollo de software se realizaba poco o ningún análisis (en el sentido metodológico del término) y el programa resultante era una sola secuencia de instrucciones desde el inicio hasta el fin del programa.

Los lenguajes empleados en esta etapa son el lenguaje de máquina, lenguaje ensamblador y algunos lenguajes de alto nivel.

1.2.2 Programación modular

Ocurre a inicio de los años 1960. Esta etapa se caracteriza por la implementación en los lenguajes y herramientas de programación de los conceptos de modularidad y programación descendente.

Los lenguajes de esta etapa son los de alto nivel como Pascal, C y Basic9

1.2.3 Programación estructurada

En mayo de 1966, Böhm y Jacopini demostraron que un programa propio puede ser escrito utilizando solamente tres tipos de estructuras: secuenciales, selectivas y repetitivas.

Esta etapa se caracteriza por la simplificación de la sintaxis de los lenguajes de programación para la utilización únicamente de las 3 estructuras de control fundamentales, a las que posteriormente se sumaron las 3 estructuras extendidas.

1.2.4 Diseño estructurado

Las primeras técnicas estructuradas se centraron principalmente en los programas. El énfasis se ponía sobre la estructura de los programas, es decir; la claridad y calidad del código. Sin embargo, en la medida en que se desarrollaba software más grande y en diferentes dominios, como software empresarial, se fueron desarrollando criterios estándar de calidad de diseño.

A mediados de los años 1970, la filosofía estructurada abarcó también a la fase de diseño. El concepto de estandarización fue aplicado al proceso de solución de problemas como una forma de introducir organización y disciplina al diseño de programas. Conceptos como modularidad y diseño descendente fueron estandarizándose, definiendo estándares para las interfaces entre los módulos y métricas de calidad, lo que permitió que los desarrolladores fueran creando librerías de programas para reutilizarlas en otros proyectos de desarrollo.

1.2.5 Análisis estructurado

En la medida en que la complejidad y el tamaño de las aplicaciones crecía, definir y observar los estándares de desarrollo, y la reutilización de código no eran suficientes para resolver el problema de la complejidad del desarrollo.

Los desarrolladores observaron que mucho de los problemas del desarrollo estaban en la definición de los requerimientos. Identificar y enunciar correctamente los requerimientos era una labor fundamental.

A finales de los años 1970, se crean técnicas para la especificación de requerimientos y el análisis de los mismos, lo que se conocen como técnicas estructuradas. Un ejemplo de estas técnicas son los Diagramas de Flujo de Datos (DFD), que son una representación gráfica del flujo de los datos a través de un sistema de información.

Diagrama de contexto

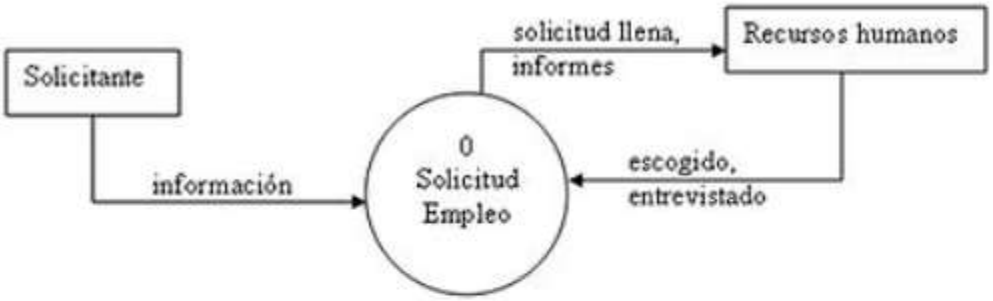


Diagrama cero

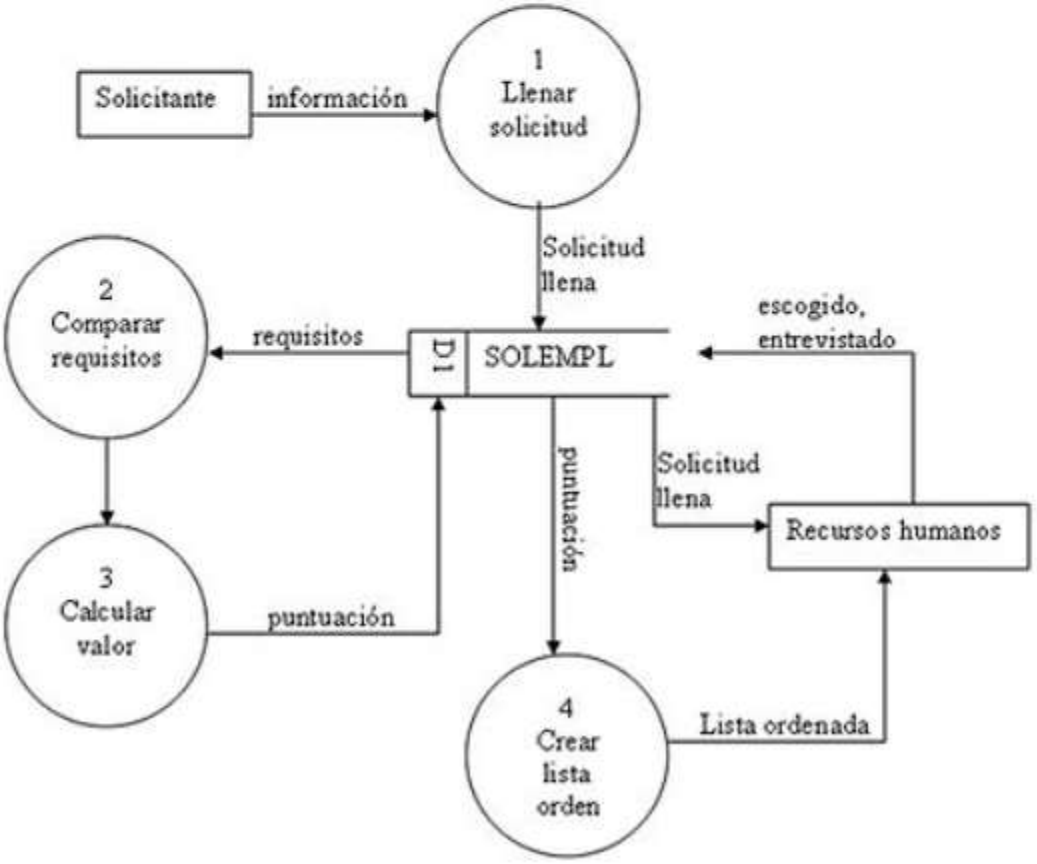


Figura 4: Diagramas de Flujo de Datos para el proceso de selección de personal.

1.2.6 Programación orientada a objetos

Ocurre a inicios de los años 1980. En la orientación a objetos el bloque de construcción ya no es el módulo, procedimiento o función, sino el objeto.

No fue hasta finales de la década de los 1990 que los lenguajes orientados a objetos comienzan a tomar auge y dicho paradigma de programación se convierte en el paradigma dominante, hasta hoy en día.

1.2.7 Análisis y diseño orientado a objetos

Al igual como sucedió con la programación estructurada en su momento, la POO facilita el desarrollo, pero no es suficiente. Para desarrollar software de calidad es necesaria la creación de métodos y técnicas de análisis y diseño orientado a objetos.

A inicios de los años 1990 se comienzan a construir nuevas técnicas, métodos y herramientas para resolver este problema. En esta etapa aparecen una serie de métodos y herramientas tan diversas que se hace necesaria una estandarización que culmina con la definición y publicación del Lenguaje Unificado de Modelado (UML).

1.2.8 Desarrollo asistido por computadora

En los últimos años se ha apostado por la creación de técnicas que permitan incrementar la productividad y el control de calidad en cualquier proceso de elaboración de software. La tecnología CASE (Computer Aided Software Engineering) brinda herramientas para todas las etapas del proceso de desarrollo, apostando por la automatización del proceso de forma análoga las tecnologías CAD/CAM.

No obstante, hay muy pocos resultados en el intento de automatizar el proceso de desarrollo en toda su dimensión. Enfoques como el Human-Centric BPM (Modelo de Procesos de Negocio Centrado en el Humano) intentan automatizar algunas de las actividades del proceso, delegando al usuario la definición y configuración del comportamiento del sistema.

1.3. Metodologías de desarrollo

En la medida en que se iban organizando las tareas del proceso de desarrollo de software, y se iba poniendo énfasis en algunas de estas tareas, el proceso iba siendo formalizado en lo que se conoce como metodología de desarrollo. La metodología de desarrollo es la forma en cómo las tareas del proceso de desarrollo, las funciones de los miembros del equipo y las responsabilidades de los involucrados son organizadas y programadas en el tiempo.

En 1970, Winston Royce creó la metodología de desarrollo en cascada, que supone la organización de las principales actividades del proceso de forma escalonada y secuencial.

Desde entonces, se han creado metodologías de desarrollo que han respondido a la necesidad de contar con un marco de trabajo para saber qué hacer y cuándo hacerlo en el proceso de desarrollo. Con el auge del paradigma de programación orientado a objetos, aparecen nuevas metodologías, hasta llegar a nuestros tiempos donde el desarrollo ágil es el nuevo paradigma de organización del proceso de desarrollo.

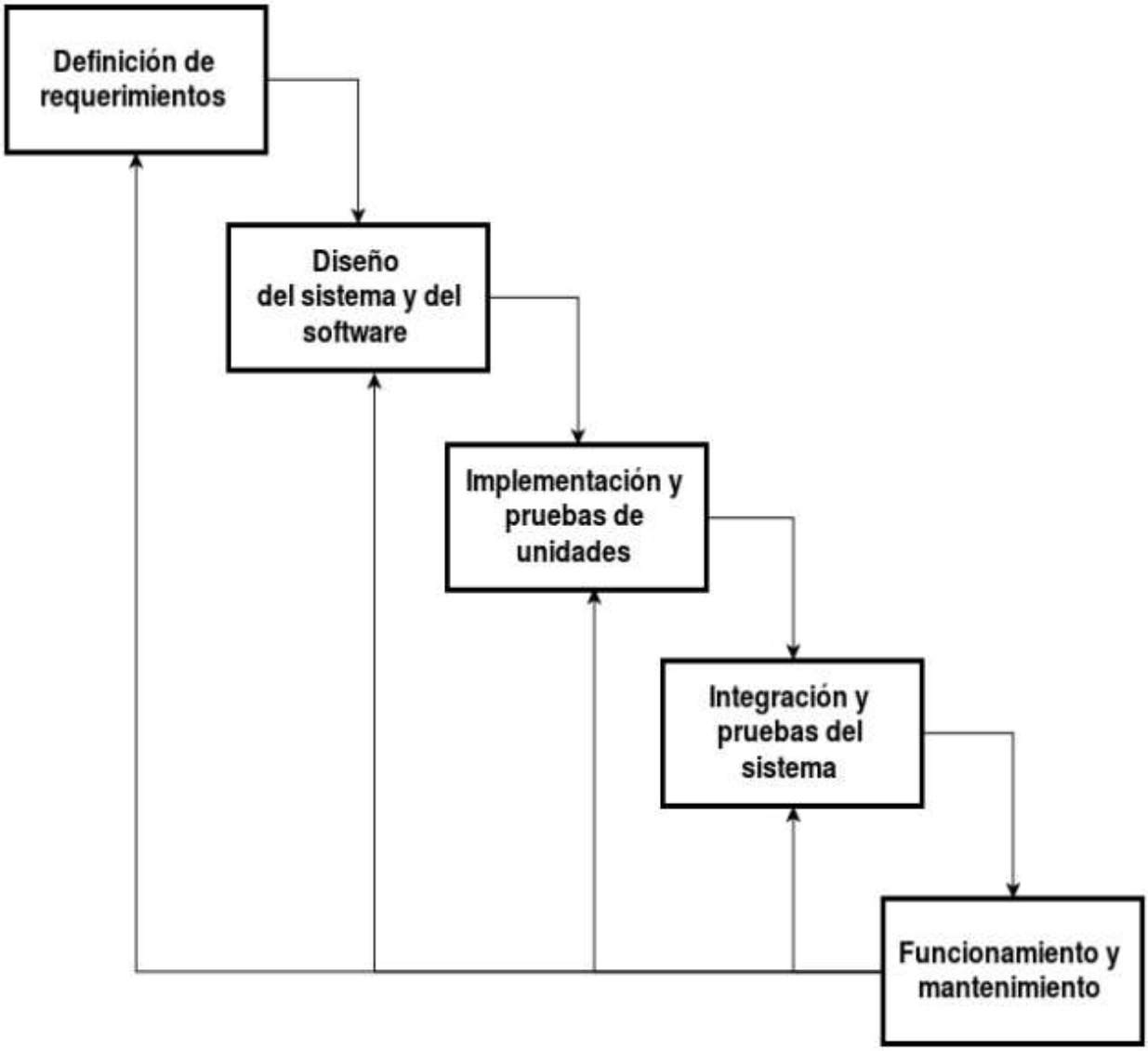


Figura 5: Modelo en cascada clásico

Algunas de las metodologías de desarrollo más importantes son:

Paradigmas de desarrollo (modelo de desarrollo) tradicional:

- Cascada.
- Incremental.
- Evolutivos.

- Prototipado.
- Espiral.

Paradigmas de desarrollo orientados a objetos:

- RUP (Rational Unified Process).
- OUP (Open Unified Process).

Paradigmas de desarrollo ágiles:

- XP (Extreme Programming)
- Scrum
- AUP (Agile Unified Process)

2.0 Evolución de las herramientas para el desarrollo de software

Una herramienta es un artefacto, mecanismo o técnica para la realización de una tarea o mejorar la calidad o eficiencia en su realización.

Las herramientas de desarrollo permiten reducir el tiempo o elevar la calidad de los resultados de las tareas del proceso de desarrollo. Estas mejoras tendrán un impacto directo en la productividad del equipo.

Se clasificará las herramientas de acuerdo a su funcionalidad, es decir; por el uso que se les da.

2.1. Herramientas para la gestión de proyectos

Permiten planificar las actividades y los recursos que se utilizarán en un proyecto. También permite el seguimiento de la ejecución del proyecto.

2.2. Herramientas para gestión de requerimientos

Permiten documentar y controlar (validar, gestionar cambios e implementación de la solución) los requerimientos durante todo el ciclo de desarrollo.

2.3. Herramientas para la programación

En esta categoría se encuentran los editores, compiladores, enlazadores, depuradores, normalmente integrados en un IDE (Interface Development Enviroment) o Entorno de Desarrollo Integrado.

En esta categoría se encuentran también los Sistemas de Control de Versiones

2.4. Herramientas de análisis y diseño

En esta categoría se encuentran herramientas para el diseño de software (normalmente a través de UML), diseño de bases de datos, diseño de interfaces de usuario (Mockups), diseño de procesos de negocio (BPMN).

2.5. Herramientas para el aseguramiento de la calidad

En esta categoría se encuentran las herramientas para la gestión de pruebas y para el soporte, como los Sistemas de Gestión de Incidencias.

2.6. Herramientas de ingeniería inversa

Las herramientas de ingeniería inversa son útiles cuando existe poca o ninguna documentación del software, cuando los cambios no han sido documentados o cuando el tamaño es muy grande y se hace necesario contar con modelos del software implementado que permitan reducir los riesgos de los cambios.

3. La productividad

La productividad es la relación entre la producción obtenida por un sistema de producción o servicios y los recursos utilizados para obtenerla.

Se puede definir la productividad como el uso eficiente de recursos -trabajo, capital, tierra, materiales, energía, información- en la producción de cualquier bien o servicio.

Por tanto, el concepto básico sigue siendo la relación entre la cantidad y calidad de bienes o servicios producidos y la cantidad de recursos utilizados para producirlos.

La productividad de un proceso de desarrollo usualmente es una medida de la velocidad con la que se desarrolla el software. Típicamente es medida en términos de la cantidad de funciones (funcionalidad requerida), líneas de código, casos de uso, clases u operaciones que se desarrollan por unidad de tiempo. Para dar respuesta a estos requerimientos, será necesario un esfuerzo por parte del equipo de desarrollo, ese esfuerzo es medido en términos de horas por persona.

La medición de la productividad es útil para realizar las estimaciones iniciales de un proyecto de desarrollo. El tamaño del software y la productividad del equipo sirve para determinar el tiempo necesario para realizar un proyecto de desarrollo.

3.1. Representación de la productividad

La productividad se suele representar con la fórmula:

productividad=producto/insumos

Es el cociente entre la cantidad de bienes producidos y la cantidad de insumos empleados para producirlos. En el ámbito del desarrollo de software, esta relación puede describirse como:

productividad= atributos del software / esfuerzo total de desarrollo

Una productividad mayor significa la obtención de más con la misma cantidad de recursos o el logro de una mayor producción en volumen y calidad con el mismo insumo.

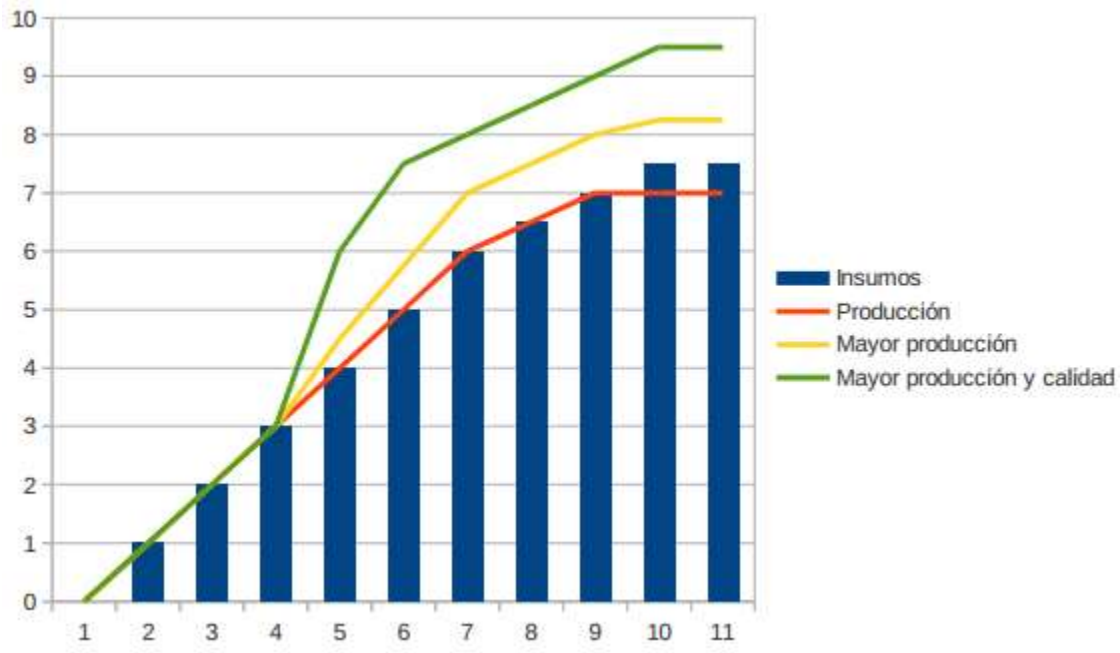


Figura 6: Comparación de niveles de resultados obtenidos contra insumos invertidos.

La productividad también puede definirse como la relación entre los resultados y el tiempo que lleva conseguirlos. El tiempo es a menudo un buen denominador, puesto que es una medida

universal y está fuera del control humano. Cuanto menor tiempo lleve lograr el resultado deseado, más productivo es el sistema de producción.

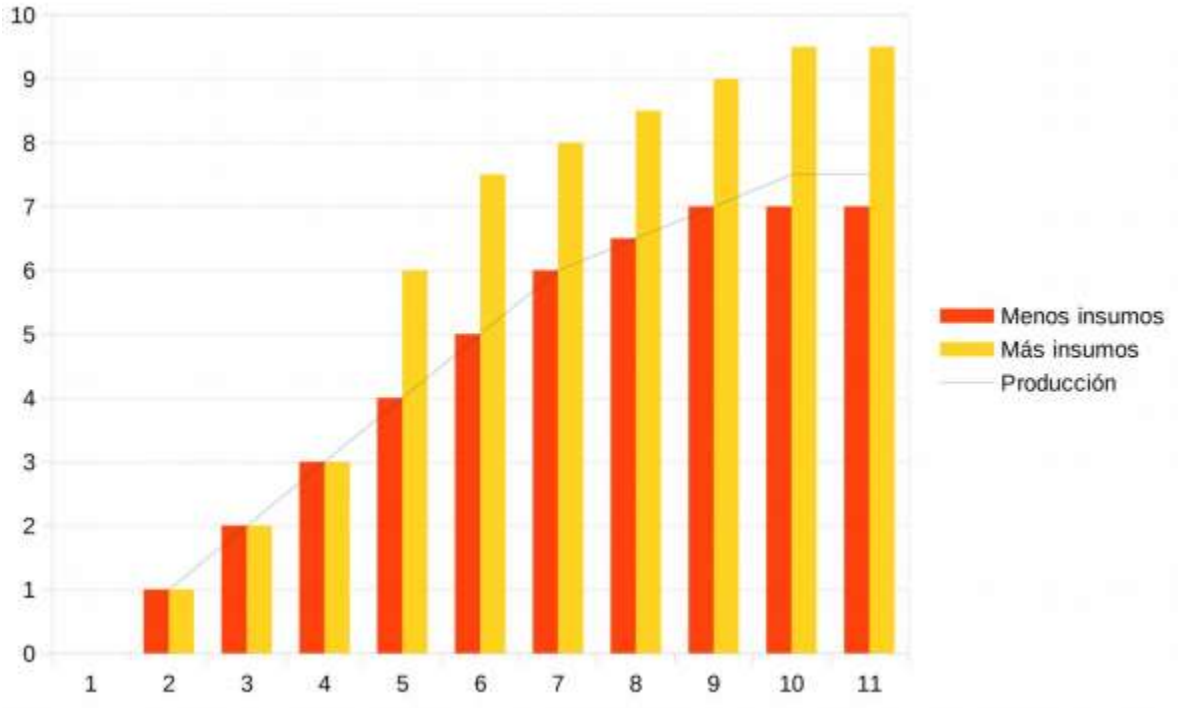


Figura 7: La productividad vista como la reducción de los insumos invertidos.

3.2. Importancia de la productividad

La productividad ha sido objeto de estudio durante largo tiempo, por la importancia que reviste en el orden económico y social de las naciones. Ser productivo es una tema clave para las estrategias de desarrollo en las agendas de los países. No obstante, también es importante en el ámbito del comportamiento empresarial. La productividad resulta un factor importante en el desempeño de las empresas para maximizar sus ganancias.

3.3. Medición de la productividad

En cada proceso productivo es necesario la estimación del costo y esfuerzo, lo mismo cabe decir para el proceso de desarrollo del software, el cual no es la excepción.

¿Qué es la estimación de costos?

Consiste en predecir los recursos (monetarios, temporales, humanos, materiales, etc.) necesarios para llevar a cabo el proceso de desarrollo del software.

Debemos resolver una serie de cuestiones fundamentales durante la estimación:

- ¿Cuánto esfuerzo se requiere para completar una actividad?
- ¿Cuánto tiempo es necesario para llevar a cabo una actividad?
- ¿Cuál es el costo total de una actividad?

3.3.1 Costos a considerar en un proyecto

- Costo del hardware y software utilizados.
- Costo de dietas, viajes y aprendizaje.
- Costo del esfuerzo (factor dominante casi siempre), este incluye:
    - Salarios del personal involucrado en el proyecto
    - Costos de seguros y seguridad social.
- Costos indirectos aplicados al personal del proyecto:
    - Costos de mantenimiento, alquiler, luz, acondicionamiento del ambiente, etc.
    - Costos administrativos, redes y de comunicaciones.
    - Costos sociales colectivos.
    - Costos de recursos compartidos (p. ej:. librería, personal, etc.)

3.4. Técnicas de estimación

En las fases más tempranas de un proyecto (iniciación o planificación), será necesario estimar el esfuerzo o los costos del proyecto. No existe una forma simple de obtener estimaciones exactas del esfuerzo requerido para desarrollar un sistema software. Típicamente, las estimaciones iniciales parten de la definición imprecisa de requerimientos por parte del usuario.

Los entornos de producción, la tecnología a utilizar y los procedimientos internos de la organización para la que se desarrolla el software, pueden también ser un factor determinante del esfuerzo que el equipo de desarrollo deba realizar. El uso de una tecnología en el que el equipo de desarrollo no esté familiarizado o no sea experto, tendrá un impacto en los niveles de productividad, en la medida en que el tiempo de desarrollo de una función del software tome más tiempo.

La estimación se realiza muchas veces valiéndose de la experiencia y de la opinión de expertos como única guía. Sin embargo, el juicio de los expertos puede dejar fuera las características particulares del equipo de desarrollo. Desconocer las competencias y habilidades del equipo puede dejar como resultado malas estimaciones.

Otra alternativa es la descomposición a partir de una base histórica de proyectos, en este caso habrá que considerar que:

- Los datos deben ser precisos.
- Los datos deben obtenerse de tantos proyectos como sea posible.
- Debe existir consistencia entre las medidas utilizadas (p. ej. líneas de código referidas a un mismo lenguaje).
- Las aplicaciones que se contemplan deben ser similares a la que se pretende estimar.

Otra técnica usada frecuentemente es la aproximación por similitud, se parte del supuesto que un proyecto similar a otro en tamaño, complejidad y tipo de funciones, probablemente dure y cueste aproximadamente lo mismo.

Los modelos empíricos de estimación son también una técnica aplicable cuando no existe una base histórica apropiada para hacer la estimación. Estos modelos, usualmente basados en la experiencia, describen las relaciones entre las características de los proyectos y el esfuerzo.

Ejemplos de estas técnicas son:

- **Modelado algorítmico de costos**: se basa en el tamaño del software.
- **Juicio de Expertos**: usan su experiencia para predecir costos de software.
- **Estimación por analogía**: comparación con proyectos similares.
- **Ley de Parkinson:** en función de los recursos disponibles.
- **Pricing to win**: en función de lo que el cliente está dispuesto a pagar.

Las métricas de tamaño nos sirven para estimar el tamaño del software, que en la función de productividad está enunciado como los atributos del software. La productividad puede ser conocida a través de la técnica de descomposición a partir de una base histórica. Por ejemplo, si sabemos cuál fue la productividad del equipo de desarrollo en otros proyectos similares, podemos estimar el esfuerzo requerido para el desarrollo, si tenemos una estimación del tamaño del software.

4. La calidad del software y la productividad

Normalmente asociamos el concepto de calidad a la fabricación de un producto o a la adquisición de un servicio. En los procesos de fabricación, solemos asociar la calidad con la similitud entre el producto terminado y su diseño.

Sin embargo, la calidad del software es un concepto complejo, en la medida en que se encuentran algunas características particulares, tales como:

1. Se desarrolla, no se fabrica en el sentido clásico del término. Todo el costo de su producción se centra en el diseño de la primera copia. De ser necesario, realizar más copias resulta una tarea trivial.
2. Se trata de un producto lógico, sin existencia física. El verdadero producto del software es el diseño de una serie de instrucciones para el computador.
3. No se degrada con el uso. La naturaleza lógica del software permite que permanezca inalterable por muy intensa que sea su utilización.
4. La complejidad del software, la ausencia de controles adecuados y el comportamiento del mercado actual provoca productos que muchas veces se entregan conscientemente con defectos, incluso públicamente declarados.
5. Un porcentaje muy grande de la producción se hace aún a la medida.
6. Es muy flexible. Se puede cambiar con relativa facilidad.

La definición más aceptada de calidad del software es la declarada en el estándar IEEE STD.610-1991.

La calidad del software es el grado con el que un sistema, componente o proceso cumple los requerimientos especificados y las necesidades o expectativas del cliente o usuario.

Esta definición centra la calidad en el producto del proceso de desarrollo: el software. Sin embargo, al igual que para otros productos, la calidad puede medirse desde la perspectiva del proceso. En la medida en que el proceso de desarrollo cuente con medidas de aseguramiento de la calidad, tendremos un grado de certeza de su calidad.

Desde esta perspectiva, la calidad del software puede medirse desde dos enfoques:

1. La calidad del proceso.
2. La calidad del producto, que también puede dividirse en:
    a. Calidad interna.
    b. Calidad externa.

c. Calidad de uso.

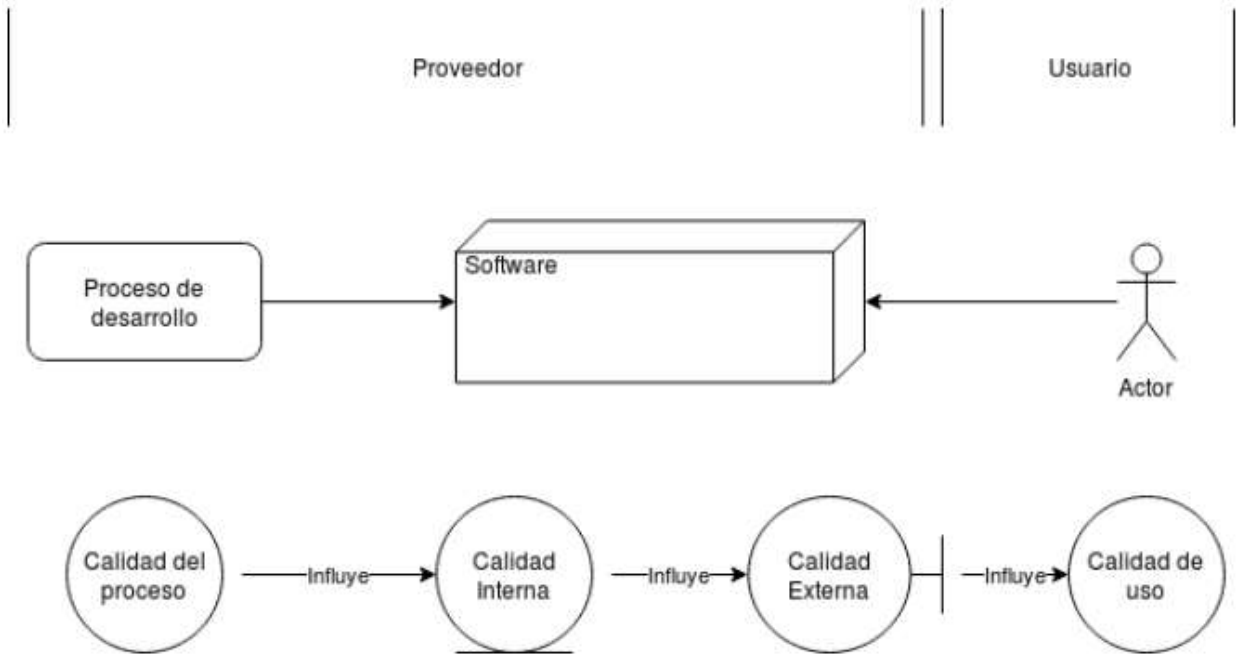


Figura 7: Elementos de la calidad del software

Existen varios modelos para la evaluación y aseguramiento de la calidad del proceso de desarrollo, como CMMi, ISO-15504, SPICE., TSP, PSP, PMI, etc. De igual manera hay algunos estándares para la medición de la calidad del producto, siendo el más aceptado en la actualidad la norma ISO/IEC 25000, llamada también SQuaRE.

En esta asignatura nos centraremos en la calidad del producto, por lo que utilizaremos el estándar anterior a SQuaRE, el ISO/IEC 9126 que permite separar estas características.

Es evidente la relación entre la calidad del producto y la calidad del proceso. Sin embargo, medir los atributos de la calidad del software (el producto) resulta una tarea compleja, en la medida que resulta difícil explicar cómo influyen las características del proceso en estos atributos. Por ejemplo, un diseño que podemos considerar poco robusto, o la creatividad del equipo para dar

respuesta a los requerimientos, volverán difícil anticipar el impacto de un cambio en el proceso de desarrollo.

Aunque en esta asignatura nos centraremos en los parámetros de calidad del producto, es importante señalar que la calidad del proceso tiene una influencia significativa en la calidad del software. La gestión y mejora de la calidad del proceso debería propiciar una reducción de los defectos en el software entregado.

El proceso de control de calidad implica comprobar que el proceso del software y el software resultante concuerdan con los estándares adoptados.

4.1. Calidad del producto

La familia de normas ISO/IEC 9126 especifica una serie de modelos y métricas para el aseguramiento de la calidad del software desde el punto de vista del producto.

ISO/IEC 9126 – Tecnologías de la Información – Calidad de los productos de software

- ISO/IEC 9126-1: Modelo de calidad
- ISO/IEC 9126-2: Métricas externas
- ISO/IEC 9126-3: Métricas internas
- ISO/IEC 9126-4: Métricas de Calidad de uso

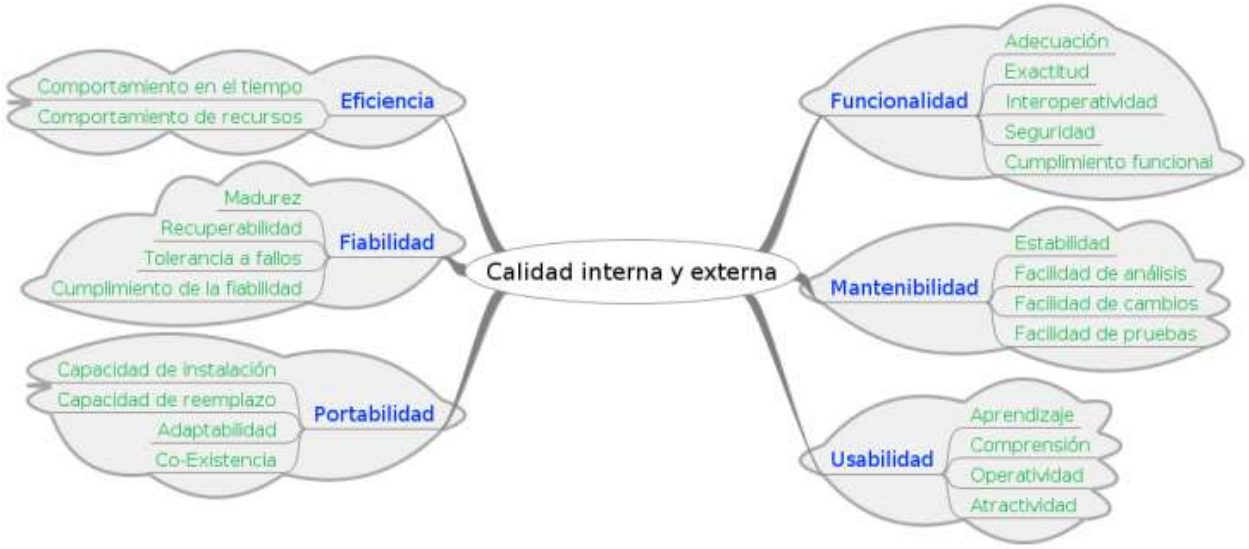


Figura 8: Modelo de calidad general de la norma ISO/IEC 9126

1. Funcionalidad: Un conjunto de atributos que se relacionan con la existencia de un conjunto de funciones y sus propiedades específicas. Las funciones son aquellas que satisfacen las necesidades implícitas o explícitas.
    a. Adecuación: Atributos del software relacionados con la presencia y aptitud de un conjunto de funciones para tareas especificadas.
    b. Exactitud: Atributos del software relacionados con la disposición de resultados o efectos correctos o acordados.
    c. Interoperabilidad: Atributos del software que se relacionan con su habilidad para la interacción con sistemas especificados.
    d. Seguridad: Atributos del software relacionados con su habilidad para prevenir acceso no autorizado ya sea accidental o deliberado, a programas y datos.
    e. Cumplimiento funcional.
2. Fiabilidad: Un conjunto de atributos relacionados con la capacidad del software de mantener su nivel de prestación bajo condiciones establecidas durante un período establecido.

   a. Madurez: Atributos del software que se relacionan con la frecuencia de falla por fallas en el software.
   b. Recuperabilidad: Atributos del software que se relacionan con la capacidad para restablecer su nivel de desempeño y recuperar los datos directamente afectos en caso de falla y en el tiempo y esfuerzo relacionado para ello.
   c. Tolerancia a fallos: Atributos del software que se relacionan con su habilidad para mantener un nivel especificado de desempeño en casos de fallas de software o de una infracción a su interfaz especificada.
   d. Cumplimiento de Fiabilidad: La capacidad del producto software para adherirse a normas, convenciones o legislación relacionadas con la fiabilidad.

3. Usabilidad: Un conjunto de atributos relacionados con el esfuerzo necesario para su uso, y en la valoración individual de tal uso, por un establecido o implicado conjunto de usuarios.
   a. Aprendizaje: Atributos del software que se relacionan al esfuerzo de los usuarios para reconocer el concepto lógico y sus aplicaciones.
   b. Comprensión: Atributos del software que se relacionan al esfuerzo de los usuarios para reconocer el concepto lógico y sus aplicaciones.
   c. Operatividad: Atributos del software que se relacionan con el esfuerzo del usuario para la operación y control del software.
   d. Atractividad
4. Eficiencia: Conjunto de atributos relacionados con la relación entre el nivel de desempeño del software y la cantidad de recursos necesitados bajo condiciones establecidas.
   a. Comportamiento en el tiempo: Atributos del software que se relacionan con los tiempos de respuesta y procesamiento y en las tasas de rendimientos en desempeñar su función.
   b. Comportamiento de recursos: Usar las cantidades y tipos de recursos adecuados cuando el software lleva a cabo su función bajo condiciones determinadas.

5. Mantenibilidad: Conjunto de atributos relacionados con la facilidad de extender, modificar o corregir errores en un sistema software.
    a. Estabilidad: Atributos del software relacionados con el riesgo de efectos inesperados por modificaciones.
    b. Facilidad de análisis: Atributos del software relacionados con el esfuerzo necesario para el diagnóstico de deficiencias o causas de fallos, o identificaciones de partes a modificar.
    c. Facilidad de cambio: Atributos del software relacionados con el esfuerzo necesario para la modificación, corrección de falla, o cambio de ambiente.
    d. Facilidad de pruebas: Atributos del software relacionados con el esfuerzo necesario para validar el software modificado.
6. Portabilidad: Conjunto de atributos relacionados con la capacidad de un sistema software para ser transferido desde una plataforma a otra.
    a. Capacidad de instalación: Atributos del software relacionados con el esfuerzo necesario para instalar el software en un ambiente especificado.
    b. Capacidad de reemplazamiento: Atributos del software relacionados con la oportunidad y esfuerzo de usar el software en lugar de otro software especificado en el ambiente de dicho software especificado.
    c. Adaptabilidad: Atributos del software relacionados con la oportunidad para su adaptación a diferentes ambientes especificados sin aplicar otras acciones o medios que los proporcionados para este propósito por el software considerado.
    d. Co-Existencia: Coexistir con otro software independiente, en un entorno común, compartiendo recursos comunes.

El desarrollo de software es un proceso más creativo que mecánico. La calidad del producto también se ve afectada por factores externos, como la novedad de una aplicación o la presión comercial para sacar un producto rápidamente.

La calidad interna es el elemento de calidad más cualitativo, por ende, permite establecer procesos de control de calidad (aseguramiento de la calidad) más expeditos. De hecho, los procesos de control de calidad interna permiten identificar fallos en fases tempranas de desarrollo, y, por ende; reducción de tiempo y esfuerzo del equipo de desarrollo, que representa mayor productividad y menos costos.

Normalmente asociamos los controles de calidad y las normas de calidad como especificaciones abstractas, especificadas con el único objetivo de vender la idea de sofisticación en nuestra concepción de ingeniería de software, pero que en la práctica no tendrá ningún impacto en el producto, ni mucho menos en el proceso de desarrollo. No obstante, en el proceso de desarrollo de software, estos controles tendrán un impacto directo en varios atributos de calidad del producto.

Uno de los componentes más importantes de los controles de calidad interna son los estándares de desarrollo. En todo proyecto de desarrollo es posible especificar patrones, normas, estándares y convenciones. Estos elementos están determinados por el diseño arquitectónico del software, la tecnología requerida para su desarrollo, convenciones generalmente aceptadas, buenas prácticas y estándares específicos para el proyecto o la organización.

En esta unidad estudiaremos algunos de los estándares más comúnmente aceptados que son utilizados para la garantía de la calidad, tanto del proceso como del producto.

4.2. Estándares

Un estándar es un modelo o patrón que tienen forma de reglas que se deben seguir para realizar un proceso. Podemos definir un estándar de desarrollo como un conjunto de reglas a aplicar en el proceso de desarrollo.

4.2.1 Estándares de diseño

Como se ha mencionado anteriormente, los estándares de diseño normalmente están determinados por los requerimientos tecnológicos y arquitectónicos del software. Por ejemplo, no sería lógico adoptar un estándar de diseño de base de datos en particular, si la aplicación que se desea desarrollar no almacenará datos en una base de datos.

El atributo que se ve impactado por este tipo de estándar es la mantenibilidad.

4.2.2 Estándares de diseño de interfaces de usuario

Normalmente este tipo de estándar se define en conjunto con el usuario. Se acuerdan algunas normas básicas para el diseño y programación de las interfaces de usuario como: disposición (layout), tipo de controles a utilizar, comportamiento general, etc.

Los atributos de software que más se ven afectados por este tipo de estándares son la funcionalidad, la usabilidad y la mantenibilidad.

4.2.3 Estándares de diseño de base de datos

El diseño de la base de datos es una de las actividades más críticas del proceso de desarrollo. Al igual que casi todos los tipos de estándar, se ve impactado por la tecnología que se requiera para la construcción del software y la adopción de algunas técnicas de diseño e implementación de base de datos.

Por ejemplo, en algunos gestores de base de datos como Oracle, el nombre de los objetos de la base de datos no puede sobrepasar de 30 caracteres. Otro ejemplo es la adopción de un patrón de diseño o una técnica de persistencia, podemos definir como estándar que todas las entidades

deberán tener como llave primaria una llave ficticia y no se deben modelar entidades con llaves compuestas.

Los atributos que más se ven impactados por este tipo de estándares son la mantenibilidad y la fiabilidad.

4.2.4 Estándares de programación

Los estándares de programación o de codificación, se refieren a los estándares que se utilizarán para la escritura del código fuente de nuestro software.

Al igual que la mayoría de los estándares, se ve impactado por la tecnología requerida. Por ejemplo, si nuestro sistema será construido en lenguaje Java, existen una serie de convenciones de codificación para ese lenguaje, emitidas y validadas por la Java Community Proccess.

En cambio, si el lenguaje de programación es PHP, un lenguaje mucho más permisivo. Los estándares o convenciones no pasan de ser recomendaciones de "buenas prácticas", por lo que en un proyecto de desarrollo, se deberían definir nuestros propios estándares.

Generalmente, los estándares de programación incluyen:

1. Nombrado de objetos (variables globales, variables locales, constantes, funciones, procedimientos, módulos, rutinas, paquetes, clases, etc.)
2. Indentación
3. Documentación interna (de archivo, de clases, de función, de método, de procedimiento, de módulo)
4. Estilo de programación (estructurada, no estructurada)

En este estándar normalmente se incluye también los estándares de codificación de procedimientos almacenados, triggers y funciones de la base de datos.

Los atributos que más se ven impactados por este tipo de estándar son la mantenibilidad y la eficiencia.

4.2.5 Estándares de documentación externa

Los estándares de documentación externa se refieren principalmente a los manuales y a cualquier otro documento que acompañe el software. Los manuales de usuario y técnico son los ejemplos más evidentes. Sin embargo, esto puede incluir manual de implementación, documentación del proceso de desarrollo, entre otros.

Los atributos que más se ven impactados por este tipo de estándar son la usabilidad y la mantenibilidad.

En términos generales, el estándar SQuaRE es una integración de los estándares ISO/IEC 9126 (Calidad del software como producto) e ISO/IEC 14158 (Evaluación del software como producto), extendiendo las definiciones de estos estándares e incorporando nuevos elementos como guías de evaluación, tanto del producto, como del proceso.

El estándar SQuaRE, agrega nuevos atributos, descompone algunos existentes en el estándar 9126 y los reorganiza en un total de 8 atributos. En general, el estándar SQuaRE es un estándar más detallado y por ende más fácil de aplicar. Sin embargo, dado que su definición incluye procedimientos de planificación, especificación y evaluación.

Referencias

Luis Joyanes Aguilar, Ignacio Zahonero Martinez. *Programación en C: metodología, algoritmos y estructuras de datos*. 2.a ed. McGraw-Hill, 2005.

Luis Joyanes Aguilar, Ignacio Zahonero Martinez. *Programacion En C, C++, Java Y Uml*. 3ª ed. McGRAW HILL, 2014.

José Luis Pérez Valero. «Técnicas de Programación y Desarrollo». Universidad Autónoma de Coahuila, s. f.