CURSO

TÉCNICAS DE PROGRAMACIÓN PERSONAL CON CALIDAD

Introducción

En la inmensa mayoría de los países los sistemas sociales, políticos, económicos y hasta culturales dependen en gran medida de complejos sistemas basados en computadoras, y evidentemente en software. La dependencia no es exclusivamente hacia su interior, sino que las relaciones entre ellos son también, en muchos casos, regidas por tecnología. El desarrollo de software de complejidad y tamaño cada vez mayores ha sido inevitable. Aunado a esto, la necesidad de que los productos de software sean seguros y con calidad, es un reclamo constante.

Roger Pressman, en su libro de ingeniería de software cuarta edición, escribía: "*El software es casi ineludible en un mundo moderno. A medida que nos adentremos en el siglo XXI, será el que nos conduzca a nuevos avances en todo, desde la educación elemental a la ingeniería genética"* cualquier cometario a esto sale sobrando.

El uso y el desarrollo de software se ha vuelto crítico y en muchos aspectos vitales; quizá sin que lo percibamos dependen de él directa e indirectamente no sólo vidas, sino la calidad de ésta. Por natural necesidad hemos transitado de la no tan simple programación de computadoras a lo que ahora se conoce como ingeniería de software. Una diferencia entre estos dos puntos es que el programador solitario de antaño ha sido sustituido por un equipo de especialistas de software. Esta diferencia, aunque básica, no es la única. El término ingeniería da una profundidad que es necesario entender completamente.

El desarrollo de software a final de cuentas tiene como objetivo la entrega de un producto, y como tal, este producto tiene un usuario. No era posible que pasara mucho tiempo sin que la comunidad que desarrolla software y los mismos usuarios se percataran de que este hecho exigía darle al desarrollo de software otras dimensiones.

Si bien en sus inicios el desarrollo de software era artesanal, y en alguna medida sigue siéndolo, era necesario incorporarle al menos por analogía con otras disciplinas e industrias: procesos, métodos técnicos y de gestión, así como herramientas. Evidentemente a la par, o quizá de forma más adecuada mediante la creación de una disciplina de estudio, de investigación y de aplicación, que fundamentalmente tomara el desarrollo de software como "*leit motiv*" La Ingeniería de Software es esta disciplina y es una disciplina joven.

El principal objetivo es introducir disciplina en el proceso de desarrollo de software del individuo. Mediante TÉCNICAS DE CALIDAD PERSONALES DE PROGRAMACIÓN se describe el desarrollo de programas pequeños desde la

asignación del problema hasta las pruebas de unidad dejando fuera todo lo relacionado a manejo de equipos de proyecto y estructura de la organización

Agradecimiento y Reconocimiento

Después de una ardua tarea de investigación se ha logrado la creación de una obra vasta en conocimiento en el desarrollo de las Tecnologías de la Información y Comunicación.

La presente obra no hubiera sido posible sin la valiosa aportación de destacados autores y especialistas en la materia. Es por ello que a manera de reconocimiento queremos agradecer su participación:

TÉCNICAS DE PROGRAMACIÓN CON CALIDAD

M. en C. Carlos Mojica Ruiz
Universidad Autónoma de Yucatán

Dr. Leonardo Chapela Castañares
Prodigia, S.A. de C.V.

Intención Educativa

Desarrollar software no es solo producir líneas de código (hacer programas) que al final den un resultado esperado. Detrás de la programación existen otros aspectos importantes, por ejemplo: calidad, proceso, métricas, planeación, productividad entre otros.

Con este curso sabrás obtener datos personales relativos a tu proceso de desarrollar programas y con ellos podrás analizar, planear y mejorar la calidad de tus programas. Esto ayudará a incrementar tu desempeño como programador mediante el entendimiento y mejoramiento de tu proceso personal de programación.

La disciplina adquirida y la aplicación de la técnica aprendida durante el curso, te harán más competitivo en el mercado laboral.

Objetivos Generales

Al terminar el curso, el alumno:

- Tendrá una disciplina y una técnica para aplicar y mejorar su proceso personal de desarrollo de programas y producirlos con mayor calidad.

Objetivos Específicos

Al terminar el curso, el alumno será capaz de:

- Definir e identificar los conceptos de la Ingeniería de Software y explicar su importancia.
- Conocer y describir cuales son las principales aplicaciones del software.
- Analizar todas las etapas del proceso de Ingeniería de Software.
- Aplicar las diferentes métricas para la productividad y la calidad del software.
- Aplicar las diferentes metodologías diseño e implementación del software.
- Conocer y aplicar las diferentes estrategias para realizar las pruebas del software.
- Discutir la planeación de proyectos y el proceso de planeación.
- Aplicar las diferentes metodologías de análisis de requisitos del sistema y del software .
- Establecer planes para el desarrollo del proyecto y su manejo, que involucran: definición de qué se va a hacer, restricciones, objetivos, estimados de tamaño y esfuerzo requerido, recursos necesarios, calendario, identificación y manejo de riesgos.
- Comparar el diseño de interfases mostrando el uso de las diferentes herramientas de desarrollo.
- Analizar en líneas generales de la construcción de un programa la planificación, desarrollo.

- Determinar el comportamiento controlado del individuo en el desarrollo de la ingeniería con resultado de alta productividad y calidad y resultados predecibles.

- Diseñar productos de software resolviendo aspectos básicos de desarrollo.

- Preparar y medir el trabajo de desarrollo de software mediante métricas establecidas.

- Establecer el proceso y asignación de pruebas a los sistemas que sean capaces de mejorar el desarrollo de sistemas.

Metodología

La práctica prevalece en este curso. A continuación se describe el método generalmente empleado.

Comprobación de Lectura

La técnica de comprobación de lectura tiene como finalidad fomentar en el alumno la habilidad de leer, analizar y comprender. La comprensión se comprobará al final de cada lección, ya que se presenta una evaluación por medio de preguntas muy puntuales acerca de la lectura.

Los materiales que se utilizarán en este método son una recopilación de diferentes autores de un tema, para homogenizar los conceptos e ideas referentes al tema.

La técnica de comprobación de lectura es una de las más empleadas en los procesos de enseñanza-aprendizaje y tiene como finalidad conformar conceptos e ideas propias al alumno, por lo que no pretende que se memoricen los temas tratados.

Fuentes de Información

- **Humphrey W**., "*A Discipline for Software Engineering*" READING, MASS. : ADDISON WESLEY, SEI Series in Software Engineering, 1995.

- **Humphrey W**., "*Introduction to the Personal Software Process*" READING, MASS. : ADDISON WESLEY, SEI Series in Software Engineering, 1997.
  Traducción al Español: “Introducción al Proceso Software Personal”, Pearson

Contenido

INTRODUCCIÓN ..... **II**

INTENCIÓN EDUCATIVA ..... **III**

OBJETIVOS GENERALES ..... **IV**

OBJETIVOS ESPECÍFICOS ..... **IV**

METODOLOGÍA ..... **V**

FUENTES DE INFORMACIÓN ..... **V**

1. LA ADMINISTRACIÓN DEL TIEMPO **1**

INTRODUCCIÓN 1
OBJETIVO GENERAL 1
OBJETIVOS ESPECÍFICOS 1
1.1. UNA DISCIPLINA EN LA INGENIERÍA DE SOFTWARE 1
EJERCICIO 51
EJERCICIO 53
1.2. ¿QUÉ ES PSP? 56
1.2.1 Principios en los que se basa PSP *58*
Ejercicio 59
1.2.2. El modelo PSP *60*
1.2.2.1 El ciclo de Mejora Continua 63
FUENTES DE REFERENCIA BIBLIOGRÁFICA 73

2. EL TAMAÑO DEL PRODUCTO **74**

INTRODUCCIÓN 74
OBJETIVO GENERAL 74
OBJETIVOS ESPECÍFICOS 74
2.1. ¿POR QUÉ PREOCUPARNOS POR EL TAMAÑO DEL PRODUCTO? 75
2.1.1. Tamaño del Producto *75*
2.1.1.1. Tamaño y Esfuerzo 78
2.1.1.2. Tamaño del Código 78
EJERCICIOS 84

3. LA PLANEACIÓN DEL PRODUCTO **87**

INTRODUCCIÓN 87
OBJETIVO GENERAL 87
OBJETIVOS ESPECÍFICOS 87
3.1. ESTIMACIÓN DEL TAMAÑO 88
3.2. MÉTODOS DE ESTIMACIÓN 90
3.2.1 Wideband-Delphi *91*
3.2.2 Estimación por analogía *92*
3.2.3 PERT *93*
3.3. DISTRIBUCIÓN DEL TIEMPO EN LAS FASES 95
EJERCICIO DE PROGRAMACIÓN 95

4. LA AGENDA DE TRABAJO **98**

INTRODUCCIÓN 98
OBJETIVO GENERAL 98
OBJETIVOS ESPECÍFICOS 98
4.1. ¿POR QUÉ ES IMPORTANTE PLANEAR? 99
4.1.1. ¿Qué tomamos en cuenta para planear? *101*

4.1.2. Pasos para Planear *101*
4.2. ESTIMACIÓN DE LA AGENDA 102
Ejemplo de una gráfica de Gantt *103*

5. SEGUIMIENTO DE LA AGENDA DE TRABAJO 106

Ejemplo de Planeación de la Agenda *107*
5.1. COMPLETANDO EL PLAN 109
5.2. VALOR GANADO 110
5.2.2. Establecimiento del Valor Planeado *110*
5.2.2.1 Dándole seguimiento al plan *112*
5.3. PROYECTANDO LA FINALIZACIÓN DEL PROYECTO 113
EJERCICIO DE PROGRAMACIÓN 115

6. LA ADMINISTRACIÓN DE LOS DEFECTOS 118

INTRODUCCIÓN 118
OBJETIVO GENERAL 118
OBJETIVOS ESPECÍFICOS 118
6.1. LA ADMINISTRACIÓN DE DEFECTOS 119
6.1.1. Pero, ¿qué es un defecto? *119*
6.1.2. Defecto vs. Error *119*
6.1.3. Remoción vs. Prevención *120*
6.1.4. La Calidad del Software y los Defectos *120*
6.1.5. Costos de Encontrar y Corregir Defectos *121*
6.1.6. ¿Quién debe remover los defectos? *121*
6.1.7. El Proceso de Administración de Defectos *121*
6.1.8. Clasificación de Defectos *122*
6.1.9. ¿Cómo saber qué registrar como defecto? *127*
6.1.9.1. ¿Cómo encontrar los defectos? 127
6.1.9.2. Densidad de Defectos 128
6.1.9.3. Eficiencia de Remoción 128
EJERCICIO DE PROGRAMACIÓN 129

1. La Administración del Tiempo

Introducción

Uno de los puntos principales para poder mejorar en cualquier área, es primero conocer qué es lo que hacemos y cómo lo hacemos, por ello, debemos iniciar el registro de las actividades al desarrollar productos de software y cuánto tiempo invertimos en ello. Esta información servirá para mejorar nuestro proceso de desarrollo de software e iniciar la creación de una base de datos que nos sirva para predecir con mayor exactitud nuestros trabajos futuros.

Comenzaremos estableciendo la necesidad de contar con mecanismos sistemáticos para producir software en tiempo, costo y calidad. Describiremos lo que es un proceso de software, para lo cual nos centraremos a analizar lo que desarrollemos antes de ponerlo en práctica. También daremos una descripción rápida de la metodología de PSP®, en la cual está basada este curso.

Objetivo General

Al terminar el curso, el alumno:

- Describirá y aplicará mecanismos para la administración del tiempo.

Objetivos Específicos

Al terminar el curso, el alumno:

- Distinguirá entre las actividades productivas y aquellas que nos distraen del trabajo.
- Medirá el tiempo que nos toma realizar las distintas actividades.

1.1. Una disciplina en la Ingeniería de Software

El objetivo de la Ingeniería de Software es proveer un marco de trabajo para mejorar la capacidad individual de producir productos y brindar servicios de software de calidad, de una manera sistemática aplicando principios de ingeniería.

Día tras día el software esta presente en más lugares y dispositivos de uso común, así por ejemplo, podemos encontrar hoy en día software en:

- Juegos de vídeo.
- Sistemas de navegación aérea.
- Sistemas controladores de automóviles, navegación marítima.

El software ha dejado desde hace tiempo ser un "ente" de laboratorio, ha traspasado las barreras de los grandes corporativos para estar instalado en ámbitos de uso cotidiano. Hoy en día, muchos países han tomado como estrategia de desarrollo económica el impulsar la industria del software, entre ellos por ejemplo, México. Muchas de las "comodidades modernas", que pasan desapercibidas contienen algún tipo de software, como por ejemplo:

- Dispositivos del hogar como: lavadoras, refrigeradores, hornos de microondas, etc.
- Dispositivos de control como: señalizadores, máquinas industriales, etc.
- Sistemas médicos.
- Sistemas militares.

Son muchas las actividades que dependen o hacen uso del software.

- ¿A quién de nosotros nos gustaría que los frenos de nuestro automóvil fallaran por un error en el software?
- ¿A quién le gustaría que le explotara el horno de microondas o se perdiera el avance en nuestro vídeo juego favorito por una falla en el software?
- ¿A quién le gustaría que durante sus transacciones electrónicas financieras se le perdieran algunos centavos?

Como forma de involucrar a los alumnos de entender y darse cuenta de la importancia que representa tener productos de software correctos, completos, seguros y confiables, dar ejemplos de desventajas e incomodidades que podrían presentarse al funcionar mal un programa de software.

¡La calidad de los productos y servicios de software debe estar entre las más altas!

¡La ética de los encargados de desarrollar software también debe estar entre las más altas!

Mucho se ha estado discutiendo últimamente sobre la especialidad de las personas que programan software. Ejemplos abundan, los ejecutivos de cuenta de un banco desarrollando "macros" de hojas de cálculo para realizar algunas prospecciones,

contadores haciendo lo mismo para elaborar las deducciones de impuestos y la tendencia creciente de los sistemas de programación orientados hacia el usuario final. Sin embargo, de lo que estamos hablando es de la creación de productos de software que van a aportar, como por ejemplo, a realizar operaciones quirúrgicas en personas, controlar misiones espaciales, modificar sistemas financieros que mueven millones de dólares en horas. El personal que desarrolle este tipo de software debe estar preparado para ello y tener una solvencia profesional para poderle confiar la información y la automatización de los procesos críticos.

Objetivos en el Desarrollo de SW

Tradicionalmente se ha fallado en cumplir con entregar el producto de software ha tiempo, al igual, más de las veces, se ha sobrepasado el costo establecido del producto o servicio y al final, su calidad no ha resultado la esperada.

Los objetivos de la Ingeniería de Software se han definido, tomadas de algunas definiciones de lo que es en sí misma la Ingeniería de Software, como el desarrollo de programas de computadoras en el tiempo, costo y calidad establecida. Sin embargo existen muchos datos estadísticos que demuestran que estos objetivos no se alcanzan en más del 60% de los proyectos de desarrollo de software.

Un ejercicio interesante para los alumnos que les permitirá conocer esta situación es crear un resumen del reporte anual de Standish Group sobre la situación de la industria del software.

¿Qué tan malo ha sido el desarrollo de Software?

A lo largo de la historia del desarrollo de software se tienen ejemplos de que en su construcción no se sigue un enfoque sistemático, ingeniería.

Por ejemplo, de datos tomados de estudios realizados por Standish Group se tiene:

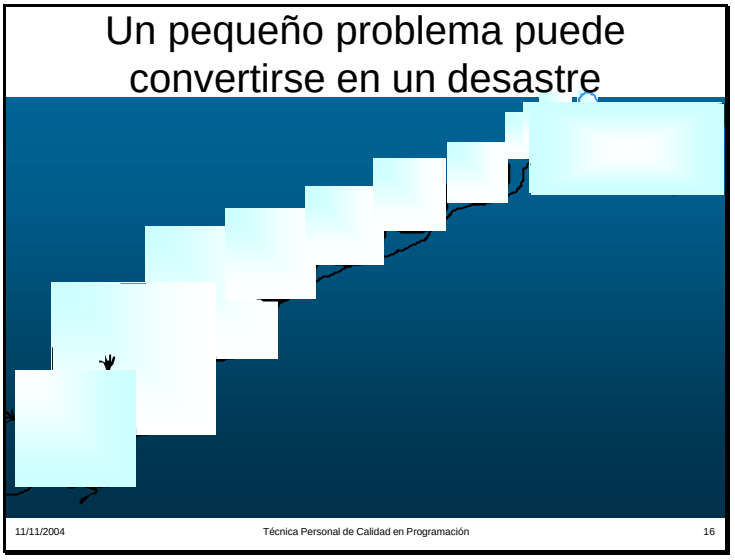
Año	Éxito	Fallo	Cancelado
1994	16%	31%	53%
1996	27%	40%	33%
1998	26%	28%	46%

Después de los primeros 35 seg. del despegue del Cohete Ariane 5 en 1996, se detonó porque amenazaba con caer en una zona poblada.

El sistema de bandas para el manejo de las maletas en el aeropuerto de Denver, Colorado, EEUU, no funcionaba para la fecha de inauguración del aeropuerto.

Durante la guerra del Golfo, un misil Patriot falló el detectar un misil Scud.

¡Todo por algún tipo de falla en el software!



Aquí lo que se trata de mostrar es la “reacción en cadena” que desata un fallo.

Ejercicio:

¿Qué otros ejemplos de desastres por culpa del software conoces?

¿Alguno donde se hayan perdido vidas humanas?

Investiga y documenta algunos de ellos

Este es un ejercicio que sirve para darnos cuenta de la reputación que ha adquirido el personal de desarrollo de software. Se puede complementar con un análisis del código de ética para los Ingenieros de Software de la IEEE-ACM haciendo el análisis entre los preceptos del código de ética y la descripción de las situaciones presentadas en los casos documentados de fallas por culpa del software.

Existe una novela, que narra las desventuras ocurridas en una compañía constructoras de robots industriales, escrita por Richard G. Epstein para utilización exclusiva en la academia. En esta historia se narra un accidente donde perdió la vida un operador de estos robots y las investigaciones apuntan a diversos fallos en su construcción, principalmente en la elaboración del software que lo controlaba.

Se anexa un archivo en español sobre el Robot Asesino de Epstein, del cual, podrá encontrar el original en:
http://onlineethics.org/cases/robot/author.html
o
http://portal.acm.org/citation.cfm?id=382055&coll=portal&dl=ACM

Ingeniería de Software I

El Caso del Robot Asesino

Programador de Silicon Valley Acusado por Homicidio no Premeditado

El Error del Programa Causó la Muerte del Operador del Robot

Especial para el SENTINEL-OBSERVER de Silicon Valley

Jane McMurdock, Fiscal de la Ciudad de Silicon Valley, anunció en la fecha la acusación de Randy Samuels con los cargos de asesinato no premeditado. Samuels trabajaba como programador en Silicon Techtronics, Inc., una de las empresas más nuevas de Silicon Valley en la arena de la alta tecnología. El cargo involucra la muerte de Bart Matthews, quien fuera muerto el pasado mes de mayo por un robot de la línea de armado.

Matthews, quien trabajaba como operador de robot en Cybernetics, Inc., en Silicon Heights, fue aplastado y murió a consecuencias de ello, cuando el robot que estaba operando produjo un mal funcionamiento y comenzó a oscilar su "brazo" violentamente. El brazo del robot alcanzó a Matthews, arrojándolo contra una pared. Matthews murió en forma casi instantánea a causa de los golpes recibidos, en un caso que conmocionó e indignó a muchos de Silicon Valley. De acuerdo con el dictamen de los cargos, Samuels fue quien escribió la pieza del programa de computadora en particular, que fue la responsable de la falla del robot. "Hay una evidencia incriminatoria", anunció triunfante McMurdock en una conferencia de prensa mantenida en la Corte.

"Tenemos la fórmula manuscrita, suministrada por el físico del proyecto, que se suponía tenía que programar Samuels. Pero negligentemente malinterpretó la fórmula, y esto llevó a una horrible muerte. La sociedad debe protegerse de los programadores que cometen errores descuidadamente o de lo contrario nadie estará a salvo, y menos que nadie nuestras familias e hijos", dijo.

El SENTINEL-OBSERVER ha podido obtener una copia de la fórmula manuscrita en cuestión. En realidad, existen tres fórmulas similares, garabateadas en un papel amarillo de un block borrador tamaño oficio. Cada una de las fórmulas describe el movimiento del brazo del robot en una dirección: este-oeste, norte-sur y arriba-abajo.

El SENTINEL-OBSERVER mostró las fórmulas a Bill Park, profesor de Física en la Universidad de Silicon Valley. Éste confirmó que estas ecuaciones podían ser usadas para describir el movimiento del brazo de un robot.

El SENTINEL-OBSERVER mostró entonces el código del programa a Bill Park, escrito por el acusado en lenguaje C de programación. Preguntamos al Profesor Park, quien está muy familiarizado con éste y muchos otros lenguajes de programación, si el código era o no correcto para las fórmulas dadas del brazo del robot.

La respuesta del Profesor Park fue inmediata: "¡No puede ser! ¡Parece que interpretó los puntos *y* de las fórmulas como barras *y,* e hizo lo mismo con las *x* y las *z.* Se suponía que tenía que usar las derivadas, pero en su lugar tomó los promedios!. Si me preguntan, ¡está muy claro que es culpable!"

El SENTINEL-OBSERVER no pudo contactar a Samuels para entrevistarlo. "Se encuentra profundamente deprimido por todo esto", nos dijo su novia por teléfono. "Pero, Randy cree que va a aliviarse en cuanto pueda decir su versión de la historia."

Los que Desarrollaron al "Robot Asesino" Trabajaron bajo una Enorme Presión

Especial para el SENTINEL-OBSERVER de Silicon Valley

Silicon Valley, EEUU
por Mabel Muckraker

El SENTINEL-OBSERVER tomó conocimiento hoy que Randy Samuels y otros que trabajaron en el proyecto del "Robot Asesino" en Silicon Techtronics, estuvieron bajo tremendas tensiones para finalizar el software del robot para el 1° de enero de este año. Según una fuente bien informada, los altos niveles gerenciales advirtieron a los integrantes del staff del proyecto que "rodarían cabezas" si no se cumplía el objetivo del 1° de enero.

Randy Samuels, programador de Silicon Techtronics, fue acusado la semana pasada bajo los cargos de asesinato no premeditado en el ahora famoso "caso del robot asesino". Samuels escribió el software defectuoso que causó que el robot industrial de Silicon Techtronics, Robbie CX30, aplastara y lesionara fatalmente a su operador, Bart Matthews. Matthews era un operador de robot en Cybernetics, Inc. Conforme a la Fiscal de Silicon Valley, Jane McMurdock, Samuels malinterpretó la fórmula matemática, "volviendo al inofensivo Robbie un asesino".

Nuestra fuente informada, quien desea mantenerse en el anonimato y a la que llamaremos "Marta" por el resto de este artículo, tiene un íntimo conocimiento de todos los aspectos del proyecto Robbie CX30. Marta dijo al SENTINEL-OBSERVER que existía una enorme fricción entre el Jefe de División Robótica, Ray Johnson y el Gerente del Proyecto Robbie CX30, Sam Reynolds. "Se odiaban a muerte" manifestó Marta al SENTINEL-OBSERVER en una entrevista exclusiva.

"Hacia junio del año pasado el proyecto se encontraba atrasado seis meses y Johnson se puso furioso. Había rumores de que echarían a toda la División Robótica, que él lideraba, si Robbie [el robot CX30] no daba muestras de ser un éxito comercial. Él (Johnson) llamó a Sam (Reynolds) a su oficina y realmente lo destruyó. Quiero decir, uno podía oír los gritos desde el fondo de la oficina. Johnson le dijo a Sam que terminara el proyecto para el primero de enero o de lo contrario "rodarían cabezas".

"Yo no estoy diciendo que Johnson le ordenara a Sam acortar camino", agregó Marta. "Creo que la idea de cortar camino estaba implícita. El mensaje fue: "acorta camino si quieres mantener tu puesto".

De acuerdo con documentos provistos por Marta al SENTINEL-OBSERVER, el 12 de junio del año pasado fueron agregados al proyecto Robbie CX30 veinte nuevos programadores. Esto ocurrió algunos días después de la tormentosa reunión entre Johnson y Reynolds que Marta contó.

De acuerdo a Marta, los nuevos contratados eran un desastre. "Johnson, unilateralmente, hizo los arreglos de estas contrataciones, seguramente desviando recursos de otros aspectos del proyecto Robbie (CX30). Reynolds se oponía con vehemencia a esto. Johnson sólo conocía acerca de la fabricación de hardware. Esa era su especialidad. No pudo haber entendido las dificultades que nosotros estábamos teniendo con el software de la robótica. Usted no puede acelerar un proyecto de software agregando más gente. No es como en una línea de montaje".

Según Marta y otras fuentes dentro del proyecto, la contratación de estos nuevos veinte programadores llevó a que se hiciera una reunión entre Johnson, Reynolds y todos los integrantes del proyecto de software del Robbie CX30. “Esta vez fue Sam (Reynolds) el que se puso furioso. Se quejó de que el proyecto no necesitaba más gente. Sostuvo que el problema principal era que Johnson y otros miembros a nivel directivo no entendían que el Robbie CX30 era fundamentalmente diferente de versiones anteriores del robot”. Estas fuentes dijeron al SENTINEL-OBSERVER que los nuevos empleados no estaban totalmente integrados al proyecto, aún seis meses después de su ingreso, cuando diez robots Robbie CX30, incluido al robot que mató a Bart Matthews, ya habían sido despachados. Según Marta, “Sam sólo quería mantener las cosas lo más simples posible. No quería que el nuevo personal complicara las cosas. “Se pasaron seis meses leyendo manuales. La mayoría de los empleados nuevos no sabían nada de robots y Sam no estaba como para perder su tiempo tratando de enseñarles”. Según Marta, la reunión del 12 de junio se hizo famosa en la corporación Silicon Techtronics porque fue en esa reunión donde Ray Johnson anunció su “Teoría Ivory Snow [“no existe el blanco perfecto, o bien, no hay blanco más blanco que el blanco nieve”] de diseño y desarrollo de software. De acuerdo a Marta, “Ray (Johnson) nos dio una gran presentación en multimedia, con diapositivas y todo. La esencia de esta “Teoría Ivory Snow” es simplemente que el blanco nieve es 99,44 por ciento puro y que no hay razón por la que el software de robótica deba ser más puro que esto. Dijo repetidas veces que ‘El software perfecto era un oxímoron’”.

Marta y otros personajes anónimos que se acercaron con información, retrataron a Johnson como un gerente con una desesperada necesidad de ser ayudado por un éxito en el proyecto. Versiones anteriores de Robbie, CX10 y CX20, fueron experimentales en naturaleza y nadie esperaba que fueran éxitos comerciales. De hecho, la División Robótica de Silicon Techtronics estaba operando con sus finanzas en rojo desde su concepción seis años atrás. O triunfaba el CX30 o Silicon Techtronics quedaría fuera del negocio de robótica industrial. “Los robots Robbie anteriores tuvieron mucha prensa, especialmente acá en Silicon Valley”, dijo otra fuente que también quiere permanecer anónima. “Robbie CX30 iba a capitalizarse con la buena publicidad generada por los proyectos anteriores. Lo único es que Robbie CX30 era más revolucionario de lo que Johnson quería admitir. CX30 representaba un paso gigante hacia adelante en términos de sofisticación. Había muchísimas preguntas acerca de los parámetros industriales en los que debería trabajar el CX30. Mucho de lo que debía ejecutar era completamente nuevo, pero Johnson nunca lo pudo entender. Él sólo nos veía como unos perfeccionistas. Uno de sus dichos favoritos era ‘La perfección es la enemiga de lo bueno’”. **Los Compañeros Acusan: el programador del “Robot Asesino” se Creía una Estrella**

Especial para el SENTINEL-OBSERVER de Silicon Valley

Silicon Valley, EEUU
por Mabel Muckraker

Randy Samuels, el que fuera programador de Silicon Techtronics que fue acusado por escribir el software que causó el horrible incidente del “robot asesino” el pasado mes de mayo, era aparentemente una ‘prima donna’ que encontraba muy difícil aceptar críticas, aseguraron hoy varios compañeros de trabajo.

En una rueda de prensa con varios compañeros de trabajo de Samuels en el proyecto del “robot asesino”, el SENTINEL-OBSERVER pudo obtener importantes revelaciones acerca de la psiquis del hombre que puede haber sido criminalmente responsable de la muerte de Bart Matthews, operador de robot y padre de tres criaturas.

Con el permiso de los entrevistados, el SENTINEL-OBSERVER permitió a la profesora Sharon Skinner del Departamento de Psicología de Software en la Universidad de Silicon Valley, escuchar una grabación de la entrevista. La Profesora Skinner estudia la psicología de los programadores y otros factores psicológicos que tienen un impacto en el proceso de desarrollo del software.

"Estaría de acuerdo con la mujer que lo llamó 'prima donna'", explicó la Profesora Skinner. "Este es un término utilizado para referirse a un programador que simplemente no puede aceptar las críticas, o más precisamente, no puede aceptar su propia falibilidad".

"Randy Samuels tiene lo que nosotros, psicólogos de programadores, llamamos una personalidad orientada hacia una tarea, lindando con una personalidad orientada hacia sí mismo. Le gusta poder completar cosas, pero su ego está muy densamente involucrado en su trabajo. En el mundo de la programación esto se lo considera un "no, no", agregó el Profesor Skinner en su oficina tapizada de libros.

La Profesora Skinner continuó explicando algunos hechos adicionales sobre equipos de programación y personalidades del programador. "Básicamente, hemos encontrado que un buen equipo de programación requiere de una mezcla de tipos de personalidad, incluyendo a una persona que esté orientada hacia la interacción, que saca una enorme satisfacción del hecho de trabajar con otra gente, alguien que pueda ayudar a mantener la paz y a que las cosas se muevan en una dirección positiva. Muchos programadores están orientados hacia lo que es la tarea, y esto puede ser problemático si se tiene un equipo donde todos son de este modo."

Los compañeros de trabajo de Samuels se mostraron muy reticentes a culpar a alguien por el desastre del robot, pero cuando se los presionó para que comentaran sobre la personalidad de Samuels y sus hábitos laborales, surgieron varios hechos importantes. Samuels trabajaba en un equipo formado por aproximadamente una docena de analistas, programadores y testers de software. (Esto no incluye a veinte programadores que fueron incorporados posteriormente y que nunca llegaron a estar activamente involucrados en el desarrollo del software de la robótica). Si bien cada individuo del equipo poseía una especialidad, casi todos estaban comprometidos en todo el proceso de software de principio a fin.

"Sam Reynolds tenía un background en procesamiento de datos. Dirigió unos cuantos proyectos de software de esa naturaleza", dijo uno de los integrantes del equipo, refiriéndose al gerente del proyecto Robbie CX30. "Pero su rol en el proyecto era más que nada de líder. Asistía a todas las reuniones importantes y lo mantenía a Ray (Ray Johnson, el Jefe de División Robótica) sobre nuestras espaldas lo más posible." San Reynolds, como ya fuera informado en el SENTINELOBSERVER de ayer, se encontraba bajo una severa presión para lograr producir un robot Robbie CX30 operativo para el 1 de enero de este año. Sam Reynolds no pudo ser ubicado para entrevistarlo ya sea sobre su rol en el incidente o sobre Samuels y sus hábitos en el trabajo.

"Éramos un equipo democrático, a excepción del liderazgo provisto por Sam (Reynolds)", observó otro miembro del equipo. En el mundo del desarrollo de software, un equipo democrático es un equipo en donde todos los miembros de éste tienen un decir igual en el proceso de toma de decisiones. "Desafortunadamente, nosotros éramos un equipo de individualistas muy ambiciosos, muy talentosos - si debo referirme a mí mismo - y muy opinadores. Randy (Samuels) era justo el peor del grupo. Lo que quiero decir es que teníamos, por ejemplo, a dos chicos y a una chica con masters de CMU, y no eran tan arrogantes como Randy. CMU significa Universidad de Carnegie-Mellon, una líder nacional en enseñanza de ingeniería de software.

Un compañero comentó sobre un incidente que Samuels causó en una reunión de Quality Assurance. Esta reunión involucraba a Samuels y a tres revisores de un módulo de

software que Samuels había diseñado e implementado. Tales reuniones son llamadas "revisiones de código". Uno de los revisores mencionó que Samuels había usado un algoritmo sumamente ineficiente para lograr un determinado resultado y Samuels "se puso todo colorado". Empezó a gritar una sarta de obscenidades y después se levantó y se fue. Nunca regresó.

"Le enviamos un memo con un algoritmo más rápido y a su tiempo usó este algoritmo en su módulo", agregó el colega.

El módulo de software del incidente de la reunión de Quality Assurance fue el primero en ser identificado como una falla en el "asesino" del operador de robot. No obstante, este colega se apuró a señalar que la eficacia del algoritmo no era un tópico en el mal funcionamiento del robot. Era sólo que Randy hacía muy difícil para la gente el poderle comunicar las observaciones. Se tomaba todo muy a pecho. Se graduó con el puntaje más alto de su clase y luego se recibió con honores en ingeniería de software en Purdue. Definitivamente es muy inteligente."

"Randy, en su pared, tiene este inmenso cartel hecho en Banner", continuó este colega, "Decía, "DENME LA ESPECIFICACIÓN Y LES DARÉ EL PROGRAMA DE COMPUTACIÓN". Ese es el tipo de arrogancia que tenía y también demuestra que tenía muy poca paciencia para desarrollar y verificar las especificaciones. Amaba el aspecto de solucionar el problema, la programación propiamente dicha". No pareciera que Randy Samuels quedó atrapado en el espíritu de la "programación sin egolatría", observó la Profesora Skinner cuando escuchó esta parte de la entrevista con los colegas de trabajo de Samuels. "La idea de una programación sin egocentrismo es que el producto de software pertenece al equipo y no a los programadores individuales. La idea es estar abierto a las críticas y estar menos atado al trabajo propio. Ciertamente que la tarea de revisión de código es coherente con esta filosofía en general." Una colega habló acerca de otro aspecto de la personalidad de Samuels: su capacidad de ayuda. "Randy odiaba las reuniones, pero era muy bueno con las relaciones de uno a uno. Siempre estaba ansioso por ayudar. Recuerdo una vez que me encontraba encerrada en un camino sin salida y él, en vez de tan sólo señalarme la dirección correcta, se hizo cargo del problema y lo resolvió él mismo. Se pasó cerca de cinco días completos en mi problema". "Por supuesto que mirado en retrospectiva, hubiera sido mejor para el pobre Sr. Matthews y su familia que Randy se hubiese dedicado tan sólo a sus propias cosas", agregó luego de una larga pausa.

El Proyecto del "Robot Asesino" Controvertido desde el Vamos

Bandos Enfrentados por el Modo en que Debía Proseguir el Proyecto

Especial para el SENTINEL-OBSERVER de SILICON VALLEY

Silicon Valley, EEUU
por Mabel Muckraker

Dos grupos, comprometidos con diferentes filosofías de desarrollo de software, casi se enfrentan violentamente durante las reuniones iniciales de planeamiento para el Robbie CX30, el robot de Silicon Techtronics que mató a un obrero de la línea de ensamble el pasado mes de mayo. Estaba en cuestionamiento si el proyecto Robbie CX30 debía proseguir de acuerdo con el "modelo de cascada" o el "modelo de prototipo" .

El modelo de cascada y el de prototipo son dos métodos comunes de organizar un proyecto de software. En el modelo de cascada, el proyecto de software pasa a través de etapas

definidas de desarrollo. La primera etapa es el análisis de los requerimientos y especificaciones, durante la cual se intenta arribar a un acuerdo en cuanto a la funcionalidad detallada del sistema. A medida que el proyecto pasa de una etapa a la siguiente, existen limitadas oportunidades de dar marcha atrás y cambiar decisiones ya tomadas. Una desventaja de este enfoque es que los usuarios potenciales no tienen oportunidad de interactuar con el sistema recién hasta bien entrado en el ciclo de vida del mismo.

En el modelo de prototipo, se pone un gran énfasis en producir un modelo o prototipo operativo bien temprano durante el ciclo de vida del sistema. El prototipo es construido con el propósito de arribar a una especificación final de la funcionalidad del sistema propuesto. Los usuarios potenciales interactúan con el prototipo en forma temprana y con frecuencia hasta que son acordados los requerimientos. Este enfoque les da a los potenciales usuarios la oportunidad de interactuar con un sistema prototipo en forma temprana durante el ciclo de desarrollo, y mucho antes que el sistema final esté diseñado y codificado.

En un memorando de fecha 11 de diciembre del año pasado, Jan Anderson, un miembro del equipo original del proyecto CX30, atacó duramente la decisión tomada por el gerente de proyecto Sam Reynolds de emplear el modelo de cascada. El SENTINEL-OBSERVER obtuvo una copia del memo de Anderson, dirigido a Reynolds, y Anderson verificó la autenticidad del memorando para este diario.

Reynolds despidió a Anderson el 24 de diciembre, justo dos semanas después que ella escribiera el memo.

El memo de Anderson hace referencia a una reunión anterior en la que ocurrió un fuerte intercambio de opiniones relacionadas con la filosofía del desarrollo del software.

En el memo, Anderson subrayó el siguiente párrafo.:

“No fueron mis intenciones impugnar su competencia durante nuestra reunión de ayer, pero debo protestar con mi mayor vehemencia contra la idea de que completemos el software de Robbie CX30 siguiendo el modelo de cascada que usted ya usó en otros proyectos. No necesito recordarle que aquellos eran proyectos de procesamiento de datos que involucraban el procesamiento de transacciones de negocios. El proyecto Robbie CX30 llevará un alto grado de interacción, tanto entre robot y componentes como entre robot y su operador. Dado que la interacción del operador con el robot es tan importante, la interfaz no puede estar diseñada como una idea de último momento.”

Randy Samuels, a quien se lo acusó de asesinato no premeditado por la muerte del operador Bart Matthews, padre de tres niños, había participado de la reunión del 11 de diciembre.

En una conversación con este diario, Anderson dijo que Samuels no tenía mucho para decir sobre la controversia cascada-prototipo, pero sí afirmó que daría “una mano” con tal que exoneraran a Samuels.

“El proyecto fue sentenciado a muerte mucho antes que Samuels malinterpretara esas fórmulas”, aclaró Anderson enfáticamente en la sala de su casa en los suburbios.

En conversación con este diario, Anderson hizo lo mejor de sí para explicar la controversia del método cascada vs. prototipo en términos simples. “El punto principal en realidad era si podíamos llegar a ponernos de acuerdo en los requerimientos del sistema sin dejar que los operadores del robot presintieran lo que teníamos en mente. Reynolds ha estado en el negocio de procesamiento de datos por tres décadas y es bueno en eso, pero nunca debería haber sido el gerente de este proyecto.”

Conforme a registros obtenidos por el SENTINEL-OBSERVER, Silicon Techtronics transfirió a Sam Reynolds de la División Procesamiento de Datos, que se encargaba de

inventario y salarios, a la División Robótica, justo tres semanas antes de la reunión del 11 de diciembre a que alude Anderson en su memo.

Reynolds fue transferido a la División Robótica por el presidente de Silicon Techtronics, Michael Waterson. Reynolds reemplazaba a John Cramer, quien gerenciaba el anterior proyecto Robbie CX10 y CX20. Cramer fue puesto a cargo del proyecto CX30, pero murió inesperadamente en un accidente aéreo. Al colocar a Reynolds a cargo del proyecto CX30, nos dice nuestra fuente, que Waterson iba en contra del consejo de Ray Johnson, Jefe de la División Robótica. De acuerdo con estas fuentes, Johnson se oponía fuertemente a la alternativa de ponerlo a Reynolds como jefe del proyecto Robbie CX30. Estas fuentes dijeron al SENTINEL-OBSERVER que la elección de Waterson por Reynolds fue puramente una decisión de recorte de gastos. Era más barato transferir a Reynolds a la División Robótica que incorporar a un nuevo líder de proyecto fuera de la corporación.

La fuente anónima que el SENTINEL-OBSERVER llamará “Marta” describió la situación de este modo: “Waterson pensaba que sería más barato transferir a Reynolds a robótica antes que intentar encontrar afuera un nuevo gerente para el proyecto Robbie. Además, Waterson tendía a sospechar de la gente de afuera del grupo. Con frecuencia mandaba memos sobre cuánto tarda la gente en aprender “el modo de hacer las cosas de Silicon Techtronics”. Desde el punto de vista de Waterson, Reynolds era el gerente y fue transferido a su nuevo puesto en Robótica como un gerente y no como un experto técnico. Claramente, Reynolds se veía a sí mismo tanto gerente como experto técnico. Reynolds no tenía conciencia de sus propias limitaciones técnicas.”

Según Marta, Reynolds era muy renuente a gerenciar un proyecto que no usara el modelo de cascada que tan bien le había servido en el procesamiento de datos. Tildó al modelo prototipo como un “modelo de moda” en la reunión del 11 de diciembre, y después de una serie de intercambios verbales la cosa se puso muy personal.

“Anderson estaba especialmente expresiva”, recuerda Marta. “Tenía mucha experiencia con interfaces con usuarios y desde su perspectiva, la interfaz robot-operador era crítica para el éxito del CX30, dado que la intervención del operador sería frecuente y a veces crítica.” En su entrevista con el SENTINEL-OBSERVER, Jan Anderson comentó sobre este aspecto de la reunión del 11 de diciembre:

“Reynolds estaba en contra de “perder el tiempo” - para usar sus propias palabras - con cualquier tipo de análisis formal de las propiedades de los factores humanos y su interfaz con el usuario. Para él, las interfaces con el usuario real eran un tema periférico.”

“Para él [Reynolds], cualquier cosa nueva era “moda”, agrega Anderson. “Las interfaces de la computadora eran una moda, el diseño orientado a objetos era una moda, la especificación formal y las técnicas de verificación eran una moda, y por sobre todo, el modelo prototipo era una moda.”

Justo una semana después de la reunión del 11 de diciembre, el grupo del proyecto Robbie recibió un memo de Sam Reynolds concerniente al plan para el proyecto Robbie CX30.

“Era el modelo de cascada, como salido de un libro”, Anderson dijo a este reportero mientras revisaba una copia del memo con el plan del proyecto. “Análisis de requerimientos y especificación, luego diseño de arquitectura y diseño detallado, codificación, prueba, entrega y mantenimiento. En el modo de ver de Reynolds, no hacía falta tener ninguna interacción del usuario con el sistema hasta muy, pero muy avanzado el proyecto.” El SENTINEL-OBSERVER se ha enterado de que el primer operador que realmente usó el robot Robbie CX30 en una función industrial fue Bart Matthews, el hombre que fue muerto en la tragedia del robot asesino. Este primer uso de Robbie CX30 en un uso industrial fue cubierto por los medios, incluyendo este periódico. Como una gran

ironía, El Informe Anual de Silicon Techtronics para los Accionistas, publicado el pasado mes de marzo, contiene en la brillante portada una foto de un sonriente Bart Matthews. A Matthews se lo muestra operando al mismísimo robot Robbie CX30 que lo aplastó hasta la muerte tan solo dos meses después de la toma fotográfica.

Silicon Techtronics Prometió Entregar un Robot Seguro

Cuestionada la Calidad de Entrenamiento del Operador Especial

para el SENTINEL-OBSERVER de SILICON VALLEY

Silicon Valey, EEUU
por Mabel Muckraker

En una conferencia de prensa de esta tarde, un grupo de programadores que se autodenominan “Comité de Justicia para Randy Samuels”, distribuyó documentos que muestran que Silicon Techtronics se obligó a sí misma a hacer entrega de robots que “no causarían ningún daño corporal a los operadores humanos”. Randy Samuels es el programador que ha sido acusado de asesinato en el infame caso del “robot asesino”.

“No podemos entender como el Fiscal pudo acusar a Randy con esos cargos cuando, de hecho, la compañía Silicon Techtronics estaba legalmente obligada a producir y entregar robots seguros a Cybernetics”, dijo el vocero del comité, Ruth Witherspoon. “Creemos que en todo esto hay un encubrimiento y que hay algún tipo de confabulación entre la gerencia de SiliTech [Silicon Techtronics] y la oficina del Fiscal. Michael Waterson era uno de los más grandes contribuyentes de la campaña de reelección de la Sra. McMurdock del año pasado”. Michael Waterson es Presidente Ejecutivo de Silicon Techtronics. Jane McMurdock es la Fiscal de la ciudad de Silicon Valley. El SENTINEL-OBSERVER confirmó que Waterson hizo varios grandes aportes a la campaña de reelección de McMurdock el otoño pasado.

“A Randy le están haciendo pagar los platos rotos por una empresa que tiene estándares de control de calidad malos y no lo vamos a permitir! Gritó Whiterspoon en una emotiva declaración a los periodistas. “Creemos que la política ha entrado en todo esto”.

Los documentos que fueron distribuidos por el comité por la “Justicia para Randy Samuels” eran porciones de lo que se llama un “documento de requerimientos”. Según Ruth Witherspoon y otros miembros del comité, este documento prueba que Samuels no fue legalmente responsable de la muerte de Bart Matthews, el desafortunado operador de robot que fue muerto por un robot de Silicon Techtronics en Cybernetics, Inc. en Silicon Heights el pasado mes de abril.

El documento de requerimientos es un contrato entre Silicon Techtronics y Cybernetics, Inc. Especifica con total detalle la funcionalidad del robot Robbie CX30 que Silicon Techtronics prometió entregar a Cybernetics. Según Whiterspoon, el robot Robbie CX30 fue diseñado para ser un robot “inteligente” que pudiera ser capaz de operarse en una variedad de funciones industriales. Cada cliente de la corporación necesitó de documentos de requerimientos separados ya que Robbie CX30 no era un “robot de llave en mano” sino un robot que necesitaba ser programado de forma diferente para cada aplicación.

No obstante, todos los documentos de requerimientos que fueron acordados bajo los auspicios del proyecto Robbie CX30, incluyendo al acuerdo entre Silicon Techtronics y Cybernetics, contienen los siguientes fundamentos de importancia:

“El robot será de operación segura y aun bajo circunstancias excepcionales (ver Sección 5.2), el robot no causará daño corporal alguno al operador humano.”

"En el caso de condiciones excepcionales que potencialmente contengan el riesgo de daño corporal (ver Sección 5.2.4 y todas sus subsecciones), el operador humano podrá ingresar una secuencia de códigos de comando, según se describe en las secciones relevantes de la especificación funcional (ver Sección 3.5.2), que detendrá el movimiento del robot mucho antes que pueda ocurrir un real daño corporal." Las "condiciones excepcionales" incluyen eventos inusuales tales como datos extraños desde los sensores del robot, movimiento errático o violento del robot o error del operador. Fue justamente esa condición excepcional la que llevó a la muerte a Bart Matthews.

Estos párrafos fueron extractados de las porciones del documento de requerimientos que trata sobre los "requerimientos no funcionales". Los requerimientos no funcionales listan en detalle las restricciones bajo las cuales operaría el robot. Por ejemplo, el requerimiento de que el robot sería incapaz de dañar a su operador humano es una restricción y Silicon Techtronics, según Ruth Whiterspoon, estaba legalmente obligada a satisfacer este punto. La parte de los requerimientos funcionales del documento de requerimientos cubre (nuevamente en sumo detalle) el comportamiento del robot y su interacción con el entorno y su operador humano. En particular, los requerimientos funcionales especificaban el comportamiento del robot bajo cada una de las condiciones excepcionales esperadas. En su declaración a los periodistas en la conferencia de prensa, Whiterspoon explicó que Bart Matthews fue muerto cuando se produjo la condición excepcional 5.2.4.26. Ésta involucra un movimiento del brazo del robot extremadamente violento e impredecible. Esta condición requiere de la intervención del operador, a saber el ingreso de los códigos de comando mencionados en el documento, pero aparentemente Bart Matthews se confundió y no pudo ingresar con éxito estos códigos.

"Si bien el programa de Randy Samuels estaba mal - él en verdad malinterpretó las fórmulas de la dinámica del robot, como se informó a los medios. La condición excepcional 5.2.4.26 estaba diseñada para protegerse de justamente este tipo de contingencia", dijo Whiterspoon a los periodistas. "Los valores del movimiento del robot generados por el programa de Randy identificaron correctamente a esta condición excepcional y el operador del robot recibió el debido aviso de que algo andaba mal".

Whiterspoon dijo que poseía una declaración firmada de otro operador de robot de Cybernetics en donde afirmaba que las sesiones de entrenamiento ofrecidas por Silicon Techtronics nunca mencionaron a ésta ni a tantas otras condiciones excepcionales. Según Whiterspoon, el operador del robot ha jurado que ni a él ni a ningún otro operador de robot les fue dicho jamás que el brazo del robot podía oscilar violentamente.

Whiterspoon citó esta declaración en la conferencia de prensa. "Ni yo ni Bart recibimos entrenamiento para manejar este tipo de condición excepcional. Dudo mucho que Bart Matthews tuviese idea de qué se suponía debía hacer cuando la pantalla de la computadora comenzó a mostrar el mensaje de error".

Las condiciones excepcionales que requieren de la intervención del operador causan un mensaje de error que se genera en la consola del operador. La Policía de Silicon Valley confirmó que cuando Bart Matthews fue muerto, el manual de referencia en su consola estaba abierto en la página del índice que contenía los códigos de ingreso para los "errores".

Witherspoon luego citó secciones del documento de requerimientos que obligan a Silicon Techtronics (el vendedor) a entrenar adecuadamente a los operadores de robots:

"El vendedor suministrará cuarenta (40) horas de entrenamiento para los operadores. Este entrenamiento cubrirá todos los aspectos de la operación del robot, incluyendo una cobertura exhaustiva de los procedimientos de seguridad que deben seguirse en caso de condiciones excepcionales que contengan

potencialmente el riesgo de daño corporal. El vendedor proveerá y administrará instrumentos de prueba apropiados que serán usados para certificar el suficiente entendimiento del operador de las operaciones de la consola del robot y de los procedimientos de seguridad. Sólo los empleados del cliente que hayan pasado estas pruebas estarán habilitados para operar el robot Robbie CX30 en una verdadera función industrial.

"El manual de referencia deberá suministrar instrucciones claras para la intervención del operador en todas las situaciones excepcionales, especialmente e inclusive aquellas que contengan potencialmente el riesgo de daño corporal."

Según Whiterspoon, las declaraciones juradas de varios operadores de robots de Cybernetics Inc., aseguran que sólo se destinó un día laborable (aproximadamente 8 horas) al entrenamiento de los operadores. Es más, casi no se dedicó tiempo alguno al tratamiento de condiciones excepcionalmente peligrosas.

"La prueba escrita desarrollada por Silicon Techtronics para habilitar a un operador de robot era considerada por los empleados de Cybernetics como un chiste", aseguró Whiterspoon. "Obviamente Silicon Techtronics no le dedicó mucho al entrenamiento y procedimientos de prueba obligatorios según el documento de requerimientos según la evidencia en nuestro poder".

Reimpreso con el permiso de ROBOTIC WORLD
El diario de ROBOTICS AND ROBOTICS APPLICATIONS

La Interfaz del "Robot Asesino"

Dr. Horace Gritty, Departamento de Ciencias de la Computación y materias relacionadas Universidad de Silicon Vallley
Silicon Valley, EEUU

Resumen: El robot industrial Robbie CX30 se suponía que debía establecer un nuevo modelo de inteligencia de robots industriales. Desgraciadamente, uno de los primeros robots Robbie CX30 mató a un obrero de la línea de montaje, y esto llevó a la acusación de uno de los que desarrollaron el software del robot, Randy Samuels. Este artículo promueve la teoría de que quien debería estar en juicio en este caso sería el diseñador de la interfaz robot-operador. El robot Robbie CX30 viola casi todas las reglas del diseño de interfaz. Este artículo se centra en cómo la interfaz del Robbie CX30 violó cada una de las "Ocho Reglas de Oro" de Shneiderman.

1. Introducción

El 17 de mayo de 1992, un robot industrial Robbie CX30 de Silicon Techtronics mató a su operador, Bart Matthews, en Cybernetics Inc., en Silicon Heights, un suburbio de Silicon Valley. La investigación de los hechos del accidente guiaron a las autoridades a la conclusión de que un módulo de software, escrito y desarrollado por Randy Samuels, un programador de Silicon Techtronics, fue el responsable del comportamiento errático y violento que a su vez llevó a la muerte por decapitación de Bart Matthews [NOTA AL PIE. Los medios de prensa manejaron la información haciendo creer que Bart Matthews había sido aplastado por el robot, pero la evidencia fotográfica dada a este autor muestra otra cosa. Tal vez, las autoridades trataban de proteger la sensibilidad pública].

Como experto en el área de interfaces con el usuario (1, 2, 3), se me pidió prestar ayuda a la policía en la reconstrucción del accidente. Para poder hacer esto, se le pidió a Silicon Techtronics que me suministrara un simulador de Robbie CX30 que incluyera por completo la consola del operador del robot. Esto me permitió investigar el

comportamiento del robot sin tener que en realidad arriesgarme seriamente. Debido a mi profundo entendimiento de interfaces con el usuario y factores humanos pude reconstruir el accidente con extrema precisión. Sobre la base de esta reconstrucción, llegué a la conclusión de que fue el diseño de la interfaz y no el por cierto imperfecto diseño del software lo que debería haber sido visto como el criminal en este caso.

A pesar de mi descubrimiento, la Fiscal Jane McMurdock insistió en proseguir el caso contra Randy Samuels. Pienso que cualquier Computador Científico competente, dado una oportunidad de interactuar con el simulador del Robbie CX30, también habría concluido que el diseñador de la interfaz y no el programador deberían haber sido acusado por negligencia, si no por homicidio no premeditado.

2. "Las ocho reglas de oro" de Shneiderman

Mi evaluación de la interfaz con el usuario del Robbie CX30 está basada en las "ocho reglas de oro" de Shneiderman (4). También utilicé otras técnicas para evaluar la interfaz, pero éstas serán publicadas en artículos separados. En esta sección ofrezco una breve revisión de las ocho reglas de oro de Shneiderman, un tema que resultará más familiar para expertos en interfaces de computación como yo y no a hackers de robótica que leyeron este oscuro periódico.

Las ocho reglas de oro son:

1. Buscar siempre la coherencia. Tal como se puede ver más abajo, es importante que una interfaz con el usuario sea coherente a muchos niveles. Por ejemplo, los diseños de pantalla deben ser coherentes de una pantalla a la siguiente. En un entorno en que se usa una interfaz gráfica con el usuario (GUI), esto también implicará concordancia de un utilitario al siguiente.
2. Permitirle a los usuarios frecuentes el uso de métodos abreviados por teclas. Los usuarios frecuentes (o, 'power users') pueden desalentarse ante tediosos procedimientos. Permitirle a estos usuarios un procedimiento menos tedioso para completar una tarea dada.
3. Dar realimentación de información. Los usuarios necesitan ver las consecuencias de sus acciones. Si un usuario ingresa un comando pero la computadora no muestra ya sea que lo está procesando o lo ha procesado, esto puede dejar al usuario confundido o desorientado.
4. Diseñar diálogos que tengan un fin. Interactuar con una computadora es algo así como dialogar o conversar. Cada tarea debe tener un inicio, un desarrollo y un fin. Es importante que el usuario sepa cuándo una tarea está finalizada. El usuario necesita tener la sensación de que la tarea ha alcanzado su término.
5. Permitir manejos simples de los errores. Los errores del usuario deben estar diseñados dentro del sistema. Otro modo de decirlo es que no debe haber ninguna acción por parte del usuario que sea considerada como un error que está más allá de la capacidad del sistema para manejarlo. Si el usuario comete un error, debe recibir información útil, clara y concisa sobre la naturaleza de tal error. Debe resultar fácil para el usuario deshacer este error.
6. Permitir deshacer las acciones con facilidad. Más genéricamente, los usuarios deben poder deshacer lo que han hecho, sea esto o no de naturaleza errónea.
7. Respaldar que el centro del control esté internamente. La satisfacción del

usuario es alta cuando el usuario tiene la sensación de que es él o ella quien tiene el control y la satisfacción del usuario es baja cuando el usuario siente que la computadora tiene el control. Diseñar interfaces para reforzar la sensación de que es en el usuario donde yace el control en el ámbito de la interacción humano-computadora.

8. Reducir la carga de la memoria inmediata. La memoria inmediata del ser humano es notablemente limitada. Los psicólogos regularmente citan a la ley de Miller que dice que la memoria inmediata está limitada a siete piezas discretas de información. Hacer todo lo posible para liberar la carga en la memoria del usuario. Por ejemplo, en lugar de pedirle al usuario que teclee el nombre de un archivo para abrirlo, presentar al usuario una lista de archivos disponibles en ese momento.

3. Revisión de la consola del robot

La interfaz del operador de Robbie CX30 violó todas y cada una de las reglas de Shneiderman. Muchas de estas violaciones fueron directamente responsables del accidente que terminó con la muerte de un operador de robot.

La consola del robot era una IBM PS/2 modelo 55SX con un procesador 80386 y un monitor EGA color con resolución 640 x 480. La consola tenía un teclado, pero no ratón. La consola estaba empotrada en una estación de trabajo que tenía, además, estantes para manuales y un área para tomar notas y para leer manuales. No obstante, el área para escribir/leer estaba a bastante distancia de la pantalla de la computadora, o sea que era bastante incómodo y cansado para el operador manejar cualquier tarea que requiriera de mirar algo en el manual y luego actuar rápidamente con respecto al teclado de la consola. La silla del operador estaba deficientemente diseñada y estaba demasiado alta con respecto a la posición de la consola y al área de escribir/leer. Esto resentía mucho la espalda del operador y también causaba excesivo cansancio de la vista.

No puedo comprender cómo un sistema sofisticado como este no pudo incluir un aparato de mejor diseño para los ingresos de los datos. Uno sólo podría concluir que Silicon Techtronics no tenía mucha experiencia en tecnología de interfaces con el usuario. El documento de requerimientos (5) especificaba un sistema manejado por menús, lo cual era una elección razonable. Sin embargo, en un utilitario en donde lo esencial era la rapidez, especialmente cuando la seguridad del operador estaba en juego, el uso de un teclado para todas las tareas de selección de opción de menús fue una elección de extremada pobreza, que requería de mucho uso del teclado para lograr el mismo efecto que podía haberse conseguido casi instantáneamente mediante un ratón. (Ver el artículo escrito por Foley et al. (6) En realidad, se me ocurrieron todas estas ideas antes que Foley las publicara, pero él me ganó).

El operador del robot podía interactuar con el robot y de este modo producir un impacto sobre su comportamiento al hacer las opciones en un sistema de menús. El menú principal consistía de 20 ítems, demasiados en mi opinión, y cada ítem del menú principal tenía un submenú tipo desplegable asociado a éste. Algunos de los submenús tenían tanto como veinte ítems - nuevamente, demasiados. Es más, parecía haber muy poca rima o razón en cuanto a por qué los ítems de los menús estaban listados en el orden en que lo estaban. Hubiese sido mucho mejor una organización alfabética o funcional.

Algunos del los ítems en los menús desplegables tenían hasta cuatro menús pop up relacionados a éstos. Éstos aparecerían en secuencias a medida que se hacía la selección correspondiente en los submenús. Ocasionalmente, una elección de un submenú abriría un cuadro de diálogo en la pantalla. Un cuadro de diálogo requiere de cierto tipo de

interacción entre el operador y el sistema, por ejemplo para resolver ciertos temas, como ser, ingresar cuál es el diámetro de un dispositivo dado a ser bajado en el baño de ácido.

El sistema de menús presenta una estricta jerarquía de elección de menús. El operador podría volver hacia atrás en esta jerarquía apretando la tecla de escape. Esta tecla escape también podría terminar los diálogos.

El uso de color en la interfaz fue muy poco profesional. Había demasiados colores en un espacio demasiado chico. Los contrastes eran muy fuertes y el resultado, para este revisor, resultó en una severa fatiga ocular en tan sólo quince minutos de uso. Hubo uso excesivo de efectos musicales tontos y flashes cuando se ingresaban opciones o códigos erróneos.

Uno debería preguntarse por qué Silicon Techtronics no intentó un enfoque más sofisticado para el diseño de la interfaz. Luego de un cuidadoso estudio del dominio de los utilitarios del Robbie CX30, he llegado a la conclusión de que una interfaz de manipulación directa, que mostrara literalmente al robot en la consola del operador, habría sido lo ideal. El entorno tan visual dentro del cual operaba el robot se prestaba naturalmente al diseño de metáforas de pantalla apropiadas para ese entorno, metáforas que el operador podría entender con facilidad. Esto permitiría que el operador manipulara el robot mediante el manejo, en la consola del robot, de la representación gráfica del robot en su entorno. He solicitado a uno de mis estudiantes en el doctorado, Susan Farnsworth, que investigara un poco más esta posibilidad.

4. En qué modo la interfaz del robot Robbie CX30 violó las ocho reglas de oro

La interfaz con el usuario de Robbie CX30 violó todas y cada una de las reglas de oro en diferentes modos. En este artículo sólo trataré unas pocas instancias de violaciones de reglas, dejando la discusión más en detalle para futuros artículos y mi próximo libro [NOTA AL PIE: CODEPENDENCIA: Cómo los Usuarios de Computadoras permiten deficientes Interfaces con el Usuario, Angst Press, Nueva York. Este libro presenta una teoría radicalmente nueva con respecto a la relación entre la persona y su máquina. Esencialmente, algunas personas necesitan una interfaz de mala calidad a los fines de evitar ciertos problemas psicológicos no resueltos en sus vidas.]. Lo que haré es destacar esas violaciones que fueron relevantes en este accidente en particular.

4.1 Buscar siempre la coherencia

En la interfaz del usuario de Robbie CX30 hubo muchas incoherencias. Los mensajes de error podían aparecer en casi cualquier color y podían estar acompañados por casi cualquier tipo de efecto musical. Además, los mensajes de error podían aparecer en casi cualquier lugar de la pantalla.

Cuando Bart Matthews vio el mensaje de error de la condición excepcional que ocurrió luego, la cual requería la intervención del operador, es probable que fuera esa la primera vez que veía ese mensaje en especial. Además, el mensaje de error apareció en un cuadro verde, sin ningún efecto de sonido. Este es el único mensaje de error de todo el sistema que aparece en verde y sin ningún tipo de acompañamiento de orquesta.

4.2 Permitir que los usuarios frecuentes utilicen métodos abreviados por teclas

Este principio no aparece de ningún modo en todo el diseño de la interfaz. Por ejemplo, hubiera sido una buena idea permitir que los usuarios frecuentes pudieran ingresar la primera letra de la opción de un submenú o menú en lugar de requerírseles el uso de las teclas del cursor y luego la tecla "enter" para elegir esa opción determinada. El mecanismo de selección de menús de este sistema debe haber provocado al operador bastante fatiga mental.

Es más, debería haberse permitido algún tipo de sistema de tecleado anticipado que permitiera al usuario frecuente ingresar una secuencia de opciones de menú sin tener que esperar a que apareciera realmente el menú en pantalla.

4.3 Ofrecer realimentación[1] de información

En muchos casos el usuario no tiene idea de si el comando que acaba de ingresar se está procesando o no. Este problema se exagera además por las inconsistencias en el diseño de la interfaz con el usuario. En algunos casos al operador se le da una realimentación detallada de lo que el robot está ejecutando. En otros, el sistema permanece misteriosamente silencioso. En general, al usuario se lo lleva a que espere algún tipo de realimentación y por consiguiente se queda confundido cuando ésta no se le da. En la pantalla, no hay una representación visual del robot y su entorno, y la visión que tiene el operador del robot a veces está obstruida.

4.4 Diseñar diálogos que tengan fin

Hay muchos casos en los que una secuencia dada de tecleado representa una idea holística, una tarea completa, pero al operador se lo deja sin el tipo de realimentación que le confirmare que la tarea ha sido en efecto completada. Por ejemplo, hay un diálogo bastante complicado que se necesita cuando se quiere sacar un elemento de un baño de ácido. Sin embargo, luego de completar este diálogo, el usuario es llevado a otro diálogo nuevo, y no relacionado con éste, sin que se le informe que el diálogo anterior ha finalizado.

4.5 Ofrecer manejo simple de los errores

El sistema pareciera estar diseñado para que el usuario se lamentara por cualquier ingreso erróneo. No sólo el sistema permite numerosas oportunidades para el error, sino que cuando un error en realidad ocurre, es probable que no se repita por algún tiempo. Ello se debe a que la interfaz con el usuario hace que recuperarse de un error sea una odisea tediosa, frustrante y a veces enfurecedora. Algunos de los mensajes de error eran directamente ofensivos y condescendientes.

4.6 Permitir deshacer las acciones con facilidad

Como se menciona en el párrafo anterior, la interfaz con el usuario hace muy difícil la tarea de deshacer entradas erróneas. En general, el sistema de menús permite deshacer fácilmente las acciones, pero esta filosofía no alcanza al diseño de los cuadros de diálogo y al manejo de condiciones excepcionales. Desde un punto de vista práctico (opuesto al teórico), la mayoría de las acciones son irreversibles cuando el sistema está en un estado de condición excepcional, y esto ayudó a llegar a la tragedia del robot asesino.

4.7 Promover que uno sea el centro del control

Muchas de las deficiencias tratadas en los párrafos precedentes disminuyeron la sensación de "tener el control". Por ejemplo, no recibir información, no poder concluir las interacciones, no permitir deshacer con facilidad las acciones en el momento en que surgen las excepciones, todas estas cosas actúan para disminuir la sensación de que el usuario posee el control sobre el robot. Hubo muchas características de esta interfaz que hicieron que el operador sintiera que hay un enorme bache entre la consola del operador y el robot en sí, mientras que un buen diseño de interfaz hubiera hecho transparente la interfaz con el usuario y le hubiera dado al operador del robot la sensación de estar en contacto directo con el mismo. En un caso, le ordené al robot mover un elemento desde baño de ácido hasta la cámara de secado y pasaron 20 segundos antes de que el robot pareciera responder. De este modo, no tuve la sensación de estar controlando al robot. Tanto la respuesta demorada del robot como la falta de realimentación en la pantalla de la computadora, me hicieron sentir que el robot era un agente autónomo, la verdad, un sentimiento como mínimo perturbador.

4.8 Reducir la carga de la memoria de corto plazo

Un sistema que se maneja por medio de menús es en general bueno en términos de la carga de memoria que crea en los usuarios. No obstante, hay una gran variación entre implementaciones particulares de sistemas de menú en lo que hace a carga de memoria. La interfaz con el usuario de Robbie CX30 tenía menús muy grandes sin una obvia organización interna. Esto crea una gran carga al operador en términos de memoria y también en términos de tiempos de búsqueda, el tiempo que le lleva al operador ubicar una opción determinada de un menú.

Muchas pantallas de diálogo requerían que el usuario ingresara con el teclado números de partes, nombres de archivos, y otra información. El sistema podría haberse diseñado fácilmente de forma de mostrarle al usuario estos números de parte, etc., sin necesitar que el usuario recordara estas cosas de su propia memoria. Esto incrementaba la carga sobre la memoria del usuario.

Para finalizar, y esto es realmente imperdonable, increíble como puede parecer, ¡no había ninguna instalación de ayuda en línea o sensible al contexto! Si bien he ido a los cursos de entrenamiento ofrecidos por Silicon Techtronics, muchas veces me encontré navegando por los manuales de referencia para poder encontrar la respuesta a aún las más básicas preguntas, tales como:"¿Qué significa esta opción de menú? ¿Qué pasa si selecciono esta opción?"

5. Una reconstrucción de "la tragedia del robot asesino"

Las fotos policiales de la escena del accidente no son nada agradables de ver. La consola del operador estaba salpicada con bastante cantidad de sangre. No obstante, la calidad de las fotos es excepcional y utilizando técnicas de ampliación pude descubrir los siguientes factores de importancia sobre el momento en que se produjo el accidente:

1. La luz NUM LOCK estaba encendida

El teclado IBM contiene un tablero numérico que puede operar de dos modos. Cuando la luz NUM LOCK está encendida, esa parte se comporta como una calculadora. Del otro modo, las teclas pueden usarse para mover el cursor en la pantalla.

2. Había sangre esparcida en el tablero numérico

Las huellas ensangrentadas indican que Bart Matthews estaba usando el tablero numérico en el momento en que fue golpeado y muerto.

3. Se encontraba titilando en verde un mensaje de error

Esto nos dice la situación de error vigente en el momento en que ocurrió la tragedia. El mensaje de error decía "ROBOT DYNAMICS INTEGRITY ERROR-45"[2]

4. Había un manual de referencia que estaba apoyado abierto sobre el área de lectura/escritura de la estación de trabajo

Uno de los cuatro volúmenes del manual de referencia estaba abierto en la página del índice que contenía el ítem "ERRORES/MENSAJES".

5. En la pantalla también había un mensaje que mostraba instrucciones al operador

El mensaje aparecía en amarillo en la parte inferior de la pantalla. En el mensaje se leía . "POR FAVOR INGRESE INMEDIATAMENTE LA SECUENCIA DE COMANDOS PARA CANCELAR EL ERROR DINÁMICO DEL ROBOT!!!"

En base a las evidencias físicas más otras evidencias contenidas en los registros del sistema, y basándose en la naturaleza del error ocurrido (error de integridad de dinámica del robot - 45, el error que estuvo causado por el programa de Randy Samuels), he llegado a la conclusión de que ocurrió la siguiente secuencia de eventos en la fatal mañana de la tragedia del robot asesino:

10:22:30 "ERROR DE INTEGRIDAD DE DINÁMICA DEL ROBOT - 45" aparece en la pantalla. Bart Matthews no lo ve porque no hay efecto de audio o señal sonora tal como ocurre con todas las otras situaciones de error. Además, el mensaje de error aparece en verde, lo que en todos los otros contextos significa que hay un proceso completándose con normalidad.

10:24:00 El robot comienza a moverse lo suficientemente violento como para que Bart Matthews lo note.

10:24:05 Bart Matthews se da cuenta del mensaje de error, no sabe lo que significa. No sabe qué hacer. Intenta con el submenú "cancelación de emergencia", un submenú de uso genérico para apagar el robot. Este involucra SEIS opciones de menú por separado, pero el Sr. Matthews no se da cuenta de que la luz NUM LOCK está encendida. Por ende, las opciones del menú no están ingresando dado que las teclas del cursor operan como teclas de calculadora.

10:24:45 El robot gira del baño de ácido y comienza a arrastrar la consola del operador, con sus brazos dentados batiéndose con gran amplitud. Nadie esperaba que un operador tuviera que huir de un robot descontrolado, así que Bart Matthews queda atrapado en su área de trabajo por el robot que avanza. Más o menos para este momento es que Bart Matthews saca el manual de referencia y empieza a buscar el error ERROR DE INTEGRIDAD DE DINÁMICA DEL ROBOT - 45 en el índice. Ubica con éxito una referencia a mensajes de error en el índice.

10:25:00 El robot ingresa al área del operador. Bart Matthews abandona la búsqueda del procedimiento del operador ante un error de integridad de dinámica del robot. En su lugar, intenta una vez más ingresar la secuencia de "cancelación de emergencia" desde el teclado numérico, momento en que es golpeado.

6. Resumen y Conclusiones

Si bien el módulo de software escrito por Randy Samuels causó en verdad que el robot Robbie CX30 oscilara fuera de control y atacara a su operador humano, un buen diseño de la interfaz hubiera permitido al operador terminar con el comportamiento errático del robot. En base al análisis de la interfaz del usuario del robot llevado a cabo utilizando las ocho reglas de oro de Shneiderman, el experto en diseño de interfaces ha arribado a la conclusión de que el diseñador de la interfaz y no el programador fue la parte más culpable en este desafortunado evento.

7. Referencias

1. Gritty, Horace (1990). The Only User Interface Book You'll Ever Need. Vanity Press, Oshkosh, WI, 212 pag. [El único libro sobre Interfaz del usuario que usted necesitará]
2. Gritty, Horace (1992). What We Can learn from the Killer Robot [Lo que podemos aprender del robot asesino], charla dada en el Simposio Internacional de la Universidad de Silicon Valley sobre Seguridad de Robots e Interfaces con el Usuario, Marzo de 1991. También por aparecer en las Notas de Alumnos de la Universidad de Silicon Valley.
3. Gritty, Horace (se espera para 1993). CODEPENDENCY: How Computer Users Enable Poor User Interfaces, Angst Press, New York [Cómo los usuarios de computadoras permiten interfaces deficientes]
4. Shneiderman, Ben (1987), Designing the User Interface, Addison-Wesley, Reading MA, 448 pag. [Diseño de Interfaces]
5. DOCUMENTO DE REQUERIMIENTOS DEL ROBOT INDUSTRIAL INTELIGENTE Robbie CX 30: versión de Cybernetics Inc., Documento Técnico Nº 91-0023XA, Silicon Techtronics Corporation, Silicon Valley, USA 1245 pag.
6. Foley, J. P., Wallace, V. L., y Chan, P. (1984): The Human Factors of Computer Graphics Interaction Techniques [Los factores humanos de las técnicas de interacción de gráficas de computación] IEEE COMPUTER GRAPHICS AND APPLICATIONS, 4(11) pag. 13-48.

Ingeniero de Software Cuestiona la Autenticidad de las Pruebas de Software del "Robot Asesino"

La Indagación de un Profesor de la Universidad de Silicon Valley Provoca Serios Cuestionamientos Legales y Éticos

Especial para el SENTINEL-OBSERVER DE SILICON VALLEY

Silicon Valley, EEUU
por Mabel Muckraker

El caso del "robot asesino dio un giro significativo ayer cuando un profesor de la Universidad de Silicon Valley presentó un informe en donde cuestiona la autenticidad de las pruebas que fueron hechas por Silicon Techtronics al software del "robot asesino". El Profesor Wesley Silber, Profesor de Ingeniería de Software, dijo en una conferencia de prensa realizada en la universidad que los resultados de las pruebas reflejados en los documentos internos de Silicon Techtronics no concordaban con los resultados de las pruebas obtenidos cuando él y sus colegas ensayaron el software real del robot.

Silicon Valley aún está reaccionando por el anuncio del Profesor Silber, que podría jugar un papel importante en el juicio a Randy Samuels, el programador de Silicon Techtronics que fue acusado por homicidio no premeditado en el ahora infame incidente del "robot asesino". Presionada por su reacción por el informe del Profesor Silber, la Fiscal Jane McMurdock reiteró su confianza en que el jurado encontrará culpable a Randy Samuels. Sin embargo, la Fiscal Jane McMurdock impresionó a los periodistas cuando agregó "pero, esto en verdad promueve la posibilidad de nuevas acusaciones".

Ruth Whiterspoon, la vocero del "Comité de justicia para Randy Samuels", también estuvo exultante cuando habló a este periódico. "McMurdock no puede tener ambas cosas". O el programador es el responsable por esta tragedia o se deberá hacer responsable a la gerencia por ello. Creemos que el informe de Silber exonera a nuestro amigo y colega Randy Samuels."

El gerente Ejecutivo de Silicon Techtronics Michael Waterson hizo la siguiente tibia declaración sobre el informe de Silber:

"Tan pronto se anunció la acusación de Randy Samuels personalmente le pedí a un estimado ingeniero de software, el Dr. Wesley Silber, que llevara a cabo una indagación objetiva sobre los procedimientos de aseguramiento de la calidad en Silicon Techtronics. Como gerente ejecutivo de este proyecto, siempre he insistido en que la calidad es lo primero, a pesar de lo que hayan podido leer en los periódicos.

"Le pedí al profesor Silber que condujera una investigación objetiva de todos los aspectos de aseguramiento de la calidad de Silicon Techtronics. Prometí al Profesor Silber que tendría acceso a toda la información relevante a esta infortunada situación. Le dije en una reunión frente a frente en mi oficina que debía proseguir la investigación hasta su final sin importar a donde terminara, sin importar las implicancias.

"Basándome en la información que yo recibía de mis gerentes, nunca se me ocurrió que hubiera un problema de que los procedimientos de aseguramiento de la calidad fueran ya sea débiles o deliberadamente alterados. Quiero asegurarle al público que la o las personas responsables de esta falta en el aseguramiento de la calidad del software dentro de la División Robótica de Silicon Techtronics serán exhortados a encontrar trabajo en otro lado."

Roberta Matthews, viuda de Bart Matthews, el operador del robot que fue muerto en el incidente, habló telefónicamente desde su casa con el Sentinel-Observer. “Aún quiero ver al Sr. Samuels condenado por lo que le hizo a mi marido. No entiendo de dónde viene toda la conmoción. El hombre que asesinó a mi esposo debería haber probado su propio software!”

El SENTINEL-OBSERVER entrevistó al Profesor Silber justo antes de su conferencia de prensa. En las paredes de su oficina estaban colgados numerosos premios recibidos a raíz de su trabajo en el campo de ingeniería de software y aseguramiento de la calidad del software. Comenzamos la entrevista pidiendo al Profesor Silber que explicara por qué a veces el software no es confiable. Contestó a nuestra pregunta citando la enorme complejidad del software.

“Los grandes programas de computadora son indiscutiblemente los artefactos más complejos creados por la mente humana”, explicó el Profesor Silber sentado frente a un monitor de grandes dimensiones. “En algún momento un programa de computación está en uno de los tantos estados posibles, y hay una imposibilidad práctica de asegurar que el programa se comportará como corresponde en cada uno de esos estados. No tenemos el tiempo suficiente para hacer tal tipo de prueba exhaustiva. De modo que usamos estrategias de prueba o heurísticas que muy probablemente encontrarán los errores o bugs, si es que existe alguno.”

El Profesor Silber ha publicado numerosos artículos sobre ingeniería de software. Estuvo en la primera plana cuando el año pasado publicó su lista de “Aerolíneas a Evitar Si su Vida Dependiera de Ello”. Esa lista enumeraba las aerolíneas de cabotaje que él consideraba irresponsables por su compra de aviones que están controlados casi por completo por software de computación.

Poco tiempo después de los cargos contra Randy Samuels en el caso del “robot asesino”, el gerente Ejecutivo de Silicon Techtronics, Michael Waterson, pidió al Profesor Silber que condujera una revisión objetiva de los procedimientos de aseguramiento de la calidad de Silicon Techtronics. La intención de Waterson era contrarrestar la mala publicidad de su empresa luego de las acusaciones de Samuels.

“El Aseguramiento de la Calidad” se refiere a aquellos métodos que usa un especialista de desarrollo de software para asegurar que el software es confiable, correcto y robusto. Estos métodos se aplican a todo lo largo del ciclo de vida de desarrollo del producto de software. En cada etapa se aplican los métodos de aseguramiento de calidad adecuados. Por ejemplo, cuando un programador escribe código, una medida de aseguramiento de calidad es probar el código confrontándolo en verdad con los datos de prueba. Otro método sería correr programas especiales, llamados analizadores estáticos, confrontándolo con el nuevo código. Un analizador estático es un programa que busca patrones sospechosos en los programas, patrones que podrían indicar un error o bug.

Estas dos formas de aseguramiento de calidad son denominadas pruebas dinámicas y pruebas estáticas, respectivamente.

El software consiste de componentes discretos o unidades que eventualmente se combinan para crear un sistema más grande. Las unidades mismas deben ser probadas, y este proceso de prueba individual de las unidades es llamado prueba unitaria. Cuando las unidades se combinan, se deben probar los subsistemas integrados y este proceso se llama prueba de integración.

El Profesor Silber comentó al SENTINEL-OBSERVER sobre su trabajo en Silicon Techtronics: “Mike [Waterson] me dijo de ir allí [a la compañía] y conducir una revisión de sus procedimientos de prueba de software y de hacer públicos mis hallazgos. Mike

parecía confiado, tal vez debido a lo que le habían dicho sus gerentes, en el sentido de que no encontrarían nada malo en los procedimientos de aseguramiento de calidad de Silicon Techtronics."

Luego de arribar a Silicon Techtronics, el Profesor Silber centró su atención en los procedimientos para ensayo dinámico de software en la compañía.

Ayudado por un grupo de graduados, el Profesor Silber descubrió una discrepancia entre el comportamiento real de la sección del código del programa (escrito por Randy Samuels) que causó que el robot Robbie CX30 matara a su operador, y el comportamiento según se lo registró en la documentación de pruebas de Silicon Techtronics. Este descubrimiento en realidad fue hecho por Sandra Henderson, una estudiante graduada en ingeniería de software que está completando su doctorado con el Profesor Silber. Entrevistamos a la Sra. Henderson en uno de sus laboratorios de computación para egresados en la Universidad de Silicon Valley. "Encontramos un problema en la prueba de la unidad", explicó la Sra. Henderson. "Acá están los resultados de la prueba, que nos dio el Sr. Waterson en Silicon Techtronics, que se supone están hechos para código C [lenguaje de programación] que Randy Samuels escribió y que causó el incidente del robot asesino. Como puede ver, todo está claramente documentado y organizado. Hay dos juegos de prueba: uno basado en una prueba de caja blanca y otro en una prueba de caja negra. Basándonos en nuestros propios estándares para probar software, estos juegos de prueba están bien diseñados, completos y rigurosos. "La prueba de caja negra implica ver la unidad de software (o sus componentes) como una caja negra que tiene comportamientos predecibles de input y output. Si en el juego de prueba el componente demuestra los comportamientos esperados para todos los inputs, entonces pasa la prueba. Los juegos de prueba están diseñados para cubrir todos los comportamientos "interesantes" que una unidad podría mostrar pero sin tener conocimiento alguno sobre la estructura o naturaleza del código en realidad. La prueba de caja blanca implica cubrir todos los pasos posibles a través de la unidad. Así, la prueba de caja blanca se hace con vasto conocimiento de la estructura de la unidad. En la prueba de caja blanca, el juego de prueba debe causar que cada sentencia del programa se ejecute por lo menos una vez de modo que ninguna quede sin ser ejecutada.

Sandra Henderson prosiguió explicando el significado de la prueba de software. "Ni la prueba de caja blanca ni de caja negra "prueban" que un programa esté correcto. No obstante, los probadores de software, tales como los que se emplean en Silicon Techtronics, pueden volverse bastante expertos en el diseño casos de prueba para descubrir nuevos bugs en el software. La actitud apropiada es que una prueba es exitosa cuando se encuentra un bug." Básicamente, el probador le da un juego de especificaciones y hace lo mejor de sí para demostrar que el código que está probando no satisface sus especificaciones", explicó la Sra. Henderson.

La Sra. Henderson luego mostró a este reportero los resultados de las pruebas que ella en verdad obtuvo cuando corrió el código crítico del "robot asesino" usando los juegos de prueba de la compañía, tanto para ensayo de caja blanca como de caja negra. En muchos casos, los resultados registrados en los documentos de prueba de la compañía no fueron los mismos que los generados por el verdadero código del robot asesino.

Durante su entrevista de ayer con el SENTINEL-OBSERVER, el Profesor Silber discutió la discrepancia. "Verá, el software que en verdad fue entregado junto con el robot Robbie CX30 no fue el mismo que el que supuestamente fue probado, ¡por lo menos de acuerdo con estos documentos!. Hemos podido determinar que el verdadero "código asesino", tal como lo llamamos, fue escrito después de que se condujeron supuestamente las pruebas del software. Esto sugiere varias posibilidades. Primero, el proceso de prueba de software, por lo menos para esta parte crítica del software, fue falseado deliberadamente. Todos sabemos que hubo una enorme presión para tener listo a este robot en una fecha

determinada. Otra posibilidad es que hubo una cierta dificultad en la versión de la gerencia en Silicon Techtronics, en cuanto a que el código correcto fue verdaderamente escrito, y probado con éxito, pero en el producto entregado se insertó el código equivocado."

Solicitamos al Profesor Silber que explicara que quería decir con "versión de la gerencia". "En un proyecto dado, un componente dado de software puede tener varias versiones, versión 1, versión 2, etc.

Esto refleja la evolución de ese componente a medida que avanza el proyecto. Se necesita tener algún tipo de mecanismo para tener control de las versiones de los componentes de software en un proyecto tan complejo como este. Tal vez el probador de software probó una versión correcta del código de dinámica del robot, pero en realidad se entregó una versión equivocada del mismo. No obstante, esto trae a colación una pregunta en cuanto a qué pasó con el código correcto."

El Profesor Silber se reclinó en su sillón. "Realmente esto es una gran tragedia. Si el código asesino hubiese sido pasado por el proceso de prueba de un modo honesto, el robot nunca hubiese asesinado a Bart Matthews. Entonces, la pregunta es, ¿qué pasaba en Silicon Techtronics que no permitió una prueba honesta del código crítico?"

El SENTINEL-OBSERVER preguntó al Profesor Silber si estaba de acuerdo con el concepto de que la interfaz del usuario fue la primordial culpable en este caso. "No creo en el argumento que esgrime mi colega, el Profesor Gritty, de que toda la culpabilidad en este caso pertenece al diseñador o diseñadores de la interfaz. Concuerdo con ciertas cosas que dice, pero no con todo. Debo preguntarme a mí mismo si Silicon Techtronics estaba poniendo mucho énfasis en la interfaz del usuario como la última línea de defensa contra el desastre. Esto es, ellos sabían que había un problema, pero pensaron que la interfaz del usuario podría permitirle al operador manejarlo."

El SENTINEL-OBSERVER preguntó entonces al Profesor Silber sobre los cargos que se le hacían en cuanto que nunca debería haber aceptado la designación de Waterson para conducir una investigación objetiva del accidente. Las críticas señalan que la Universidad de Silicon Valley, y en particular el Profesor Silber, tenían muchos intereses comunes con Silicon Techtronics, y de ese modo no podía ser elegido para conducir una investigación objetiva.

"Pienso que mi informe habla por sí mismo," replicó el Profesor Silber, visiblemente molesto por nuestra pregunta. "Ya les he dicho a ustedes los periodistas una y otra vez que no se trató de una investigación gubernamental sino de una interna de la corporación, de modo que creo que Silicon Techtronics tenía el derecho de elegir a quien se le ocurriera. Creo que yo les resultaba una persona con integridad."

Ayer tarde, Sam Reynolds, el gerente de Proyecto del CX30 contrató a una abogada, Valerie Thomas. La Sra. Thomas hizo estas declaraciones en favor de su cliente:

"Mi cliente está escandalizado de que alguien de Silicon Techtronics haya podido engañar al Profesor Silber en lo concerniente a las pruebas de software del robot Robbie CX30. El Sr. Reynolds asegura que el software fue probado y que él y otros sabían muy bien el hecho de que había algo que no funcionaba en el software de dinámica del robot. Sin embargo, el Sr. Ray Johnson, el superior inmediato de mi cliente en Silicon Valley, decidió que el robot fuera entregado a Cybernetics, Inc., basándose en la teoría del Sr. Johnson: "Nada es tan blanco como la nieve".[3]

Conforme a esa teoría, el software estaba casi libre de bugs y por ende podía ser liberado. Según el Sr. Johnson, el riesgo de falla era muy pequeño y el costo por demorar más la entrega del robot era muy alto. "Según mi cliente, el Sr. Johnson creyó que las condiciones del medio ambiente que podría llegar a disparar un comportamiento errático y

violento del robot eran extremadamente improbables de ocurrir. Aún más, el Sr. Johnson creyó que el operador del robot no podría estar en peligro debido a que la interfaz del usuario fue diseñada de modo de permitir al operador detener el robot fijo en sus guías en el caso de un movimiento del robot que comprometiera la vida del operador."

El Sr. Johnson, Jefe de la División Robótica de Silicon Techtronics, no pudo ser ubicado para obtener sus comentarios. Randy Samuels será juzgado el mes entrante en la Corte de Silicon Valley. Cuando se lo contactó por teléfono, Samuels derivó todas las preguntas a su abogado, Alex Allendale.

Allendale tenía esto para decir con respecto a los descubrimientos del Profesor Silber:

"Mi cliente remitió el software en cuestión del modo usual junto con la documentación usual y con la esperanza de que su código fuera probado exhaustivamente. Desconocía hasta el momento de que saliera a la luz el informe del Profesor Silber, que el código involucrado en esta terrible tragedia no había sido probado adecuadamente o que los resultados de prueba pudieran haber sido falsificados.

"El Sr. Samuels nuevamente quiere expresar su gran pesar por este accidente. Él, más que nadie, quiere que se haga justicia en este caso. El Sr. Samuels nuevamente extiende sus más sentidas condolencias a la Sra. Matthews y a sus hijos."

Empleado de Silicon Techtronics Admite Falsificación de las Pruebas del Software

Mensajes Tomados del Correo Electrónico Revelan Nuevos Detalles en el Caso del "Robot Asesino"

Una Asociación de Computadores Científicos Lanza una Investigación sobre Violaciones al Código de Ética

Especial para el SENTINEL-OBSERVER DE Silicon Valley

Silicon Valey, EEUU
por Mabel Muckraker

Cindy Yardley, una probadora de software de Silicon Techtronics, admitió hoy que ella fue la persona que creó las pruebas de software fraudulentas del "robot asesino". Las pruebas fraudulentas fueron reveladas a principios de esta semana por el profesor Wesley Silber de la Universidad de Silicon Valley, con lo que se ha dado en llamar "El Informe Silber".

Se cuestionan los procedimientos de aseguramiento de la calidad que fueron realizados en el código del programa escrito por Randy Samuels, el programa acusado por asesinato no premeditado en el incidente del robot asesino. El Informe Silber afirma que los resultados de las pruebas reflejados en documentos internos de Silicon Techtronics son inconsistentes con respecto a los resultados de las pruebas obtenidos cuando fue probado el verdadero código del robot asesino.

Ayer al mediodía, en un acontecer inesperado, anunció su renuncia al cargo de Jefe de Seguridad de Silicon Techtronics, el Sr. Max Worthington, en una conferencia de prensa que fue transmitida en vivo por CNN y otros informativos.

Worthington sacudió a los periodistas cuando comenzó su conferencia de prensa con el anuncio “Yo soy Marta”.

Worthington describió de este modo sus responsabilidades en Silicon Techtronics: “Básicamente, mi trabajo era proteger a Silicon Techtronics de todos los enemigos, locales y extranjeros. Por extranjeros quiero significar adversarios de otras corporaciones. Mi papel era más que nada de dirección. Aquellos que trabajaban bajo mi supervisión tenían muchas responsabilidades, incluyendo la de proteger la planta en sí, estar alertas por espionaje industrial e incluso sabotaje. También yo era responsable de vigilar a los empleados que pudiesen estar abusando de drogas o que de algún modo estuviesen siendo desleales con Silicon Techtronics.”

Luego Worthington apuntó a una pila de volúmenes que había en una mesa a su izquierda. “Estos volúmenes representan tan solo algunos de los mensajes electrónicos de empleados que yo hice a lo largo de los años para mi superior, el Sr. Waterson. Estas son impresiones de mensajes por E-mail que los empleados de Silicon Techtronics se enviaron entre sí y a personas de otros sitios. Puedo decir con gran certeza que nunca jamás se le dijo a ningún empleado que se hacía este tipo de requisa electrónica. No obstante, creo que la evidencia muestra que algunos empleados sospechaban que esto podía estar pasando.”

Varios periodistas preguntaron a los gritos quién en Silicon Techtronics estaba al tanto de esta requisa.

Worthington respondió. “Nadie sabía de esto a excepción del Sr. Waterson y yo, y uno de mis asistentes que era el responsable de en verdad conducir el monitoreo. Mi asistente producía un informe especial, resumiendo toda la actividad por E-mail de la semana, y ese informe era para que lo viera Waterson y yo solamente. Si se lo solicitaba, mi asistente podía dar un recuento más detallado de las comunicaciones electrónicas”.

Worthington explicó que estaba poniendo a disposición de la prensa las transcripciones del correo electrónico porque quería que saliera a luz toda la verdad sobre Silicon Techtronics y el incidente del robot asesino.

Los mensajes de E-mail entre empleados de Silicon Techtronics en verdad revelaron nuevas facetas del caso. Un mensaje de Cindy Yardley al Jefe de División Robótica, Ray Johnson, indica que ella falsificó a su pedido los resultados de las pruebas. Acá está el texto del mensaje:

```
a:     Ray Johnson
de:    Cindy Yardley
re:    Software de Samuels

Terminé de crear los resultados de las pruebas de software para ese
software problemático, según tu idea de usar una simulación en vez del
software propiamente dicho. Adjunto encontrarás el documento de prueba
modificado, mostrando la simulación exitosa.
¿Le deberíamos decir a Randy sobre esto? Cindy
```

La respuesta de Johnson al mensaje de Yardley sugiere que él sospechaba que el correo electrónico podía no ser seguro:

```
En respuesta a:   Cindy Yardley
de:    Ray Johnson
re:    Software de Samuels

Sabía que podría contar contigo. Estoy seguro de que tu dedicación a
Silicon Techtronics te será pagada con creces. Por favor, en el futuro
```

usa un medio de comunicación más seguro cuando discutimos este tema. Te aseguro que el modo en que manejamos esto fue completamente transparente, pero yo tengo mis enemigos acá mismo en la propia SiliTech.

Ray.

Estas comunicaciones fueron intercambiadas justo unos días antes que se enviara el robot Robbie CX30 a Cybernetics Inc. Este hecho es importante porque las pruebas de software falsificadas no fueron parte de un encubrimiento en el incidente del robot asesino. Estos hechos parecen indicar que el propósito de falsificar las pruebas de software era asegurarse de que el robot Robbie CX30 fuera entregado a Cybernetics, Inc. en la fecha que fue negociada entre Silicon Techtronics y Cybernetics.

Las transcripciones del correo electrónico revelan que hubieron repetidos mensajes de Ray Johnson a diferentes personas en el sentido de que la División Robótica iba a ser cerrada definitivamente si el proyecto Robbie CX30 no era completado en término. En uno de los mensajes, diserta con su líder de proyecto, Sam Reynolds, acerca de la "Teoría Ivory Snow".

a: Sam Reynolds
de: Ray Johnson
re: ¡no seas un perfeccionista!

Sam:

Tú y yo hemos tenido nuestras diferencias, pero debo decirte que personalmente me caes bien. Por favor entiende que todo lo que hago es con el propósito de SALVAGUARDAR TU TRABAJO Y EL TRABAJO DE TODOS EN ESTA DIVISIÓN. Yo te veo a ti y a toda la gente que trabajan para mí en la División Robótica como mi familia.
Waterson fue claro: quiere tener el proyecto del robot completado en término. Y punto. Entonces, no tenemos otro recurso más que el de " Ivory Snow". Sabes lo que quiero decir con eso. No tiene que ser perfecto. La interfaz del usuario es nuestro respaldo si esta versión del software para el robot tiene algunas fallas. El operador del robot va a estar seguro porque podrá cancelar cualquier movimiento del robot en cualquier momento. Concuerdo contigo en cuanto a que los requerimientos no funcionales son en algunas partes demasiado vagos. Lo ideal sería que si estos no fueran tiempos de apuros, cuantificáramos el tiempo que le llevaría al operador detener el robot en un caso de accidente. Sin embargo no podemos renegociar esto ahora. Como tampoco tenemos tiempo para diseñar requerimientos no funcionales nuevos y más precisos. No puedo enfatizar suficientemente de que estos son tiempos de apurarse. A Waterson no le cuesta nada deshacerse de toda la División Robótica. Sus amigos del Wall Street sólo le van a decir ¡Felicitaciones!. Verás, para Waterson nosotros tan sólo somos del montón.

Ray.

En este mensaje Ray Johnson parecería estar menos preocupado por la seguridad de comunicarse por correo electrónico.

El SENTINEL-OBSERVER entrevistó ayer por la tarde a Cindy Yardley en su propia casa. No se pudieron contactar ni a Ray Johnson ni a Sam Reynolds.

La Srta. Yardley estaba notoriamente ofuscada porque sus mensajes por E-mail fueran dados a conocer a la prensa. "De alguna forma me siento aliviada. Sentí una enorme

culpa cuando ese hombre fue muerto por un robot que yo ayudé a producir. Una tremenda culpa."

El SENTINEL-OBSERVER preguntó a la Srta. Yardley si es que ella había hecho una elección ética al acceder a falsear los resultados de las pruebas de software. Respondió con gran emoción. "Nada, pero nada a lo largo de mi experiencia y background me preparó para algo como lo que me pasó. Estudié ciencias de la computación en una universidad de gran nivel y allí me enseñaron sobre pruebas de software, ¡pero jamás me dijeron que alguien con poder sobre mí me pediría generar una prueba falseada!"

"Cuando Johnson me pidió que lo hiciera, me llamó a su oficina, como para mostrarme las trampas del poder; verá, algún día me gustaría estar en un puesto gerencial. Me senté en su oficina y vino directamente y me dijo "Quiero que falsifiques los resultados de las pruebas del software de Samuels. No quiero que Reynolds se entere de nada de esto".

Yardley contuvo las lágrimas. “Me aseguró de que probablemente nadie vería jamás los resultados de las pruebas dado que el robot era perfectamente seguro. Era tan sólo una cuestión interna, un tema de prolijidad, en caso de que alguien de Cybernetics o de un puesto alto dentro de la corporación le diera curiosidad de ver los resultados de las pruebas. Le pregunté si estaba seguro de que el robot era seguro y todo eso y me dijo “¡Es seguro! La interfaz del usuario es nuestra línea de defensa. En alrededor de seis meses podemos enviar una segunda versión del software del robot y para ese entonces este problema de Samuels estará resuelto.”

Yardley se reclinó en su asiento como si lo que dijera a continuación necesitara de un énfasis especial. “Entonces me dijo que si yo no falsificaba las pruebas, todos los de la División Robótica perderían sus trabajos. Sobre esa base decidí falsificar las pruebas, trataba de proteger mi trabajo y el de mis compañeros.”

La Srta. Yardley está al presente cursando un grado de Maestría en Administración de Empresas en la Universidad de Silicon Valley.

Luego el SENTINEL-OBSERVER pregunto a la Srta. Yardley si aún sentía que había tomado una decisión ética, en vista de la muerte de Bart Matthews. “Creo que fui manipulada por Ray Johnson. Él me dijo que el robot era seguro”.

Otra revelación, contenida el las transcripciones del correo electrónico dadas a conocer, fue el hecho de que Randy Samuels hurtó parte del software que usó en el proyecto del robot asesino. Este hecho se reveló en un mensaje que Samuels envió a Yardley cuando ella probó por primera vez su software y dio resultados erróneos:

```
En respuesta a:   Cindy Yardley
de:    Randy Samuels
re:    maldito si lo sé

Por mi vida, no puedo entender qué es lo que anda mal en esta función
balancear_brazo(). Verifiqué la fórmula de la dinámica del robot una y
otra vez y pareciera estar implementada correctamente. Como sabes, la
función balancear_brazo() invoca a 14 funciones diferentes. A cinco de
ellas las tomé tal cual del paquete estadístico PACKSTAT 1-2-3. ¡Por
favor no se lo digas a nadie! No son éstas los que causarían el
problema, ¿ o sí?

Randy.
```

Los expertos le dijeron al SENTINEL-OBSERVER que tomar software de paquetes comerciales de software como el PACKSTAT 1-2-3 es una violación a la ley. El software tal como el inmensamente popular PACKSTAT 1-2-3 está protegido por el mismo copyright que protege al material impreso.

Mike Waterson, Presidente Ejecutivo de Silicon Techtronics, emitió una enojosa declaración porque Max Worthington había dado a conocer las transcripciones del correo electrónico “confidencial”. Las declaraciones de Waterson decían, en parte, que “Yo le pedí a nuestros abogados que intervinieran en este tema. Consideramos que esas transcripciones son propiedad exclusiva de Silicon Techtronics. Nuestra intención es efectuar cargos ya sea civiles o criminales contra el Sr. Worthington.”

Como reacción a lo ocurrido ayer en el caso del robot asesino, la ACM o Association for Computer Machinery anunció su intención de investigar si algún miembro de la ACM de Silicon

Techtronics ha violado el Código de Ética de la ACM. La ACM es una asociación internacional de computadores científicos con 85.000 miembros.

La Dra. Turina Babbage, presidente de la ACM, hizo una declaración en la Conferencia de Ciencias de la Computación de ACM que se lleva a cabo cada invierno y que esta temporada se hará en Duluth, Minnesota.

Un extracto de las declaraciones de la Dra. Babbage sigue a continuación:

"Todos los miembros de la ACM están ligados por el Código de Ética y Conducta Profesional de la ACM [NOTA AL PIE: Un borrador de este código fue dado a conocer en Comunicaciones de la ACM, Mayo 1992. Por favor nótese que las declaraciones hechas por la ficticia Dra. Babbage contienen citas del verdadero código de ACM] Este código establece, en parte, que los miembros de ACM tienen el imperativo moral de contribuir con el bienestar de la sociedad y los hombres, evitar daños a terceros, ser honestos y confiables, dar crédito adecuado a la propiedad intelectual, acceder a los recursos de comunicación y de computación sólo cuando así lo estén autorizados, respetar la privacidad de terceros y honrar la confidencialidad.

Más allá de eso, existen responsabilidades profesionales tales como la obligación de cumplir los contratos, acuerdos y responsabilidades asignadas, y de dar evaluaciones profundas y completas de los sistemas de computación y de sus impactos, poniendo especial énfasis en los riesgos potenciales.

Varias de las personas involucradas en el caso del robot asesino son miembros de la ACM y hay causas para creer que han incurrido en violación del código de ética de nuestra asociación. Por lo tanto, estoy solicitando al directorio de la ACM designar una Fuerza de Tareas para investigar a los miembros de la ACM que puedan haber violado groseramente el código.

No tomamos este paso a la ligera. Esta sanción ha sido aplicada sólo rara vez, pero el incidente del robot asesino no sólo ha costado una vida humana, sino que ha causado mucho daño a la reputación de la profesión de computación."

La Revista Dominical del SENTINEL-OBSERVER - Una Conversación con el Dr. Harry Yoder

por
Robert Franklin

Harry Yoder es una figura muy bien conocida en el campo universitario de Silicon Valley. El profesor de Tecnología y Ética de la Computación de Samuel Southerland ha escrito numerosos artículos y textos sobre ética y el impacto social de las computadoras. Sus clases son muy famosas, y muchos de sus cursos están completos mucho antes de que finalice el período de inscripción. El Dr. Yoder ha recibido su Doctorado en ingeniería eléctrica del Instituto de Tecnología de Georgia en 1958. En 1976 recibió un grado en "Maestría en Divinidad" del Harvard Divinity School. En 1983 recibió un Master en Ciencias de la Computación de la Universidad de Washington. Ingresó en la facultad de la Universidad de Silicon Valley en 1988.

Entrevisté al Dr. Yoder en su oficina del campus. Mi intención era obtener su reacción con respecto al caso del robot asesino y “leer su pensamiento” sobre los temas éticos que involucra el caso.

SENTINEL-OBSERVER : Ir de la ingeniería eléctrica al estudio de religión parece un gran salto.

Yoder: Yo era un ingeniero electricista por profesión, pero todos los seres humanos tienen una vida interior, ¿no lo cree así?

SENTINEL-OBSERVER : Sí

Yoder: De qué se trata su vida interior?

SENTINEL-OBSERVER : Trata de hacer lo correcto. También se trata de lograr la excelencia en lo que hago. ¿Es eso lo que lo llevó a la Escuela de Divinidad de Harvard? ¿Usted quería clarificar su vida interior?

Yoder: Sucedían muchas cosas en la Escuela de la Divinidad, y muchas de ellas eran muy poderosas. Sin embargo, más que nada yo quería comprender la diferencia entre lo que estaba bien y lo que estaba mal.

SENTINEL-OBSERVER : ¿Y qué hay de Dios?

Yoder: Sí, estudié mi propia religión cristiana y a la mayoría de las religiones del mundo, y todas ellas tenían cosas interesantes que decir acerca de Dios. No obstante, cuando yo discuto sobre ética en un foro tal como este, que es secular, o cuando discuto ética en mis cursos de ética en la computación, no coloco a esa discusión en un contexto religioso. Pienso que la fe religiosa puede ayudarle a una persona a abrazar la ética, pero por otra parte, todos sabemos que ciertas personalidades notorias que se han autopronunciado religiosas han sido altamente no éticas. De este modo, cuando yo discuto sobre ética de la computación, el punto de partida no es la religión, sino más bien un acuerdo común entre mis estudiantes y yo de que queremos ser gente ética, que el luchar por la excelencia ética es una tarea humana que vale la pena. Por lo menos, lo que no queremos es herir a otros, no queremos mentir, robar, hacer trampas, asesinar, etc.

SENTINEL-OBSERVER : ¿Quién es el responsable de la muerte de Bart Matthews?

Yoder: Por favor discúlpeme si lo remito nuevamente a la escuela de la Divinidad de Harvard, pero creo que uno de mis profesores de allí tiene la respuesta correcta a su pregunta. Este profesor era un hombre mayor, tal vez de setenta años, de la Europa Oriental, un rabino. Este rabino dijo que de acuerdo al Talmud, una tradición antigua de la ley judía, si se derrama sangre inocente en un pueblo, entonces los líderes de ese pueblo deben ir a los límites del mismo y realizar un acto de penitencia. Esto es además de la justicia que se le aplicará a la persona o personas que cometieron el asesinato.

SENTINEL-OBSERVER : Ese es un concepto interesante.

Yoder: ¡Y uno de verdad! Un pueblo, una ciudad, una corporación son sistemas en que la parte está ligada al todo y el todo a la parte.

SENTINEL-OBSERVER : Usted quiere decir que los líderes de Silicon Techtronics, tales como Mike Waterson y Ray Johnson, deberían haber asumido la responsabilidad por este incidente desde el vamos. Además, tal vez otros individuos, como ser Randy Samuels y Cindy Yardley, comparten una carga especial de responsabilidad.

Yoder: Sí, responsabilidad, no culpabilidad. La culpabilidad es un concepto legal y la culpabilidad o la inocencia de las partes involucradas, sean ya en lo criminal o lo civil, será decidida en la corte. Estimo que una persona es responsable por la muerte de Bart Matthews si su acción ha ayudado a causar el incidente - es una cuestión de causalidad, independiente de los juicios éticos y legales.
Las cuestiones de responsabilidad podrían serle de interés a los ingenieros de software y gerentes, quienes tal vez querrían analizar qué es lo que anduvo mal, de modo de evitar que similares problemas ocurrieran en el futuro.

Mucho de lo que salió de los medios con respecto a este caso indica que Silicon Techtronics era una organización enferma. Esa enfermedad creó el accidente. ¿Quién creó la enfermedad? La gerencia creó a esa enfermedad, pero también los empleados que no tomaron las decisiones éticas correctas contribuyeron con la misma.

Tanto Randy Samuels como Cindy Yardley eran recién egresados. Se graduaron en ciencias de la computación y su primera experiencia en el mundo laboral fue en Silicon Techtronics. Uno debería preguntarse si recibieron alguna enseñanza sobre ética. Relacionado a esto está la cuestión de si alguno de ellos tenía con anterioridad experiencia en trabajos en grupo. En el momento en que se los asignó al desarrollo del robot asesino, ¿ellos vieron la necesidad de ser personas éticas? ¿Vieron que el éxito como profesionales requiere de un comportamiento ético? Hay mucho más para ser un científico en computación o un ingeniero de software que tan sólo la habilidad y el conocimiento de la técnica.

SENTINEL-OBSERVER : Sé con seguridad que ninguno de los dos tomó cursos sobre ética o ética de la computación.

Yoder: Lo sospechaba. Veamos a Randy Samuels. Basándome en lo que leí en su periódico y en otros lados, era básicamente de los del tipo “hacker”. Amaba la computación y la programación. Comenzó a programar en los primeros años de la secundaria y continuó a lo largo de toda la carrera universitaria. El punto importante es que Samuels aún era un “hacker” cuando entró en Silicon Techtronics y ellos le permitieron que él siguiera siendo así.

Estoy usando el término “hacker” de un modo un tanto peyorativo y tal vez esto no sea justo. El punto que estoy tratando de remarcar es que Samuels nunca maduró más allá de su angosto enfoque como hacker. En Silicon Techtronics, Samuels aún mantuvo esta actitud en lo que hacía a sus funciones de programador, la misma que tenía cuando estaba en la secundaria. Su percepción de la vida y de sus responsabilidades no creció. Él no maduró. No hay evidencia de que tratara de desarrollarse y convertirse en una persona ética.

SENTINEL-OBSERVER : una dificultad, en lo que hace a enseñar ética, es que en general los estudiantes no les gusta que se les diga “esto está bien y aquello está mal”.

Yoder: Los alumnos necesitan entender que el tocar temas de ética es parte de ser computadores científicos o ingenieros de software profesionales.

Una cosa que me ha fascinado acerca de la situación de Silicon Techtronics es que a veces es difícil ver los límites entre lo legal, lo técnico y lo ético. Los temas técnicos involucran temas de gerencia y de computación. He llegado a la conclusión de que este desvanecimiento de los límites resulta del hecho de que la industria de software aún se

encuentra en pañales. Los temas éticos surgen abruptamente en parte porque hay una ausencia de lineamientos técnicos y legales.

En particular, no existen prácticas normalizadas para desarrollar y probar software. Hay estándares, pero no lo son realmente. Una broma muy común entre los computadores científicos es que lo bueno de los estándares es que hay muchos para elegir.

Ante la ausencia de prácticas normalizadas aceptadas universalmente para ingeniería de software, surgen muchos juicios de valor, probablemente más que para cualquier otra forma de producción. Por ejemplo, en el caso del robot asesino, hubo una controversia con respecto al uso del modelo de cascada versus el de prototipo. Debido a que no había un proceso de desarrollo de software
estandarizado, esto se transformó en una controversia, y los temas éticos surgen por el modo en que se resuelve la controversia. Usted recordará que el modelo de cascada fue elegido no por sus méritos sino porque el gerente de proyecto tenía experiencia en éste.

SENTINEL-OBSERVER : ¿Usted cree que Cindy Yardley actuó éticamente?

Yoder: Al principio su argumento parece poderoso: ella, efectivamente, mintió, para así salvar los puestos de trabajo de sus compañeros, y por supuesto, el de ella. Pero, ¿siempre es correcto mentir, para crear una falsedad, en un marco profesional?

Un libro que he usado en mis cursos de ética de computación es el "Ethical Decision Making and Information Technology" (Toma de Decisión Ética y Tecnología de la Información) de Kallman y Grillo [NOTA AL PIE: Este texto es un texto real y está publicado por McGraw-Hill]. En este libro se dan algunos de los principios y teorías que están detrás de la toma ética de decisiones. Yo uso este y otros libros para ayudar a que los alumnos desarrollen sus apreciaciones sobre la naturaleza de dilemas éticos y toma ética de decisiones.

Kallman y Grillo presentan un método para la toma de decisión ética y parte de su método consiste en el uso de cinco pruebas: la prueba de la mamá: ¿le diría Ud. a su mamá lo que hizo?; la prueba de la TV: ¿le diría Ud. a una audiencia nacional de TV lo que hizo?; la prueba del olfato: ¿lo que Ud. hizo tiene mal olor?; la prueba de ponerse en los zapatos del otro: ¿le gustaría que el otro le haga lo que Ud. hizo?; y la prueba del mercado: ¿sería su acción una buena estrategia de venta?

Lo que hizo Yardley reprobó todas estas pruebas, pienso que casi todos concuerdan conmigo. Por ejemplo, ¿pueden imaginar a Silicon Techtronics usando una campaña publicitaria que diga algo como?:

"En Silicon Techtronics el software que usted recibirá de nosotros está libre de bugs, porque aún cuando haya uno, distorsionaremos los resultados de las pruebas para esconderlo, usted nunca se enterará.. **"La ignorancia es la felicidad"**

Esto demuestra que el altruismo aparente no es un indicador suficiente de un comportamiento ético. Uno podría preguntarse qué otros motivos no declarados tenía la Srta. Yardley. ¿Podría ser que la ambición personal la llevara a aceptar la explicación que le dio Ray Johnson y su afirmación de que el robot era seguro?

SENTINEL-OBSERVER : ¿Existe alguna fuente de guía ética para gente que se ve confrontada con un dilema ético?

Yoder: Algunas empresas dan guía ética, en la forma de políticas de la corporación, y existe un documento así en Silicon Techtronics, o por lo menos así me dijeron. Yo no lo vi. Un empleado también podría remitirse a los lineamientos éticos que proveen sociedades profesionales, tales como la ACM. Además, el o la empleada podría leer sobre el tema para obtener una mejor percepción de la toma ética de decisiones. Por supuesto que uno siempre debe consultar con su propia conciencia y con sus convicciones más profundas.

SENTINEL-OBSERVER : ¿Usted cree que Randy Samuels actuó éticamente? Yoder: Robar software en el modo que lo hizo es tanto ilegal como no ético.

Pienso que el punto más importante con Randy Samuels nunca fue discutido en los medios de prensa. Honestamente dudo que Samuels tuviera el conocimiento necesario para su puesto. Este tipo de conocimiento se lo llama conocimiento de la especialidad. Samuels tenía conocimiento de computación y programación, pero no tenía un sólido conocimiento de física, en especial de la mecánica clásica. Su falta de conocimiento en el dominio de la aplicación fue una causa directa del horrible accidente. Si alguien con conocimientos de matemáticas, estadísticas y física hubiera programado al robot en lugar de Samuels, probablemente hoy Bart Matthews estuviera vivo. No tengo dudas de ello. Samuels malentendió la fórmula física porque no entendió su significado e importancia en la aplicación en el robot. Puede ser que la gerencia sea en parte responsable por la situación. Puede que Samuels les haya dicho acerca de sus limitaciones y la gerencia habrá dicho. “Y bueno, qué importa”

Samuels tenía dificultades en trabajar en equipo, hacer revisiones en conjunto, y programar sin egoísmo. ¿Es posible que estuviera intentando esconder su falta de experiencia en el área?

SENTINEL-OBSERVER : ¿Cree que Ray Johnson actuó éticamente?

Yoder: ¡Este tema del “Ivory Snow”! El problema con la teoría del “Ivory Snow” es que es tan solo eso, una teoría. Si fuera más que una teoría, o sea una metodología real para mantener la probabilidad de la falla dentro de límites estadísticamente determinados, como lo que se llama “ingeniería de software en sala limpia” [cleanroom software engineering], entonces habría menos culpabilidad en ese punto.

Basándome en la información que dispongo, la teoría de Ivory Snow fue tan sólo una racionalización para sacarse de encima a software fallado y entregarlo en término al cliente. Esta teoría sólo es válida, ética y profesionalmente, si al cliente se le informa de los bugs de los que se tiene conocimiento, o de impurezas, utilizando la jerga. En el caso de Silicon Techtronics, la teoría Ivory Snow funcionó así: ¡sabemos que no es puro, pero el cliente cree que sí lo es!

Desde luego, presionar a Cindy Yardley como lo hizo Ray Johnson tampoco es ético. ¿El creía en lo que le dijo a la Srita. Yardley, es decir, que el robot era seguro, o fue eso una mentira del momento? Si el creía que el robot era seguro, entonces ¿por qué cubrirse con pruebas falsas? Si la interfaz con el usuario era tan importante como la última línea de defensa, entonces ¿por qué evitar pruebas más rigurosas de la interfaz?

SENTINEL-OBSERVER : ¿Qué piensa de Mike Waterson en todo esto?

Yoder: Si Johnson es el padre de la teoría Ivory Snow, Waterson el es abuelo. Su exigencia de que el robot estuviera completado para una fecha determinada o de lo contrario "rodarían cabezas", puede haber causado que Johnson formulara la teoría Ivory Snow. Verá, es evidente que Johnson pensaba que era imposible entregar a Cybernetics Inc. el robot CX 30 para una fecha determinada, a menos que el software fuera con bugs.

En muchos sentidos pienso que Waterson actuó sin ética e irresponsablemente. Pone a Sam Reynolds a cargo del proyecto del robot, cuando aún él, Reynolds, carecía de experiencia con robots e interfaces con el usuario modernas, Reynolds rechazó la idea de desarrollar un prototipo, lo que podría haber permitido el desarrollo de una mejor interfaz.

Waterson creó una atmósfera opresiva entre sus empleados, que en sí mismo es falto de ética. No sólo amenazó con despedir a todos los de la División Robótica si el robot no se terminaba a tiempo, sino que hurgó en comunicaciones por correo electrónico privadas de toda la corporación, un derecho controvertido que algunas empresas alegan tener.

Mi creencia personal es que este tipo de investigación es falto de ética. La naturaleza del E-mail es algo así como un híbrido de correspondencia común y conversación telefónica. Monitorear o espiar la correspondencia ajena está considerado no ético, tal como lo es interferir un teléfono. Por cierto, estas actividades también son ilegales bajo la mayoría de las circunstancias. O sea, creo que monitorear a los empleados del modo que lo hizo Waterson es un abuso de poder. SENTINEL-OBSERVER: ¿Usted cree que en esto el fiscal tiene un caso? Yoder: ¿Contra Randy Samuels?

SENTINEL-OBSERVER : Sí. Yoder: Lo dudo, a menos que ella tenga información que hasta ahora no se ha hecho pública. El asesinato no premeditado, a mi entender, implica un tipo de acto irresponsable y negligente, que causa la muerte de un tercero. ¿Se aplica esta descripción a Samuels? Pienso que la mejor apuesta de la fiscal es hacer hincapié en su falta de conocimiento en el área de aplicación, si puede mostrarse que Samuels se involucró deliberadamente en un fraude.

La semana pasada leí que el 79% de la gente está a favor de la absolución. La gente es proclive a acusar a la compañía y a sus gerentes. La otra noche, uno de los noticieros dijo, "Samuels no es un asesino, es un producto de lo que lo rodea".

SENTINEL-OBSERVER : Podría nuevamente decir su posición sobre el tema de la responsabilidad final en el caso del robot asesino?

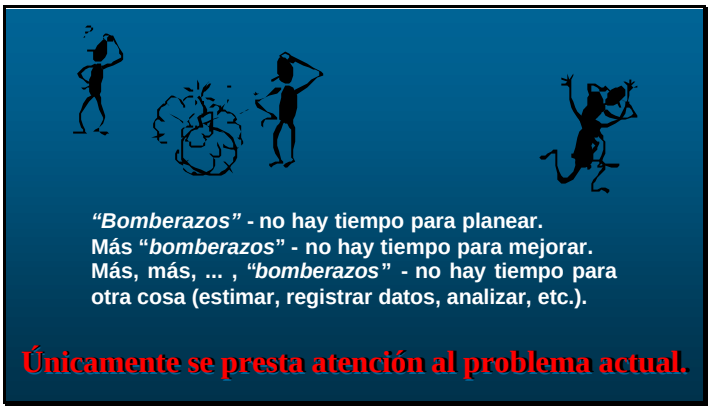
Yoder: En mi mente, el tema de la responsabilidad del individuo versus la responsabilidad de la corporación, es un tema muy importante. La corporación creó un entorno en el que podían ocurrir este tipo de accidentes. Aún así, los individuos, dentro de ese sistema, actuaron sin ética e irresponsablemente, y fueron los que de hecho causaron el accidente. Una compañía puede crear un entorno que saca a flote lo peor de sus empleados, pero cada empleado también puede contribuir a empeorar ese ambiente corporativo. Este es un lazo cerrado que se alimenta a sí mismo, un sistema en el sentido clásico. Entonces, hay cierta responsabilidad de la corporación y cierta responsabilidad de los individuos en el caso del robot asesino.

SENTINEL-OBSERVER : Muchas gracias Profesor Yoder.

¿Te has preguntado por qué no alcanzas tus metas?

Es una pregunta en general para los alumnos para descubrir los posibles motivos, causas que dan pie a que fallemos en alcanzar los propósitos, ¿por qué no terminas tus proyectos de software a tiempo, en el costo y con la calidad establecida?, o cualquier otro proyecto, aprobar un examen, ganar una competencia deportiva, ahorrar, etc.

¿¡Falta Tiempo?!



Muchas de las veces, la respuesta inmediata es "es que no tengo tiempo", pero, ¿cómo saben que efectivamente no tienen tiempo?, ¿en qué consumen el tiempo que tienen?, si tienen problemas, ¿cómo los atienden?, ¿descuidan otras tareas, actividades para solucionar lo "urgente", lo "prioritario"?

¿Por qué la industria del software no puede alcanzar sus objetivos?

Y si extrapoláramos las ideas, respuestas a nivel personal a la industria del software, ¿serían las mismas?, probablemente sí, pues al fin y al cabo quien produce el software son personas, claro que habrá algunos otros factores propios de una organización ajenos a lo personal.

Evolución

Debido a la penetración constante y creciente del software a ámbitos de la vida cotidiana, los propios usuarios son los que exigen más y más, sistemas de software más sofisticados en menos tiempo, estas exigencias también de negocio incrementan la complejidad que se multiplica al combinarse con cuestiones de mercado. Ahora, los avances tecnológicos, el ritmo con que se liberan, están dictados por esta demanda.

Reflexiona

¿Me dejaría operar con software creado por mí?

Ejercicio

Tómate algunos minutos y reflexiona la siguiente pregunta, piensa en todo el software y la manera en que lo has creado. Anota tus reflexiones, el porqué sí y el porqué no, comparte estas reflexiones con tus compañeros.

Por ejemplo, si estuvieras en el quirófano y vieras que el aparato para realizarte una intervención en tus ojos es controlado por aquel software que alguna vez desarrollaste.

De esta respuesta se puede poner un antecedente para dejar en claro que realmente hace falta aplicar mecanismos para realizar un mejor trabajo. Si la respuesta general es que sí se dejarían operar, preguntar entonces, confiarían en el software y hardware con el cual interactuaría el software que ellos crearan

(sistemas operativos, administradores de bases de datos, memorias RAM, unidades de procesamiento, etc.).

El trabajo del Ingeniero de Software

Todo trabajo (producto/servicio) de un Ingeniero de Software se debe ajustar a:

- Producir/Brindar productos/servicios de calidad.
- Hacerlos dentro del costo.
- Completarlos en el tiempo.

En esencia esto es el trabajo de cualquier profesional, el punto aquí es hacer una analogía con los objetivos de la ingeniería de software, es decir, el trabajo del Ingeniero de Software es cumplir y hacer cumplir los objetivos de la Ingeniería de Software.

Qué vs. Cómo

El cómo hacer es tan importante como el qué se va a hacer.

Es decir, debemos también manejar y controlar la manera en que hacemos las cosas y no únicamente pensar en el resultado final. Es a través de la ejecución adecuada de los procesos que llegaremos al producto deseado.

Producto

Si nos centramos en el producto (el qué), podemos no darnos cuenta de cómo mejorarlo.

En cierta parte de la historia automotriz, algunas compañías se centraban en corregir las fallas en cada uno de los automóviles, es decir, pensaban más en el qué, en el producto, corregían las fallas en cada uno de los productos.

Si nos centramos en el *proceso* (el cómo), podemos:

- Predecir la repetición de la salida.
- Conocer las tendencias del proyecto.
- Controlar las características de calidad en el producto.

Llegó el tiempo, en que una de ellas, pensó más en el porqué salían defectuosos sus productos, e investigó la manera en que los producía, encontrando fallas en su cómo eran construidos, así corrigió esa manera de desarrollar sus automóviles y la falla no se volvió a presentar. Mejorar los procesos tendrá un impacto positivo en el producto resultante, si no en funcionalidades, al menos en el costo.

Podemos definir a un proceso como una la combinación de un conjunto de actividades (procedimientos) que deben ser realizadas por un conjunto de personas que tienen las habilidades para ello y que además utilizarán herramientas que les permitirán realizar su trabajo en forma más eficiente. Así, podemos definir al proceso de desarrollo de software como el conjunto de actividades que tienen que ejecutarse para producir el software, las cuales serán ejecutadas y controladas por personas con el conocimiento y las habilidades para ello (los Ingenieros de Software), que utilizarán herramientas que les permitan realizar su trabajo más rápido y poder comunicar de mejor manera la intención de cada uno de los productos de trabajo que realizan de forma tal que se entienda que se han cumplido los requisitos establecidos, incluyendo los de calidad, en cada uno de estos productos.

Ejercicio

Define un Proceso:

1. Describe el proceso que sigues para bañarte.
2. Describe el proceso para hacer un pastel.

El ejercicio tiene como objetivo que los estudiantes se den cuenta que definir implica un esfuerzo.

- Qué se tiene que hacer.
- En qué orden se tiene que hacer.
- Quién lo tiene que hacer.
- Qué hay que tomar en cuenta antes de realizar las actividades.
- Qué recursos se necesitan para realizar las actividades.
- Cuáles son los resultados de las actividades.
- Cómo saber que ya se terminó cada actividad.
- Qué saber y qué hacer para poder mejorar la forma de realizar el trabajo y que estos puntos deben estar presentes en la definición de procesos.

Creatividad vs. Disciplina

La creatividad emerge y se mejora con disciplina.

Los procesos nos sirven para no pensar en las cuestiones mecánicas y tener más tiempo para la parte creativa en la construcción de las soluciones.

Entonces, si existen procesos que nos dicen qué hacer, cómo hacerlo y cuándo hacerlo, se pudiera pensar que el proceso de desarrollo de software mecaniza la creación de productos de software haciendo de esta actividad algo "fría". No, el hecho de tener procesos para desarrollar software nos permite concentrarnos en lo importante, en las cuestiones críticas dejando que los procesos nos guíen a cada paso.

Utilizando procesos, ¿podemos cambiar?

El desarrollo de software es una actividad muy intensa, *personas en forma individual* siguen escribiéndolo.

Cualquier mejora en la eficiencia o productividad *de estas personas*, resultará en ganancia en los proyectos y en la industria en general.

Al final, quienes escriben el software son personas, en proyectos de muchos ingenieros de software a cada uno le corresponde una parte, la calidad del trabajo final depende de la suma de las habilidades y conocimiento de cada uno de los miembros del equipo de desarrollo, así que cualquier ganancia en modificar la forma de desarrollar software, en los procesos que se siguen a nivel de equipo o personal, tendrá una repercusión en la calidad del producto.

¿Qué hace falta?

Necesitamos una *disciplina* de Ingeniería de Software que provea a las personas que desarrollan los componentes, un método para planear, dar seguimiento y administrar efectivamente los defectos. Más aún, que les permita aprender de sus éxitos y fracasos.

¿De dónde tomar el conjunto de procesos que nos ayude?, de eso se trata precisamente este curso, de aprender técnicas que aplicadas en conjunto permitirán potenciar las habilidades y conocimiento en el desarrollo de software.

Hay que distinguir entre:

- La realización del producto.
- La administración del proyecto.

Planeando el trabajo.- qué hacer y cómo hacerlo.

Realizando el trabajo.- según lo planeado.

Buscando la calidad.- dando seguimiento, evaluando y mejorando el producto y el proceso.

Entonces, para resumir, una cosa es el producto que vamos a desarrollar y otra cosa muy distinta el cómo lo vamos a desarrollar, necesitamos un conjunto de procesos que nos permitan planear el trabajo, realizar el trabajo y al mismo tiempo nos ayude en el camino de mejorar la forma que tenemos de trabajar. Ese es el conjunto de habilidades y conocimiento que la metodología a estudiar nos presenta y que necesitamos aprender y aplicar si en verdad queremos convertirnos en Ingenieros de Software.

PSP®, será parte de ese proceso

PSP® es un método basado en procesos, es un método de mejora continua a nivel personal.

Con PSP® tenemos la oportunidad de aprender a usar y ver los beneficios de trabajar de una manera *disciplinada* y orientada a procesos.

PSP® fue desarrollado en el Instituto de Ingeniería de Software de la Universidad de Carnegie Mellon por Watts Humphrey.

Sólo para poner en antecedentes las técnicas están basadas en la metodología desarrollado por Watts Humphrey, aunque es importante decir que por sí mismas las técnicas ya existían, lo que hizo Humphrey fue ponerlas dentro de un marco de trabajo aplicativo, describiendo guías de la aplicación paso a paso.

1.2. ¿Qué es PSP?

PSP® comprende:

PSP = Planeación* + Diseño + Revisiones de Diseño* + Codificación + Revisiones de Código* + Pruebas

PSP® enseña, re-enseña desde una perspectiva sistémica muchos conceptos de Ingeniería de Software, en este curso veremos partes de esos procesos, sentaremos las bases para aplicar en su totalidad PSP®.

Tradicionalmente al desarrollar software a nivel personal (y muchas veces a nivel de organización), las actividades o fases más comunes, si no las únicas son las de Codificación y Pruebas. Después, quizá se realice diseño, pero ¿planeación y revisiones de código y diseño?

En este curso abarcaremos parte de la planeación, cuestiones de codificación y administración de defectos.

PSP® es un marco de trabajo basado en ***procesos***.

Enseña y ayuda a los Ingenieros de Software a:

- Planear y dar seguimiento.
- Cumplir sus compromisos.
- Crear productos de calidad.
- Resistir a presiones irracionales
- Identificar *sus áreas de mejora*.

Lo anterior con el registro y análisis de los datos emanados de *los propios procesos de los Ingenieros de Software*.

Con crear productos de calidad se refiere a que se tienda al concepto de "cero defectos". Resistir presiones irracionales es negociar los tiempos de desarrollo con base a datos históricos y estadísticas que demuestren la tendencia de la productividad de los ingenieros de software, para que éstos tengan herramientas con las cuales demostrar que los tiempos que proponen son los correctos.

PSP® está basado en la mejora continua, y lo demuestra al pedir realizar análisis de los indicadores de rendimiento que se registran y de analizar la manera en que se está trabajando para proponer nuevas formas, más económicas en tiempo y costo, que incrementen la productividad.

Ejemplos en otras disciplinas

Otros profesionales, como los químicos, cirujanos, pilotos aviadores, músicos, demuestran competencias básicas antes de realizar aún los procedimientos más simples.

¿Por qué los Ingenieros de Software no?

Con la aplicación sistemática de las técnicas presentadas en este curso podremos obtener la competencia necesaria.

¿Qué tiene que hacer un piloto para demostrar que está listo para hacerse responsable de una aeronave?

¿Qué tiene que demostrar un cirujano para poder realizar operaciones?

¿Qué esperaríamos que demuestre un Ingeniero de Software para confiarle la construcción de productos de software?

Que demuestre que sus productos corren sin problemas, que son fáciles de usar, que hacen lo que deben de hacer.

¿Qué utilizaremos?

Formas.- Un conjunto de plantillas en donde registraremos diversos indicadores de la manera en que construimos nuestros productos de software.

Table C16 Time Recording Log

Student _____ Date _____
Instructor _____ Program # _____

Date	Start	Stop	Interruption Time	Delta Time	Phase	Comments

PSP® está basado en un conjunto de formas, guías y descripción de procesos. En el curso utilizaremos algunas de estas formas y explicaremos y utilizaremos los procesos, con variaciones, de varias las técnicas presentadas en PSP®.

Las formas te ayudan a evitar el desperdiciar el tiempo pensando el formato y lo que debe llevar. Pero, las formas buenas son difíciles de construir, por lo que deben ser revisadas periódicamente para ver que cumplan con las necesidades del momento.

1.2.1 Principios en los que se basa PSP

1. La calidad de los sistemas de software depende de la calidad de sus peores componentes.
2. La calidad de los sistemas de software depende de la calidad de sus peores componentes.
3. El ingeniero de software debe conocer su rendimiento, medir, dar seguimiento y analizar su trabajo y aprender de las variaciones de su rendimiento e incorporar estas lecciones a sus prácticas personales.

¡Una cadena es tan fuerte como su eslabón más débil!

¿Para qué nos sirve PSP®?

1. Conocer nuestro rendimiento: al medir nuestro trabajo, reconocer que funciona mejor y aprender a como repetirlo y mejorarlo.
2. Entender las variaciones: lo que es repetible y lo que podemos aprender de las variaciones.
3. Incorporar estas lecciones en una creciente documentación de prácticas personales.

Para dejar de ser únicamente programadores y convertirnos en verdaderos Ingenieros de Software

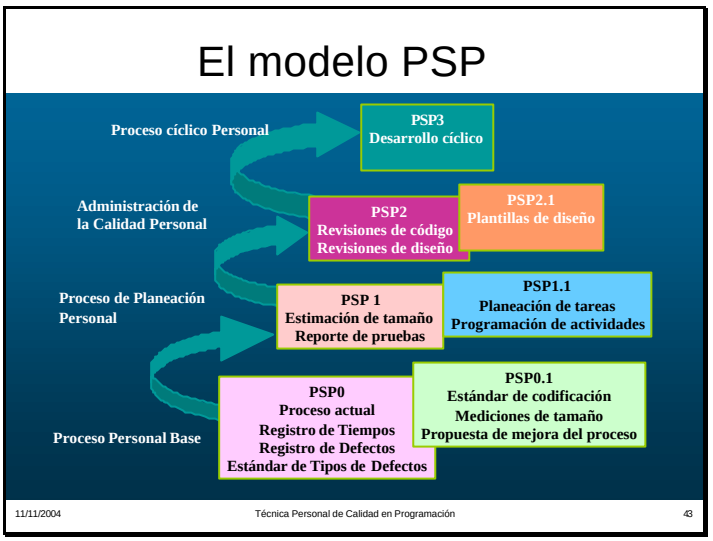
Ejercicio

Del siguiente artículo realiza un resumen:

– Ferguson, Pat, Watts S. Humphrey, Soheil Khajenoori, Susan Macke, and Annette Matvya, "Introducing the Personal Software Process: Three Industry Case Studies," IEEE Computer, Vol. 30, No. 5, May 1997, pp. 24-31.

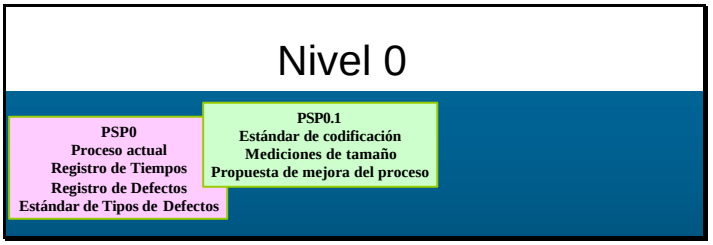
Este artículo describe en términos generales lo que es PSP®. Si el profesor no ha tomado los cursos de PSP® o no está capacitado, es recomendable que lea también este artículo.

1.2.2. El modelo PSP



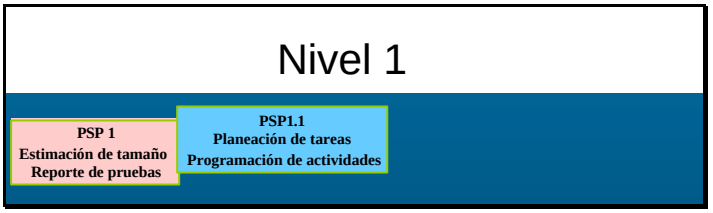
Este modelo está explicado en las siguientes láminas. Aquí hay que mencionar que está descrito por niveles más para cuestiones de aprendizaje que para su aplicación, el beneficio completo de PSP® está basado en la aplicación total de sus conceptos.

Nivel 0



Registro de las métricas básicas del proceso actual de desarrollo del programador, dichas métricas es el tiempo y los defectos con base a un estándar de clasificación de los mismos, de igual forma en este nivel se introduce la aplicación de estándares de programación y registros de medición del tamaño de los programas generados.

Nivel 1



Enseña al programador a estimar el tamaño y el tiempo que ocupará en cada una de las fase de implementación de sus programas; esta estimación está basada en métodos estadísticos e información histórica que el mismo programador ha generado a lo largo de la aplicación del nivel 0 en varios programas.

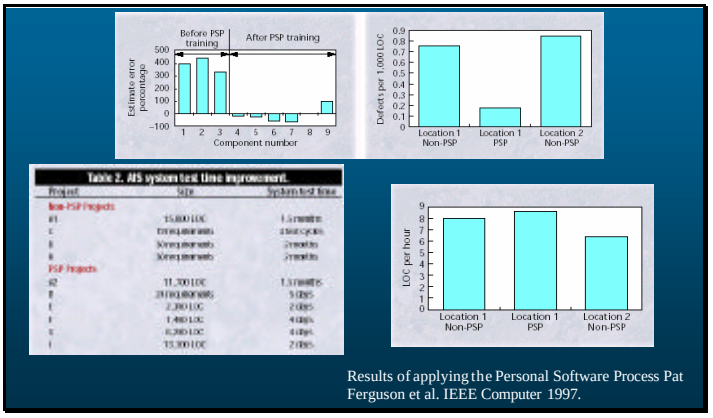
Nivel 2

Las revisiones permiten verificar el aumento de la calidad de los productos generados por el programador y minimizar el número de defectos inyectados en su diseño y en su programación, esto permite que el número de defectos en las fases de pruebas y entrega al cliente disminuyan significativamente y que por lo tanto el tiempo de detección y corrección de defectos se minimicen, de igual forma en este nivel se incorporan formatos para realizar el diseño de los programas en forma estándar.

Nivel 3

Se incorporan técnicas de desarrollo cíclico a fin de permitir utilizar los procedimientos definidos por el Personal Software Process a proyectos de mayor tamaño y de mayor duración.

Algunos resultados obtenidos



La interpretación de estos y otros resultados pueden leerse del artículo señalado. Se les puede pedir a los alumnos que lean y hagan un resumen.

¿Es PSP® la solución?

PSP® no es la panacea a todos los problemas de la Ingeniería de Software. Todo es posible de ser perfeccionado, pero hay que empezar por algo y PSP® es una buena opción.

Además, sus técnicas las podemos aplicar a otras actividades de nuestras vidas.

Existen muchos seguidores como detractores de PSP®. Sin embargo como se señala, es un buen principio, un muy buen principio para mejorar la construcción de productos de software.

Muchas de las técnicas que vamos a estudiar las podemos aplicar en otros quehaceres de nuestra vida, y dan también buenos resultados.

El inicio hacia nuestra mejora en el desarrollo de productos y servicios de software.

Poniendo las bases, este es el primer nivel de PSP®, aquí estudiaremos la manera en que invertimos el tiempo que tenemos para trabajar. Sentaremos las bases para la mejora. Haciendo una analogía, cuando nos sentimos enfermos y vamos al médico, ¿qué es lo primero que nos hace el médico?

Nos realiza una exploración, nos manda a realizar unos análisis. Esto es porque necesita poner un punto de referencia para que después de que nos dé el tratamiento pueda comparar si el tratamiento está resultando o no.

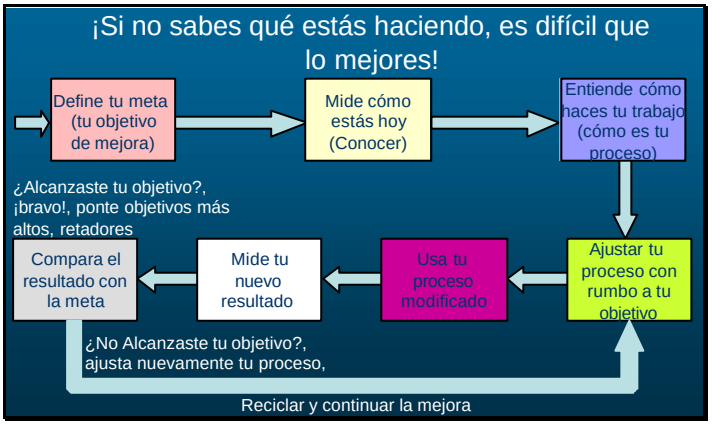
Esto mismo haremos, primero conoceremos nuestro estado en el desarrollo de software para después, siguiendo los procedimientos descritos saber si nos funcionan o no.

Reflexiona

1. ¿Cómo te preparas para aprobar un examen?
2. ¿Cómo se prepara un equipo deportivo para ganar una competencia?
3. ¿Cómo le haces para pasar de un nivel a otro en un juego de video?
4. ¿Cuánto tiempo se invierte en la preparación?

El propósito es hacer reflexionar a los alumnos para que determinen los procesos que siguen en las actividades descritas, cuánto tiempo emplean en alcanzar el objetivo

1.2.2.1 El ciclo de Mejora Continua



Con las láminas anteriores se comenzó a poner las bases para darnos cuenta de qué hacemos y cómo lo hacemos, esto es el punto de partida para alcanzar los objetivos que nos tracemos. Primero definimos a dónde queremos llegar, después medimos cómo estamos y/o cómo está la competencia, trabajamos en entender cómo hacemos las cosas, si no está funcionando o encontramos mejores maneras de hacerlas, modificamos nuestro cómo hacer, lo probamos, medimos el resultado de esta nueva forma de trabajar y si no funciona, realizamos las modificaciones necesarias para conseguir los objetivos iniciales. Si funciona, nos ponemos metas más altas y comenzamos el ciclo.

La base de la mejora personal

El primer proceso que desarrollaremos será el que te permita la recolección de datos sobre el tiempo que inviertes en cada tarea de software.

Estos datos te ayudarán a analizar en qué inviertes tu tiempo, en dónde inviertes más tiempo, es decir, en dónde te tardas más.

Y quizá, primero, que te des cuentas de qué es lo que haces y así podrás decidir si vale o no la pena invertir tanto tiempo en ello.

Ejercicio 1

Realízate las siguientes preguntas y registra la siguiente información, piensa en un día normal de escuela o trabajo:

1. ¿Cuánto tiempo dormí?
2. ¿Cuánto tiempo estudié/trabajé?
3. ¿Cuánto tiempo me llevó trasladarme de un lugar a otro (casa-escuela/trabajo-casa)?

Ejercicio 2

´

Ahora piensa en lo siguiente:

1. ¿Cuántas veces te levantaste durante la noche (para ir al baño, tomar agua, despertaste por pesadillas, etc.)?
2. En el trabajo/escuela, ¿cuántas veces fuiste al baño, a tomar agua?, ¿cuánto tiempo empleaste para tomar tu almuerzo, hablando por teléfono, platicar con amigos, leyendo el periódico?

Resultados

Del tiempo que registraste en la pregunta 1 del ejercicio 1, réstale el tiempo que empleaste en las actividades de la pregunta 1 del ejercicio 2. Haz lo mismo con la pregunta 2 de ambos ejercicios.

Ahora tienes un panorama más claro de cuanto tiempo empleas en algunas tareas diarias. ¿Realmente trabajas/estudias lo que pensaste inicialmente?

La situación más probable es que los alumnos se sorprendan de la cantidad (mínima), del tiempo efectivo que realmente le dedican a los estudios.

Nuestra bitácora

Utilizando la siguiente plantilla, registra las actividades de 3 a 5 días consecutivos (dormir, ver tele, transportación, comer, hacer tareas, juntas, hablar por teléfono, ir al baño, etc.).

Fecha	Actividad	Hora inicio (en minutos)	Hora finalización (en minutos)	Tiempo total empleado (en minutos)

Este ejercicio permitirá comenzar a registrar datos, crear el hábito de registrar la información que estamos generando. En el archivo anexo, Forma de registro de tiempos contiene la plantilla completa y la explicación de su uso.

Administrar nuestro tiempo

Para poder cambiar, necesitamos primero, conocer cómo estamos. Por ello el ejercicio de la bitácora de registro de actividades.

Recuerda es tu tiempo, qué haces y cómo lo inviertes es tu responsabilidad y va en función de tus necesidades.

- ¿En qué invertí mi tiempo?
- ¿Qué dejé de hacer?, en su lugar ¿qué hice?, ¿por qué no las hice?
- ¿Qué me consume más tiempo?
- ¿En qué invierto menos tiempo cuando debiera invertirle más?

Después de que haya concluido el tiempo asignado para la tarea de la bitácora, el profesor debe revisar que esté bien llenada y no haya discrepancias en la información (omisiones e inconsistencias). El profesor puede pedir ahora que los alumnos hagan un análisis, síntesis de las actividades principales que realizan y el tiempo total que le dedican por día y semana.

El tiempo es uno de los recursos más importantes, hay que tener en cuenta que para que podamos sacarle el mayor provecho, éste tiene que ser y estar organizado.

Una mala organización nos hará perder mucho tiempo.

Antes de actuar hay que planear y programar cómo vamos a distribuir el tiempo, ya que éste es limitado.

Debemos administrar nuestro tiempo de la manera más efectiva para cumplir nuestros compromisos.

Si no tenemos orden, si no sabemos qué hacer y cuándo hacer cada cosa el caos cunde. Es importante que de ahora en adelante sepamos qué tenemos que hacer con anterioridad para organizarnos mejor. **LOS COMPROMISOS ADQUIRIDOS DEBEN SER CUMPLIDOS.**

Elementos que nos harán perder tiempo

- Mala organización (mala o falta de planificación).
- Exceso de compromisos.
- Intrusiones (llamadas telefónicas, reuniones mal planeadas o a destiempo).
- Falta de delegación.

Pedir a los alumnos que hagan un nuevo análisis y descubran los motivos por los cuales no están cumpliendo sus objetivos, qué les impide o distrae de realizar sus compromisos.

Actividades Preactivas y Reactivas

Para aprovechar al máximo el tiempo debemos clasificar las actividades en:

- Preactivas: aquellas que están programadas (reuniones, tareas, exámenes).
- Reactivas: las que no están planeadas.

¡Debemos minimizar el tener actividades reactivas!

Del análisis realizado, qué porcentaje de las actividades son reactivas y cuántas preactivas, ¿qué pueden hacer para que las actividades preactivas sean las que predominen?

Para un trabajo más eficaz:

- Tener organizado y limpio el lugar de trabajo.
- Cuidar el aspecto personal.
- Cada cosa en su momento.
- Tener una agenda diaria (tener planeado lo de hoy).
- Cumplir los compromisos.
- Lista priorizada de los pendientes.

Sugerencias para mejorar la organización del tiempo

1. Poner horarios realistas a nuestras actividades.
2. Objetivos cortos y largo plazo.
3. Tener un horario diario y semanal lo más preciso posible.
4. No planear hacer muchas cosas al mismo tiempo.
5. Replanear cuando sea necesario.
6. Ser constante en la aplicación de lo planeado.
7. Realizar las mismas tareas en los mismos momentos.
8. Reordenar las actividades para que se lleven a cabo en los momentos que tengamos mejor rendimiento.
9. No dejar para mañana lo que se puede hacer hoy.

10. Utilizar la curva de trabajo.
11. Tomar breves descansos.

Ser una persona organizada significa ser eficaz; saber organizarse es un **hábito** que hemos de lograr, aunque cueste un poco de trabajo.

Con tu bitácora, analiza los datos, trata de justificar cada actividad (¿por qué, para qué la realizo, necesita todo ese tiempo, qué me distrajo mientras la realizaba?).

Comienza a planear tus actividades diarias y semanales, sigue llevando la bitácora y realizando los análisis para mejorar la organización de tu tiempo.

¿En la Ingeniería de Software me sirve todo lo anterior?

La idea es relacionar todo la experiencia de la vida cotidiana al desarrollo de software, contestar las preguntas de cómo aplicarían lo aprendido a la manera en que desarrollan el software (los alumnos).

Sí, por supuesto, durante el desarrollo de un producto de software el Ingeniero de Software realiza muchas tareas, tiene interrupciones (llamadas de los clientes por ejemplo), adquiere compromisos de entrega, etc.

¡Tiene que cumplir con su trabajo en tiempo!

En Resumen

La base lógica de la administración del tiempo es la siguiente:

Se empleará la misma cantidad del tiempo y de la misma manera esta semana como se hizo la semana pasada.

Por supuesto que hay excepciones. Por ejemplo, durante el periodo de exámenes, podría no atenderse al mismo número de clases por estar estudiando. Otro ejemplo, el caso de la presencia de un huracán.

Del segundo párrafo se despende la justificación del ejercicio, ¿qué tanto tiempo efectivo tengo disponible para dedicárselo a las tareas productivas?, ¿cuáles son los períodos disponibles para trabajar?

Para realizar planes realistas, se le tiene que dar seguimiento a la manera en que se emplea el tiempo.

La gente recuerda algunas cosas y olvida otras. Por ejemplo, la cantidad de tiempo empleado para hacer la tarea; mientras que el tiempo empleado en *relajación* y comidas, podría ser más de lo pensado.

Para saber a donde se va el tiempo, se necesita mantener un registro preciso.

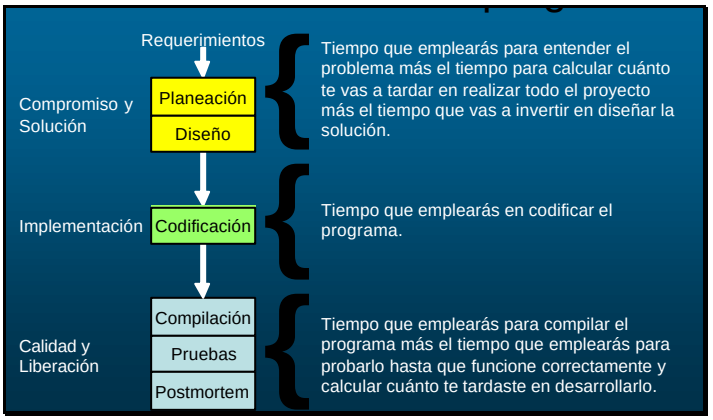
Para saber qué hacemos debemos

1. Categorizar las actividades.
2. Registrar el tiempo empleado en cada actividad.
3. Registrar el tiempo de manera estandarizada.
4. Guardar los datos en un lugar conveniente.

¡Cuando la gente dice que está trabajando duro, lo que realmente quiere decir es que está trabajando más horas (no que son más productivos)!

Con categorizar las actividades nos referimos a enmarcar dentro de algo más grande a un conjunto de actividades más pequeñas. Así por ejemplo estar codificando diversas rutinas, cada una de ellas debe ser registrada para saber cuánto tiempo nos tomó codificarlas, sin embargo, nos interesará saber el tiempo total de la codificación, por lo que la acción de estar codificando la enmarcamos en la fase de Implementación.

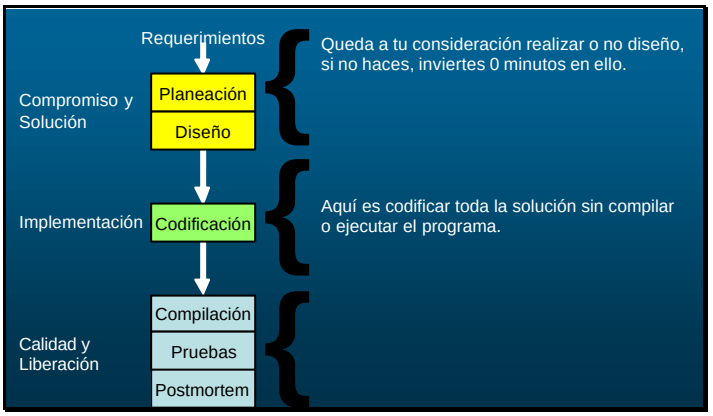
El proceso que seguiremos para desarrollar nuestro 1er programa



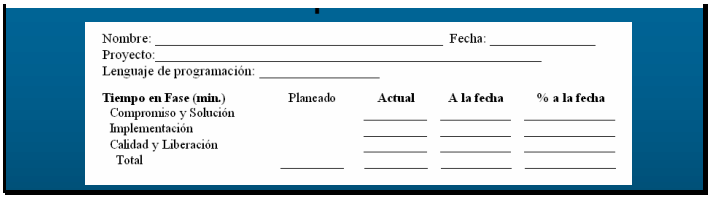
El proceso de desarrollo de software tradicionalmente se ha descrito por las actividades marcadas en los modelos de ciclo de vida del software, obtención de requisitos, análisis, diseño, codificación, pruebas, liberación y mantenimiento. Sin embargo, existe otro conjunto de procesos que también deben estar presentes para asegurar el éxito en le proyecto de desarrollo de software, muchos de estos procesos sólo tienen sentido cuando se trabaja en equipo. Aquí, describiremos

nuestro proceso de desarrollo de software en tres fases, como muestra la gráfica. Esto se hace para simplificar el registro de los datos en este curso introductorio.

Hacer énfasis que la etapa de codificación implica codificar toda la solución del problema, sin hacer ninguna compilación o ejecución del programa. En el momento en que se compile o ejecute por primera vez el programa (esté finalizado o no), se pasa a la etapa nombrada Calidad y Liberación.



Forma de Registro de Compromisos



Registra tu nombre y la fecha en que inicias la solución del problema, el nombre del proyecto y el lenguaje de programación que utilizarás.

En la columna Planeado fila Total, registra el total del tiempo que crees vas a tardar en desarrollar el programa.

En la columna Actual, registra en cada fila el tiempo que tardaste en cada fase.

En la columna A la fecha, la suma de todos tus tiempos por fase hasta ahora (en tu primer programa esta columna y la columna Actual tendrán los mismos datos).

En la columna % a la fecha, el porcentaje del tiempo empleado en cada fase respecto al total.

Forma de Registro de Actividades

Utiliza esta forma como si fuera la forma de la bitácora que utilizaste anteriormente.

Registra el nombre de tu proyecto y tu nombre.

En la columna Actividad registra cada actividad que realices en el desarrollo del programa.

En la columna Fase, la fase en la cual realizas la actividad.

En la columna Inicio y Fin, la hora en que comienzas y finalizas la actividad.

En la columna Tiempo empleado, el total de tiempo empleado en la actividad.

En la columna Comentarios, cualquier comentario pertinente referente a la actividad.

En la columna Fecha, la fecha en que realizas la actividad.

Interrupciones

Una interrupción es cualquier cosa que nos distraiga de la actividad productiva que estemos realizando. Así por ejemplo, supongamos que estamos implementando una función y suena nuestro teléfono, el contestar el teléfono es una nueva actividad que nos distrae de la actividad de codificación.

Debemos registrar las interrupciones para saber en qué hemos estado invirtiendo el tiempo y cuanto tiempo nos toman las actividades ajenas al desarrollo del proyecto.

En la forma de registro de actividades, utiliza un renglón por cada actividad productiva y por cada interrupción.

Ejemplo de llenado de la Forma de Registro de Actividades

Laura Daniela inicia el desarrollo de su proyecto a las 8:33 AM, finaliza la fase de compromiso y solución a las 9:47 AM. A las 9:50 se levanta a tomar un vaso de agua y regresa a las 10:00 AM para continuar con la implementación de la solución, a las 10:25 suena su teléfono, es un cliente preguntándole sobre un proyecto pasado, cuelga el teléfono a las 10:38, continúa su trabajo y finaliza la implementación a la 1:54 PM.

A las 2:00 PM toma su almuerzo y regresa a trabajar a las 2:36. A las 3:30 ha finalizado la fase de calidad y liberación.

Fecha	Actividad	Fase	Inicio	Fin	Tiempo empleado (minutos)	Comentarios
3/Jul/2004	Entendiendo el problema	CS	8:33	8:49	16	
	Planeando	CS	8:50	9:00	10	
	Diseñando	CS	9:01	9:47	46	
	Tomar agua		9:50	9:59	9	
	Implementando altas de usuarios	I	10	10:25	25	
	Atendiendo teléfono, cliente X	I	10:26	10:38	12	
	Implementando altas de usuarios	I	10:39	13:54	195	
	Lunch		14:00	14:35	35	

Fecha	Actividad	Fase	Inicio	Fin	Tiempo empleado (minutos)	Comentarios
3/Jul/2004	Compilando	CL	2:36	2:56	20	Mala inserción de datos por mala definición de campos
	Probando	CL	2:57	3:25	28	
	Registro de datos finales	CL	3:26	3:30	4	

Laura Daniela empleó 72 minutos en la etapa de compromiso y solución, 320 minutos en la etapa de implementación y 52 minutos en la etapa de calidad y liberación.

En total empleó 444 minutos en desarrollar el proyecto.

Ejercicio de Programación 1

Escribe un programa en el que se proporcione una palabra como entrada y se determine el número posible de palabras a formar. Al igual, lista cada palabra posible.

Por ejemplo, se da la palabra MAR y es posible formar 5 palabras más (con esas 3 letras se forman 6 palabras, la que sirve como entrada y las otras 5). Tales palabras son: MRA, RAM, RMA, AMR y ARM. Cada nueva palabra debe ir en su propia línea.

Utiliza la forma de registro de compromisos y de actividades.

Orden de Entrega de los Documentos

- Forma de Registro de Compromisos.
- Forma de Registro de Actividades.
- Código Fuente del Programa.
- Pantallas de la Interfase Gráfica.
- Pantallas de los Resultados.

Este es el orden en que debe ser entregada la tarea de programación.

Fuentes de Referencia Bibliográfica

- **Watts Humphrey**, *Introduction to the Personal Software Process*, 1996, Addison-Wesley Professional.
- **Watts Humphrey**, *A Discipline for Software Engineering*, 1995, Addison-Wesley Professional.
- **Hayes, Will**, *"The Personal Software Process: An Empirical Study of the Impact of PSP on Individual Engineers",* CMU/SEI-97-TR-001.
- **Humphrey, W.S.**, *"Using a Defined and Measured Personal Software Process",* May 1996, IEEE Software.
- **Ferguson, Pat, Watts S. Humphrey, Soheil Khajenoori, Susan Macke, and Annette Matvya**, *"Introducing the Personal Software Process: Three Industry Case Studies*," Vol. 30, No. 5, May 1997, IEEE Computer, pp. 24-31.

2. El Tamaño del Producto

Introducción

Una vez entendido lo que hacemos y cuanto tiempo nos toma llevarlo, nos preocuparemos ahora por mejorar nuestra productividad, es decir, comenzaremos a entender qué tan productivos somos al desarrollar productos de software. Para esto introduciremos el concepto de *tamaño del producto*, el cual relacionaremos con el tiempo que invertimos en desarrollarlo, de tal manera que obtengamos información que nos permita realizar mejores estimaciones cuando aprendamos a planear nuestro trabajo.

Debido a que la dimensión de los proyectos que estamos atacando queda a nivel personal, relacionaremos el tamaño del producto final a las tres fases que describimos en la unidad 1, Compromiso y Solución, Implementación, y Calidad y Liberación. Mediremos únicamente el tamaño del código y relacionaremos el tiempo invertido a las tres fases.

Existen diversas técnicas para contabilizar (medir) el código, cada una con sus detractores y promovedores, no existe un consenso sobre qué unidad de medida utilizar, lo importante es medir, y revisar continuamente nuestros datos históricos para hacerlos pertinentes a la tecnología y tipos de proyectos que realicemos.

Objetivo General

Al término de esta unidad el alumno será capaz de:

- Describir el concepto de productividad al desarrollar productos de software.

Objetivos Específicos

Al término de esta unidad el alumno será capaz de:

- Describir técnicas para contabilizar el tamaño del código.
- Describir el concepto de estándar de codificación y de conteo de código.

2.1. ¿Por qué preocuparnos por el tamaño del producto?

Existen dos razones principales:

1) Para conocer nuestra productividad.

2) Para poder realizar planes adecuados.

La productividad se refiere a la cantidad de códigos podemos escribir en una unidad de tiempo, código que cumpla con su cometido, es decir, que cumpla con los requerimientos.

Otra de las razones por las que hay que saber manejar el tamaño de los productos es para poder realizar planes adecuados, ya que al conocer el tamaño de cada uno de los diferentes subproductos que componen el producto final nos será más fácil saber el avance real que llevamos.

Hacer recapitulación sobre el hecho de que en la unidad 1 se comenzó a registrar el tiempo que nos está tomando realizar los proyectos, distinguiendo algunas fases de desarrollo.

También comentar sobre que conocer de antemano el tamaño de los productos finales servirá para que se tomen diversas acciones respecto a por ejemplo, saber cuantos CD´s serán requeridos para distribuir el software (sabiendo el tamaño final del producto), cuanto costará la impresión del manual de usuario (sabiendo el número de páginas que contendrá, etc.).

2.1.1. Tamaño del Producto

En el desarrollo de un proyecto generamos diversos productos:

- Requerimientos.
- Diseño.
- Código.
- Casos de Prueba.
- Manuales de Usuario.

Al iniciar cualquiera de estos u otros productos, ¿sabemos de qué tamaño resultará el esfuerzo para crearlos?

Aquí es importante analizar el concepto de esfuerzo, el cual está definido por la cantidad de trabajo a realizar por unidad de tiempo. Esta cantidad de trabajo es lo inicialmente llamaremos el tamaño del producto.

El tamaño del Producto en la Planeación

El proceso de planeación comienza con la estimación del tamaño del trabajo a realizar, de esta manera se descubre la cantidad de trabajo que se requiere. Para poder estimar el tamaño, se requiere de un mecanismo consistente y repetible.

Se está amplificando la justificación de medir el tamaño del producto para realizar mejores planes. El decir estimar, significa que debemos de saber con anterioridad a la creación de los productos cuál va es su tamaño, además de que esta predicción se debe realizar de una manera sistemática.

El Estimar el Tamaño del Proyecto es el Primer Paso para la Planeación.

¿Qué unidades de medida debemos usar?

Los indicadores deben ser:

- Útiles para la planeación.
- Precisos.
- Factibles de ser automatizados.

Dentro de un proyecto se pueden medir únicamente 3 cosas: Los Procesos, Los Productos y Los Recursos.

A cada uno de estos se les puede medir: El Tamaño, El Tiempo, El Costo, La Calidad y Los Recursos Críticos.

La dificultad radica en saber cómo, qué unidad de medida específica debemos utilizar para cada proceso, producto y recurso en particular.

Uno de los puntos principales a tomar en cuenta cuando selecciones qué unidad de medida utilizar es que esta medida de tamaño debe estar directamente relacionada al costo de desarrollo.

Se debe dejar en claro que el motivo por el cual nos preocupa este punto es que a través del tamaño podremos estimar el tiempo en que desarrollaremos el producto, de esta manera, también obtendremos el costo del mismo.

Una vez seleccionada una unidad de medida precisa, se requieren medios automáticos, económicos y exactos para obtenerla.

Ejemplo

Consideremos un ejemplo sencillo:

Tarea: "Leer 6 ejemplares de *El mundo del Fútbol*"

Al finalizar esta tarea hemos recolectado los siguientes datos

Ejemplar	Tiempo de lectura	Páginas	Minutos/página
1	50	25	2
2	30	18	1.66
3	45	22	2.04
4	124	80	1.55
5	77	37	2.08
6	40	22	1.81
Total	366	204	
Promedio	61	34	1.79

Si el nuevo ejemplar consta de 33 páginas, ¿cuánto tiempo te tomará leerlo?

Hacer notar las variaciones que existen entre la cantidad de páginas leídas y el tiempo que toma leer el total de cada ejemplar.

El profesor debe iniciar un intercambio de ideas preguntándoles a los alumnos el porqué consideran que se dan estas variaciones.

Precauciones

El tamaño por sí sólo no es un indicador adecuado. La ***complejidad*** del trabajo también es importante.

Quizá las variaciones se hayan debido a:

- Tipo de información.
- Presentación de la información.
- Etc.

El profesor aquí al explicar diversos ejemplos de complejidades ajenas al software, debe finalizar haciendo la pregunta: ¿Qué consideran que puede ser complejo en la construcción de los proyectos de software?, ¿y específicamente en la codificación?

2.1.1.1. Tamaño y Esfuerzo

Encontrar el indicador adecuado para el tamaño y el esfuerzo es complicado. Debemos de describir la forma de conteo del tamaño del producto para que sea:

- **Fácil de comunicar**.- Si al utilizar el mismo método, ¿podrán otros saber precisamente qué ha sido medido, incluido y excluido?
- **Repetible**.- ¿Podrá alguien repetir la medición y obtener el mismo resultado?

2.1.1.2. Tamaño del Código

En este curso, nos preocuparemos únicamente por el tamaño del código.

Hay muchas posibles unidades de medida:

- Líneas de código (LOC).
- Puntos Funcionales.

- Páginas, pantallas, scripts, reportes, etc.

Uno pudiera pensar que tomar el programa final como unidad de medida pudiera ser suficiente, sin embargo, esto sería una unidad de medida demasiada grande.

Para poder realizar una mejor planeación hace falta descomponer los programas en algo más sencillo, más pequeño, que permita estimaciones más exactas y poder tener un mejor control durante el seguimiento de los proyectos.

Un enfoque típico para medir el tamaño del código son las LOC.

Hay que tener cuidado al utilizarlas, ya que su medición puede variar por diferentes motivos:

- Complejidad del programa.
- Experiencia del Ingeniero de Software.
- Lenguaje de programación.
- Estándares utilizados en la codificación.

Aquí el profesor puede iniciar una reflexión pidiendo a los alumnos que registren en algún lugar la cantidad de código que creen han escrito a lo largo de sus vidas.

Las LOC no son adecuadas en todos los casos, por ejemplo, no son adecuadas para:

- Menús.
- Reportes.
- Pantallas.
- Documentos.

2.1.1.2.1. Tipos de código

Existen diversos tipos de código:

- Código de pruebas.
- Código de apoyo.

- Código del producto.
- Código generado automáticamente.

Cada uno de ellos debe ser contabilizado por separado.

Un punto importante es que las el código generado automáticamente no debe ser contabilizado como parte de la productividad, pero sí como parte del tamaño total del producto. El código generado automáticamente es aquel que es traducido del diseño o la especificación por alguna herramienta CASE.

El decir que sea contabilizado por separado significa que se debe mantener un registro del tamaño del código por cada uno de los tipos mencionados.

El código de pruebas es aquel que escribimos cuando dentro del programa que estamos desarrollando incluimos código que servirá para probarlo, generalmente este código es borrado o comentado antes de liberar el programa.

Cuando el programa que estamos desarrollando tiene que interactuar con otros módulos que no existen, entonces creamos un prototipo de estos módulos, el código de estos prototipos es lo que llamamos código de apoyo.

2.1.1.2.2. Formas de Contar el Código

Existen dos formas en que podemos contabilizar el código:

- Por líneas físicas.
- Por líneas lógicas.

El conteo por líneas físicas es aquel en donde cada línea (finalizada por un retorno de carro) es contabilizada.

Las líneas lógicas son aquellas que representan un agrupamiento lógico de líneas, que en su conjunto tienen un significado.

Ejemplos de Líneas Físicas

1)

```
if A < B then
        C:= C + 1
```

```
else
        C:= C – 1
4 LOC
```

2)

```
        if A < B then C:= C + 1
        else C:= C – 1
        2 LOC
```

Cada una de las líneas terminadas por un retorno de carro es contabilizada. Hacer hincapié sobre el hecho de la forma en que se escribe el código, lo cual puede afectar la productividad. Suponiendo que se tardó 4 minutos en escribir el código, se tendría una productividad par el primer caso de 4 LOC/4 min = 1 LOC/min; en el otro caso 2 LOC/4 min = .5 LOC/min

Aquí el punto es ser consistente en la forma en que escribamos nuestro código

Ejemplos de Líneas Lógicas

1)

```
        if A < B then
                C:= C + 1
        else
                C:= C – 1
        1 LOC
```

2)

```
        for i:= 0 to z
          i:= i * 2
        if i > 20 and i < 30 then
        z:= 2
```

```
else if i > 30 and i < 40 then
z:= 3
else
        z:= 4
2 LOC
```

Las líneas lógicas son aquellas que representan un agrupamiento lógico de líneas, que en su conjunto tienen un significado.

Cómo agrupar estas líneas en una estructura lógica es algo que cada uno debe definir y ser constante en contar estas estructuras de la misma manera.

2.1.1.2.2.1. Líneas Lógicas Vs. Líneas Físicas

- **Líneas Lógicas**
  - Invariable a los cambios de edición.
  - Definición única.
  - Compleja de contar.
- **Líneas Físicas**
  - Fácil de contar.
  - Variable.
  - No existe una definición única.

Para las líneas lógicas.- invariable a los cambios de edición significa que no importa si se le agrega y quita código, siempre representará una estructura (una única LOC) a contar. Definición única, que una vez definida qué es y cómo es una estructura se mantendrá así. Compleja de contar, no es tan sencilla su automatización y hay que tener cuidado para realmente dimensionar durante el conteo el alcance de la estructura definida.

Para las líneas físicas.- es sencilla la automatización del conteo, variable y que no existe una única definición significa que la manera en que se escribe el código, como en ejemplo, podría dar resultados variables.

Líneas Físicas

En este curso tomaremos como estándar a utilizar para medir el tamaño de los programas las líneas físicas.

No debemos contar los espacios en blanco, ni los comentarios.

Si en una línea hay código y comentarios, sí debe ser contada.

```
if a < z then a:= 1; // a debe ser impar.
```

La LOC anterior se cuenta como una.

Para ser constantes en nuestra forma de escribir y contar nuestro código debemos de seguir algún estándar:

- **Para escribir:** estándar de codificación.
- **Para contar:** estándar de conteo.

El profesor debe proporcionar ejemplos de lo que es un estándar de codificación y un estándar de conteo. Ayudarse de los ejemplos anexos.

El estándar de codificación especifica el cómo debe el Ingeniero de Software escribir su código, describiendo por ejemplo, si va a utilizar encabezado en cada módulo de los programas, qué contendrá este encabezado. Cómo va a escribir las declaraciones de variables, si las variables deberán tener nombres significativos, si cada variable estará en una sola línea o se pueden poner varias variables en una sola, la manera de indentar el texto, si las constantes serán escritas solo en mayúsculas, etc.

Un ejercicio de investigación adecuado para este ejercicio es el encontrar el estándar de codificación propuesto por la compañía que desarrolló el lenguaje de programación o compilador que está utilizando cada alumno.

En Resumen

- Medimos para poder realizar estimaciones y tomar decisiones sobre la evolución de nuestros proyectos.
- Estimar el esfuerzo para escribir un programa es más complejo que estimar el tiempo para leer una revista.
- No hay métodos que garanticen una estimación exacta de los programas.
- La clave para realizar estimaciones exactas es:
- Realizar muchas estimaciones y comparar contra la realidad.

Ejercicios

1) Definir:

- Tu estándar de codificación.
- Tu estándar de conteo.

Ejercicio de Programación

2) Escribe un programa para contar las líneas físicas de tus programas. No debes contar las líneas en blanco ni los comentarios.

Cuenta manualmente las líneas de código de tus programas 1 y 2 y compáralos con el resultado de tu programa 2.

Los alumnos deberán reportar el resultado de sus pruebas, el profesor debe indicar el formato de reporte de pruebas y asegurarse que este formato sea seguido.

Orden de entrega de los documentos

- Forma de Registro de Compromisos.
- Forma de Registro de Actividades.
- Código Fuente del Programa.
- Estándar de Codificación.
- Estándar de Conteo.

- Pantallas de la Interfase Gráfica.
- Pantallas de los Resultados.

Forma de Registro de Compromisos Unidad 2

Nombre: _____Fecha: _____
Proyecto:_____
Lenguaje de programación: _____

Tamaño del Progama (LOC) **Total de LOC físicas**	**Planeado**	**Actual**	**A la fecha**	
Tiempo en Fase (min.)	**Planeado**	**Actual**	**A la fecha**	**% a la fecha**
Compromiso y Solución				
Implementación				
Calidad y Liberación				
Total				
Productividad				
		Total de LOC físicas / Total tiempo	Total de LOC físicas / Total tiempo	

A continuación se muestra un ejemplo de cómo se debe llenar la Plantilla de Estándar de Conteo de LOC.

Plantilla de Estándar de Conteo de LOC

Nombre: Estándar de codificación para curso de programación Lenguaje: Delphi
Autor: Carlos Mojica Fecha: 16/Sep/2001

Tipo de conteo Físico/Lógico	**Tipo** Físico	**Comentarios**
Tipo de Sentencia	**Incluir**	**Comentarios**
Ejecutables No ejecutables:	**Sí**	
Declaraciones Directivas del compilador	**Sí** **Sí**	
Comentarios		**Todos los comentarios se contarán por separado de las LOC.**
En su propia línea	**Sí**	
Con código	**Sí**	
Banners	**Sí**	
Líneas en blanco	**No**	
Aclaraciones		
Nulos Sentencias vacías	**Sí** **Sí**	
Instanciaciones genéricas	**Sí**	**Cada instanciación deberá estar en una línea**
Begin...End Begin...End	**No** **No**	
Pbas de condición	**Sí**	
Evaluación de expresiones		**Como una línea si viene después de un if, else, elseif, case**
Símbolos End Símbolos End	**No** **No**	
Then, else, otherwise	**Sí**	**Como una línea si viene después de un if, else, elseif, case**
Elseif	**Sí**	**Como una línea si viene después de un if, else, elseif, case**
Palabras reservadas	**No**	
Etiquetas	**No**	**No se utilizarán etiquetas**

3. La Planeación del Producto

Introducción

Una vez que hemos aprendido a medir nuestra productividad al desarrollar software y la hemos entendido, aprenderemos algunos métodos de estimación del tamaño del producto y el esfuerzo necesario para desarrollarlo. Se estudiarán las técnicas de estimación Wideband-Delphi, por Analogía y de PERT al igual que la media de la productividad para estimar el tamaño y el esfuerzo respectivamente.

Las técnicas serán descritas en términos de su uso y no se discutirá sobre los antecedentes teóricos estadísticos de su fundamentación. Para un tratamiento estadístico completo se recomienda acudir a los diversos textos al respecto. En el libro A Discipline for Software Engineering, existen capítulos y apéndices que dan algunos antecedentes estadísticos al respecto de las técnicas.

Objetivo General

Al término de esta unidad el alumno será capaz de:

- Describir técnicas para estimar el tamaño y esfuerzo en proyectos de desarrollo de software.

Objetivos Específicos

Al término de esta unidad el alumno será capaz de:

- Definir y aplicar técnicas de estimación del tamaño del código.
- Definir y aplicar técnicas de estimación del esfuerzo de desarrollo de proyectos de software.

3.1. Estimación del Tamaño

El requisito esencial para realizar cualquier plan es la habilidad para estimar el proyecto:

- Su tamaño, respecto a un indicador.
- Recursos necesarios (tiempo, personas, materiales, etc.).

Nota: La estimación del tamaño es lo primero.

El orden de estimación en un proyecto grande es: Tamaño, Tiempo (Esfuerzo), Calidad, Costos, Recursos, Riesgos. Para proyectos a nivel personal en este curso nos concentraremos en estimar el tamaño y el tiempo.

Un programa tiene diferentes componentes:

- Menús.
- Pantallas.
- Reportes.
- Archivos.
- Funciones:
    - Sencillas o complejas.
    - De texto, matemáticas.
    - Lógicas, de entrada/salida, etc.

De ahora en adelante debemos encontrar cuáles son las partes que compondrán a nuestros programas, es decir, qué componentes tenemos que construir para satisfacer los requerimientos. El trabajo de encontrar estos componentes es parte de lo que se conoce como diseño conceptual y entra en la fase que hemos definido como Compromiso y Solución.

Para estimar el total del tamaño de un programa:

- Estimar el tamaño de cada componente.
- Sumar las estimaciones de los componentes.

Si la estimación no te parece lógica, realiza el ajuste necesario.

Las estimaciones no son una ciencia exacta, a pesar de que existen diversos métodos para estimar todos dependen de datos históricos, al final todos estos métodos envuelven algún tipo de juicio de experto, lo cual depende de la experiencia. Este juicio será mucho mejor si está basado en datos históricos más que en una adivinanza.

Ejemplo 1

Programa	Componente	Tiempo	LOC físicas
1	Login/Password	45	24
2	Regresión lineal simple	322	134
3	Búsqueda parecida	123	78
4	Validación IVA	12	22
5	Ordenamiento de enteros	66	43

Aquí se muestra un ejemplo de registro de componentes con su tiempo de desarrollo y las líneas de código físicas que contienen. Es momento oportuno para señalar a los estudiantes que es importante que en la forma de registro de actividades se anote el desarrollo de cada componente.

Ejemplo 2

Programa	Componente	Tipo	Tiempo	LOC físicas
1	Login/Password	Entrada/ Salida	45	24
2	Regresión lineal simple	Matemática	322	134
3	Búsqueda parecida	Texto	123	78
4	Validación IVA	Lógica	12	22
5	Ordenamiento de enteros	Datos	66	43

En este ejemplo se ha añadido la columna de tipo con la cual se clasifica a cada uno de los componentes, de esta manera será más fácil ubicar un componente que estemos buscando.

3.2. Métodos de Estimación

Ya que el objetivo principal de la estimación del tamaño es tener estimaciones exactas para hacer mejores planes, debes experimentar con varios métodos de estimación y usar el que mejor te ajuste.

- **Wideband-Delphi.**
- **Por analogía.**
- **PERT.**

Existen diversos métodos para estimar el tamaño, pasando de sólo usar la experiencia hasta aquellos que utilizan técnicas estadísticas paramétricas. Un buen ejercicio para los alumnos es el asignarles que investiguen y presenten alguna otra técnica de estimación aparte de las mostradas.

3.2.1 Wideband-Delphi

Es una técnica basada en juicio de experto. Se le pide a varias personas por separado realicen la estimación y se la proveen a un coordinador.

El coordinador calcula el promedio de la estimación. Registra el promedio y las estimaciones de los expertos en una forma.

Si el coordinador juzga que no es lógico el resultado, realiza una nueva ronda proveyendo los resultados.

Ejemplos

1) Se les pide a 3 personas realizar sus estimaciones sobre un producto.

a) Las estimaciones iniciales son:

- A = 13,800 LOC
- B = 15,700 LOC
- C = 21,000 LOC

b) Luego, el coordinador:

- Calcula el promedio, 16,833 LOC
- Proporciona a cada persona este promedio y las otras estimaciones sin decir de quien es cual.

c) Las personas que realizaron las estimaciones se reúnen y discuten las estimaciones.

d) Sus segundas estimaciones son:

- A = 18,500 LOC
- B = 19,500 LOC
- C = 20,000 LOC

e) Luego, el coordinador:

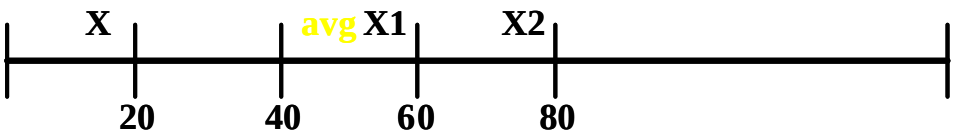
- Calcula el promedio, 19,333 LOC
- Pregunta a los que realizaron las estimaciones si están de acuerdo con esta estimación

Ventajas:

- Puede producir resultados muy precisos.
- Se usan habilidades de la organización.
- Trabaja para cualquier tamaño de producto.

Desventajas:

- Recae sobre pocos expertos.
- Consume tiempo.
- Está sujeta a percepciones personales.



Esta es la manera en que el coordinador proporciona la información a los expertos para que realicen una nueva estimación.

3.2.2 Estimación por analogía

También está basada en la experiencia, es decir, la utilización de datos históricos. Una vez encontrados los componentes del programa, se busca en la base de datos por un componente similar.

Si se encuentra alguno similar tomar el tamaño del componente encontrado como una base para la estimación.

Cada vez que se mencione la experiencia, hacer hincapié de lo importante que es contar con una base de datos histórica de mediciones. Un ejercicio oportuno en este momento es que los estudiantes revisen sus dos programas anteriores y registren cada componente en la Forma de Registro de Tamaños.

Supongamos que necesitamos realizar un componente que orden números flotantes, si buscamos en la base de datos históricas podremos encontrar que tenemos uno que ordena números enteros, 43 LOC.

Utilizando tú juicio puede ajustar estas LOC a que sean más o menos o quizás la utilices como base combinándola con la técnica Wideband-Delphi.

3.2.3 PERT

Esta técnica también tiene sus fundamentos en técnicas estadísticas. Está basada en tomar 3 valores:

- El valor mínimo esperado.
- El valor más probable.
- El valor máximo esperado.

y aplicar una fórmula.

Otro ejercicio interesante, dependiendo del nivel de estudios, es el dejar un trabajo de investigación y exponer los fundamentos teóricos de PERT.

a = Valor Inferior

b = El valor esperado

c = Valor Superior

$$E = \frac{a + 4b + c}{6}$$

Los valores a, b y c pueden ser obtenidos utilizando la técnica de Wideband-Delphi o la técnica por analogía.

Ejemplo:

Valor mínimo pensado = 87

Valor más probable = 95

Valor máximo pensado = 120

E = (87 + 4*95 + 120)/6= 97.83 = 98 LOC

Para determinar cuánto tiempo nos puede tomar el desarrollar nuestro proyecto, podemos acudir a la técnica de analogía, una vez que hemos estimado el tamaño total del proyecto podemos buscar en nuestra base de datos histórica de estimaciones.

También podemos recurrir a utilizar una técnica estadística utilizando nuestros datos históricos de productividad.

$$\textbf{Tiempo estimado} = \frac{\textbf{LOC estimadas del proyecto}}{\textbf{Media de la Productividad}}$$

$$\textbf{Media de la Productividad} = \frac{\textbf{Total de LOC de todos los proyectos realizados}}{\textbf{Total del tiempo empleado en los proyectos realizados}}$$

No es la mejor aproximación pero sí da un mejor reflejo que si sólo utilizáramos analogía y mejor aún que si adivináramos, para un mejor detalle de las técnicas estadísticas referirse al libro “A discipline for Software Engineering”.

La información necesaria se toma de la Forma de Registro de Compromisos.

Ejemplo de estimación del esfuerzo

Tomando los datos del ejemplo de la Forma de Registro de Tamaños tenemos:

- LOC totales a la fecha = 301
- Minutos totales a la fecha = 568

Media de la productividad = $\frac{301}{568}$ = 0.53

Tiempo estimado = $\frac{98}{0.53}$ = 184.9 = 185 minutos

3.3. Distribución del Tiempo en las Fases

Una vez estimado el tiempo total de desarrollo, debes usar este dato y el porcentaje a la fecha de los tiempos empleados en cada fase para distribuirlo en tus nuevas estimaciones.

Ejemplo:

Has estimado que tardarás 185 minutos en desarrollar el nuevo programa y tienes en tu forma de Registro de Compromisos los siguientes datos:

Fase	% a la fecha
Compromiso y Solución	25
Implementación	60
Calidad y Liberación	15

En tu nuevo programa tendrás durante la planeación:

Fase	Planeado
Compromiso y Solución	46
Implementación	111
Calidad y Liberación	28

Ejercicio de Programación

Desarrollar un programa para contar el número de LOC físicas de cada función y procedimiento de un programa en particular. Este mismo programa debe arrojar el

total de LOC del mismo. Utiliza tu estándar de conteo y codificación para construir la solución.

Orden de entrega de los documentos

- Forma de Registro de Compromisos.
- Forma de Registro de Actividades.
- Código Fuente del Programa.
- Estándar de Codificación.
- Estándar de Conteo.
- Pantallas de la Interfase Gráfica.
- Pantallas de los Resultados.

Nota: El orden de entrega de la documentación es importante, si no se cumple, disminuir puntos de la calificación.

Forma de Registro de Compromisos Unidad 3

Nombre:_____ Fecha: _____
Proyecto:_____
Lenguaje de programación: _____

Tamaño del Progama (LOC) **Total de LOC físicas**	**Planeado**	**Actual**	**A la fecha**	
Tiempo en Fase (min.)	**Planeado**	**Actual**	**A la fecha**	**% a la fecha**
Compromiso y Solución				
Implementación				
Calidad y Liberación				
Total				
Productividad				
		Total de LOC físicas / Total tiempo	Total de LOC físicas / Total tiempo	

Forma de Registro de Tamaños

Nombre:_____Fecha: _____
Lenguaje de programación: _____

Programa	Componente	Tipo	Tiempo	LOC

4. La Agenda de Trabajo

Introducción

Ya que hemos calculado el esfuerzo que vamos a necesitar para desarrollar nuestros productos de software, procede el distribuir esta cantidad de esfuerzo en el tiempo. Por ejemplo, si ya hemos estimado que requeriremos de 18 horas para finalizar nuestro trabajo, quiere decir que ¿trabajaremos 18 horas seguidas? o quizá podamos trabajar 3 horas diarias durante 6 días. Cómo distribuir el esfuerzo del trabajo en el tiempo que tengamos disponible es tema de esta unidad, es decir, aprenderemos a planear nuestro trabajo.

Si hemos realizado un plan es porque lo vamos a seguir, sino, ¿qué caso tiene planear? Sin embargo, aún falta poder controlarlo, es decir, ¿cómo podemos estar seguros que a medida que transcurre el tiempo estamos alcanzando nuestros objetivos en el tiempo fijado? En este tema, se presentará la técnica de valor devengado, la cual nos servirá para poder saber cual es el porcentaje de avance real que lleva nuestro proyecto, más aún, poder saber cuánto más tiempo requeriremos para finalizar nuestro trabajo.

Objetivo General

Al término de esta unidad el alumno será capaz de:

- Planear el proyecto y darle seguimiento para poder asumir y cumplir los compromisos adquiridos.

Objetivos Específicos

Al término de esta unidad el alumno será capaz de:

- Describir técnicas para planear pequeños proyectos, proyectos a nivel personal.

- Describir técnicas para darle seguimiento a proyectos a nivel personal.

4.1. ¿Por qué es importante planear?

Antes de comprometernos a realizar un trabajo debemos estar seguros de:

- El propósito del compromiso.
- Las tareas que tienen que ser realizadas y cómo serán manejadas.
- El esfuerzo requerido para cumplir.

No podemos adquirir un compromiso sin conocer el alcance del mismo, refiriéndonos a compromiso a la realización de un proyecto en tiempo, costo y calidad. Por ello es que necesitamos descomponer en tareas manejables el problema a solucionar de tal forma que podamos dimensionar la cantidad de esfuerzo requerido para finalizar con éxito La planeación es el primer paso dentro de cualquier proceso, y aquí estamos hablando del proceso de desarrollo de software.

¿Por qué son necesarios los planes?

1) Proveen una base de negocio para realizar el trabajo:

- Estableciendo el precio.
- Definiendo la agenda.
- Acordando el trabajo.

2) Establecen un marco de trabajo para la administración:

- Definiendo los compromisos.
- Ayudando a coordinar el trabajo.
- Permitiendo el seguimiento del estado del proyecto.

Debido a que en los planes se encuentran detalladas todas y cada una de las tareas en términos de alcance, responsables, tiempo de inicio y finalización, podemos establecer el costo. De esta manera quedan establecidos los compromisos por parte de los ejecutores.

En la ejecución del proyecto, los planes sirven para dar seguimiento al cumplimiento de los compromisos y poder coordinar el trabajo que debe ser ejecutado para cumplirlos.

No hay planeación perfecta

La planeación es una habilidad que se desarrolla.

Aún los planes más simples están sujetos a errores.

- Eventos no previstos.
- Complicaciones inesperadas.
- Errores en la planeación.

La mejor estrategia es planear a detalle:

- Identificar las tareas.
- Estimar basándose en experiencias similares.
- Hacer un buen juicio con lo demás.

La planeación es un juego de adivinar el futuro y adelantarse a los problemas que puedan presentarse. La cantidad y calidad de la información que tengamos al momento de planear es decisiva para determinar que tan bueno o malo será el plan, a menor información existe mayor riesgo de un plan imperfecto. ¿Qué tanto y cuánto planear? son las interrogantes a responder, a nivel personal, el nivel al que está orientado este curso, es conveniente detallar lo más posible el conjunto de tareas que como Ingenieros de Software vamos a realizar.

Alguien, de alguna manera, ya ha descompuesto el problema total y nos ha dado el pedazo de problema que tenemos que desarrollar, a este nivel van enfocadas las técnicas descritas en este capítulo.

Como ejemplos de eventos no previstos pudiéramos pensar en desastres naturales como temblores, huracanes, apagones, enfermedades. Sobre las complicaciones algo que se pensó era sencillo resulto más complejo, por lo que se previó menos tiempo dando pie a errores en la planeación.

4.1.1. ¿Qué tomamos en cuenta para planear?

Básicamente pensamos en dos cosas:

- El producto a construir.
- El tiempo en que debe/será construido.

El producto debe ser descompuesto en sus subproductos, en tareas y servicios necesarios para su construcción. Una vez obtenida esta lista de productos y servicios, es necesario encontrar las dependencias de construcción entre ellos, es decir, qué se debe hacer primero, qué después y así sucesivamente.

Ya qué sabemos qué se tiene que hacer y en qué orden y habiendo establecido el tiempo necesario para desarrollar las tareas y servicios queda la tarea de establecer en qué lapso de tiempo serán desarrollados, es decir, la manera en que pensamos vamos a invertir nuestro tiempo, acabar en un día, una semana, un mes, un año, etc.

4.1.2. Pasos para Planear

1) Tener una definición clara del producto a construir en términos de:

1.1) Características del producto.

1.2) Tamaño de cada característica.

1.3) Tiempo requerido para cada característica.

1.4) Tiempo disponible para trabajar en el desarrollo.

Forma de Registro de Tamaños

Nombre:_____ Fecha: _____
Lenguaje de programación: _____

Programa	Componente	Tipo	Tiempo	LOC

2) Preparar el plan.

Los requisitos del producto a construir son esenciales pues ellos delimitan el alcance del producto, aunque hay que tomar en cuenta algunas otras cosas como restricciones de equipo u otro material que se vaya a utilizar. A partir de estos requisitos encontramos los componentes que formarán parte del producto y les estimamos su tamaño y el tiempo de desarrollo. Con esta información estamos preparados para documentar el plan al comenzar a generar la agenda de trabajo.

4.2. Estimación de la Agenda

Para realizar la agenda se necesitan 3 cosas:

1) La estimación de las horas directas del proyecto.
2) Un calendario de las horas directas disponibles.
3) El orden en que las tareas serán realizadas.

Luego se necesitan realizar:

1) Las estimaciones del tiempo necesario para cada tarea.
2) Acomodar este tiempo a lo largo del calendario.

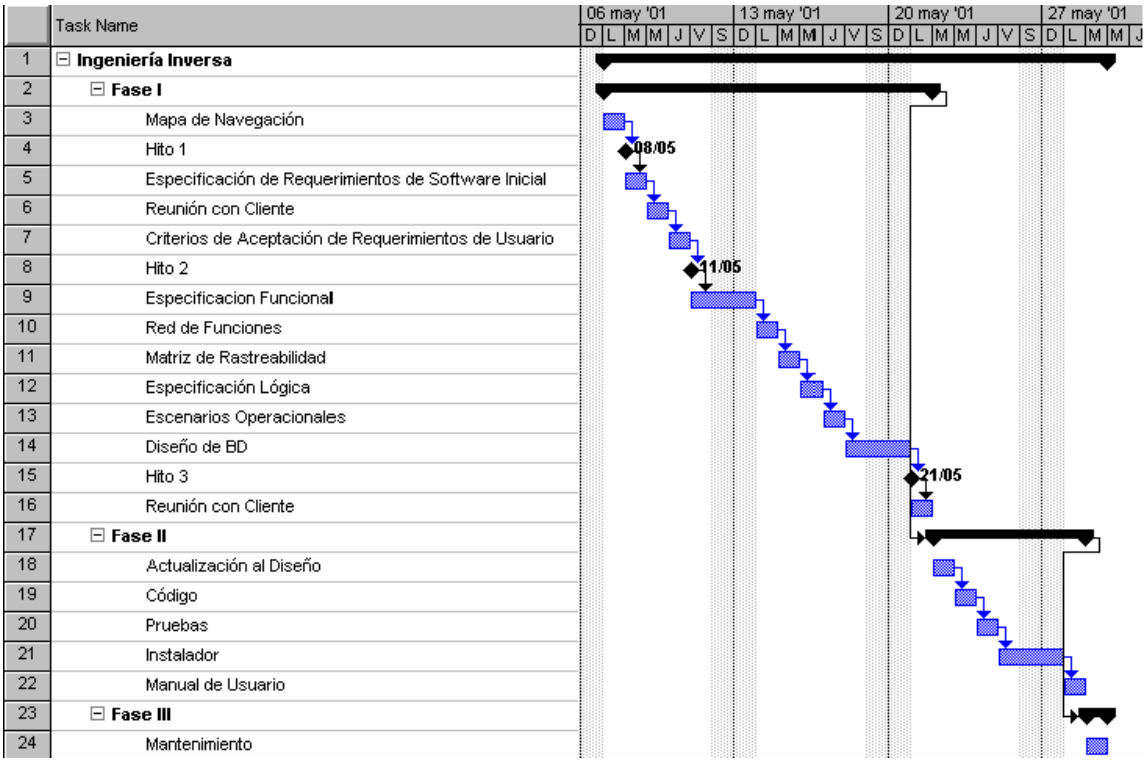
La agenda de trabajo describe todas las actividades a realizarse y el momento en que deben iniciar y terminar, es decir, las fechas de inicio y finalización. Para crearla debimos ya haber estimado el esfuerzo requerido para el proyecto.

El calendario de las horas directas disponibles se refiere a que debemos saber cuáles son los momentos, períodos de tiempo, que tenemos disponibles para

llevar a cabo la ejecución de las tareas. Quizá el proyecto que vamos a desarrollar no sea la única actividad que tengamos.

Con la información anterior ya estamos en posición de poder distribuir el esfuerzo requerido por cada tarea en los espacios de tiempo que tengamos disponibles.

Ejemplo de una gráfica de Gantt



Lo más representativo para describir una agenda es la gráfica de Gantt, en la cual barras describen el esfuerzo requerido por las tareas y el inicio de la barra comienza en el momento en que debe iniciar la tarea y finaliza, la barra, también correspondiendo con la finalización planeada de la tarea. En este curso no generaremos gráficas de Gantt (el nombre viene por la persona que desarrolló por primera vez este tipo de gráfica), pero con la información que manejaremos podemos construirla sin dificultad.

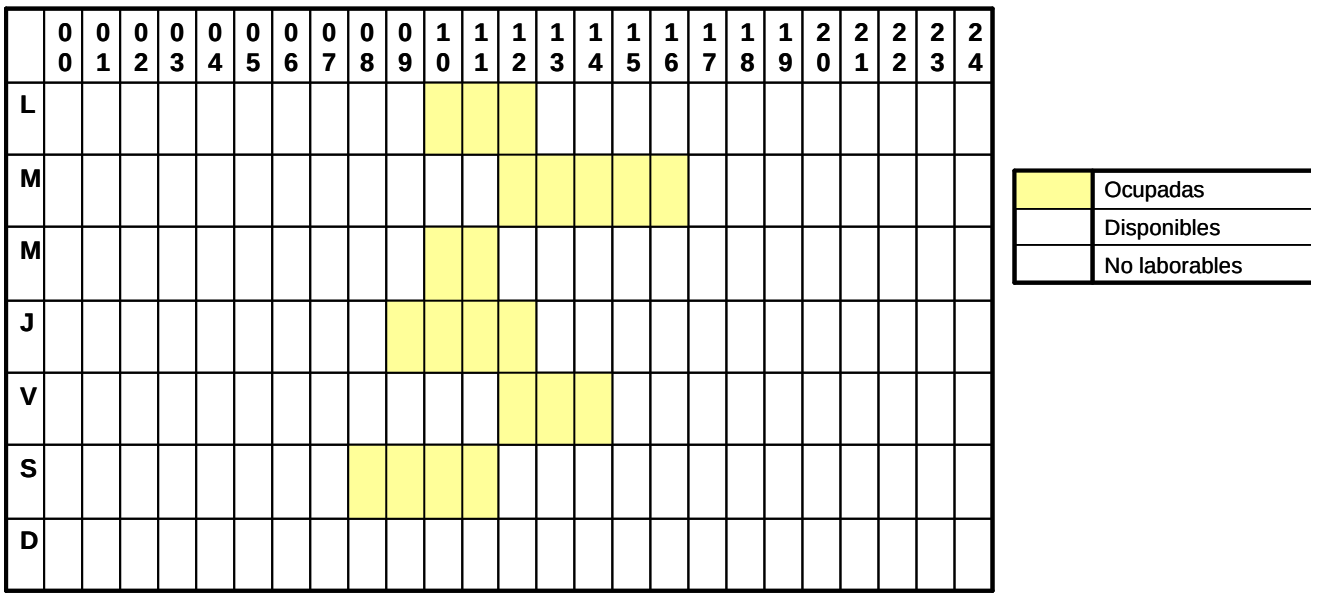
Horas disponibles directas

Producir un calendario con el tiempo disponible.

- 52 semanas en un año con 40 horas de trabajo por semana = 2080 horas.
- Con 3 semanas de vacaciones y 10 días de asueto; un año = 1880 horas (90%).
- Con 10% para reuniones, 55 hrs. para correo-e…; un año de trabajo representa entre 1000 a 1400 horas (50 a 65%).

Aquí un ejercicio de reflexión es pedirles a los alumnos revisen su Forma de Registro de Actividades y calculen cuánto tiempo realmente trabajan a la semana, al mes, al año.

Horas disponibles directas



El saber cuáles son los momentos que tenemos disponibles es importante, ya que en ellos podremos asignar esfuerzo para realizar las tareas del proyecto que estamos planeando.

El orden de las tareas

El orden de las tareas debe estar dictado por la estrategia de desarrollo.

- Cada tarea necesita un criterio de finalización.
- Se deben considerar las dependencias entre las tareas.

- También considerar los costos y las prioridades de tiempo o de cualquier otra índole.

Determinar el orden de las tareas planeadas.

- El orden de las tareas pudiera cambiar a medida que avanza el proyecto.

Realizar un bosquejo o mejor, un diseño conceptual, de la solución del problema nos ayuda a encontrar las dependencias existentes entre las diversas tareas, es decir, qué se tiene que hacer primero. Muchas veces, por diversas cuestiones como por ejemplo requerimientos del usuario, pudiéramos cambiar el orden inicial que le dimos a la consecución de las tareas. Sin embargo, debemos de tener cuidado en no poner por delante tareas que dependen de otras aún sean exigencias del cliente.

Ya que hemos quedado que planear es predecir, y al estar construyendo la agenda estamos prediciendo la mejor manera en que debemos construir el producto, a medida que avanza el proyecto adquirimos nuevo conocimiento o simplemente las prioridades cambian, por lo que la re-planeación será algo común.

5. Seguimiento de la Agenda de Trabajo

¿Habrá algo inusual que afecte al proyecto?

Distribuir el tiempo de las tareas sobre el calendario.

- Identificar los hitos principales del proyecto.
- Usar un formato estándar.

Generalmente un error que cometemos al planear es pensar que todo va a salir bien, pero la realidad es otra. Hay que pensar en qué cosas pueden salir mal y tenerlas contempladas al momento de realizar y revisar las estimaciones, más cuando estamos distribuyendo las tareas en el tiempo.

El asignar hitos, el identificar cosas, personas o hechos claves en el proyecto nos permitirá dar un mejor seguimiento al proyecto. Los hitos no tienen duración, son puntos en el tiempo que nos permitirán saber si hemos alcanzado o no logros a lo largo del proyecto para poder tomar decisiones de si seguir o no con la realización del proyecto, de si debemos o no cambiar la estrategia que estamos siguiendo.

La estandarización en el manejo de la agenda nos permitirá ser consistente en el registro y análisis de la información a través del tiempo, a través de diferentes proyectos.

Plantilla de planeación de tareas

Plantilla de Planeación de Tareas

Nombre:_____ Fecha:_____

Proyecto:_____

Tarea		Plan					Actual		
Id	Nombre	Minutos	Valor planeado	Minutos acumulados	Valor planeado acumulado	Fecha	Fecha	Valor ganado	Valor ganado acumulado

Para comenzar a llenar esta plantilla:

1) Listar las tareas en el orden respectivo de finalización.

2) Registrar los minutos que se espera tome cada tarea.

3) Añadir los minutos en la columna de minutos acumulados.

En este punto, comenzar a preparar la plantilla de planeación de la agenda.

Se tiene que explicar la utilización de las plantillas, Plantilla de Planeación de Tareas y la Plantilla de Planeación de la Agenda.

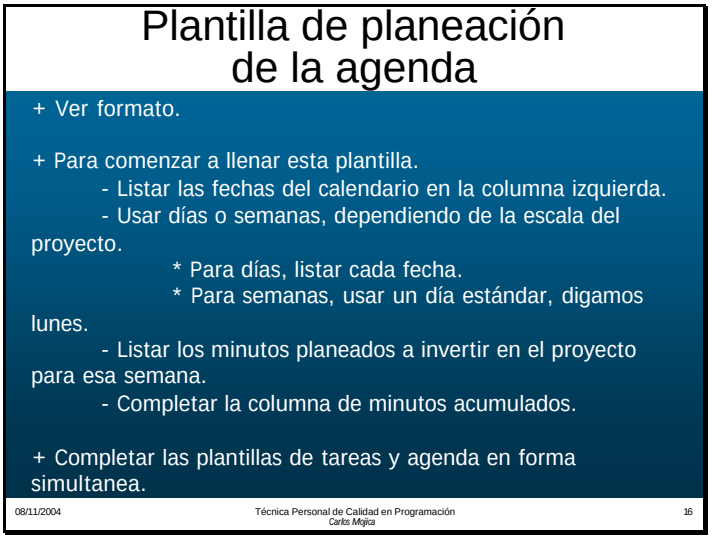
En esta primera lámina se describe el uso de la Plantilla de Planeación de Tareas. En la columna Id se pone un número entero único con el cual hacer referencia a las tareas y bajo la columna Nombre se listan las tareas en el orden en que se piensan completar. Recordar que en cada tarea listada debe quedar explícito su criterio de finalización, un truco sencillo es por ejemplo mencionar la finalización del producto o acción: Planeación finalizada, programa compilado, defectos corregidos, prueba finalizada.

Una vez llenada esta lista (en términos de planeación se le conoce como estructura de descomposición de trabajo), se asigna el tiempo de desarrollo, bajo la columna Minutos.

Ejemplo de Planeación de la Agenda

Usando la plantilla de planeación de tareas, comenzar con el desglose de las tareas y el tiempo estimado para cada una de ellas.

Id tarea	Nombre tarea	Horas	Horas acumuladas
1	A	2	2
2	B	5	7
3	C	4	11
4	D	7	18
5	E	3	21
6	F	5	26
7	G	6	32
8	H	3	35
9	I	2	37



Ejemplo de Planeación de la Agenda

Usando la plantilla de planeación de la agenda, estimar los minutos a emplear en el proyecto.

Número de día	Fecha	Horas directas	Horas acumuladas
1	03/jul	3	3
2	04/jul	5	8
3	05/jul	5	13
4	10/jul	5	18
5	11/jul	4	22
6	12/jul	6	28
7	13/jul	5	33
8	14/jul	5	38

La columna Número de día es un consecutivo para saber el total de días que trabajaremos en el proyecto, la columna Fecha denota eso, la fecha en la cual trabajaremos en el proyecto, las Horas directas es el total del tiempo que pensamos invertir en el proyecto en la Fecha dada, y finalmente la columna de Horas acumuladas, es la suma acumulada del tiempo distribuido en el período del proyecto que pensamos invertirle. Así para el 3 de julio pensamos invertir únicamente 3 horas al proyecto, el día 4, 5 y 10 de julio invertiremos en cada uno 5 horas al proyecto.

En una lámina anterior estimamos que el proyecto va a durar 37 horas (la suma del tiempo de las 9 tareas que se están planeando), en la agenda hemos reservado en nuestro calendario 38 horas para el proyecto.

5.1. Completando el Plan

Para cada tarea:

- Encontrar las horas acumuladas para completar esa tarea en la plantilla de tareas.
- Encontrar la semana en la plantilla de la agenda cuando esas horas se han excedido por primera vez.
- Registrar la fecha de la semana en la columna de fecha para esa tarea en la plantilla de tareas.

Ahora se tiene ya la agenda de las tareas.

Ejemplo de Planeación de la Agenda

Registrar la agenda de la tarea: el día en el cual las horas acumuladas para cada tarea será alcanzada.

Id tarea	Horas	Horas acumuladas	Día
1	2	2	1
2	5	7	2
3	4	11	3
4	7	18	4
5	3	21	5
6	5	26	6
7	6	32	7
8	3	35	8
9	2	37	8

Regresando a la Plantilla de Planeación de Tareas, en la columna Fecha de la parte de Planeación se registra el día en el cual se piensa se va a terminar cada una de las tareas, esto se hace revisando la columna Horas (minutos) acumulados dentro de la parte de planeación de la Plantilla para la Planeación de la Agenda. Por ejemplo la tarea número 1 tiene una duración planeada de 2 horas, para el día

3 de julio se ha programado trabajar 3 horas, lo que quiere decir que finalizaremos la tarea 1 y nos sobrará una hora. Esta hora sobrante será empleada para iniciar la tarea 2, cuya duración planeada es de 5, por lo que nos quedarán 4 horas. Para el día 4 de julio se han reservado 5 horas, suficientes para finalizar la tarea 2 e iniciar la tarea 3, es decir, que la tarea 2 la finalizaremos el día 2. De esta manera se procede con las demás tareas.

5.2. Valor Ganado

El propósito del valor ganado es:

- Establecer un valor para cada tarea.
- Permitir dar seguimiento al progreso comparándolo contra lo planeado.
- Facilitar el seguimiento aún cuando el plan cambie.

Los principios tras el valor ganado son:

- Provee un valor común para cada tarea.
- Este valor es el porcentaje que esta tarea tiene respecto al total de horas planeadas del proyecto.
- No se gana nada por tareas parcialmente completadas.
- Si el proyecto sufre cambios grandes, se requieren nuevos planes.

Es común encontrar la siguiente situación, supongamos que tenemos un proyecto que hemos establecido que vamos a finalizarlo en 10 días, por lo cual muchas de las veces se asume, erróneamente, que al finalizar el día 1 se tendrá 10% de avance, el día 2, 20% de avance y así por cada uno de los días hasta llegar el día 10 donde se alcanzará el 100%. Quizá si pueda darse esta situación, pero debemos estar seguros de eso. ¿Cómo hacerlo?, encontrando una unidad que permita distribuir la cantidad de trabajo alcanzado una vez que es finalizada cada tarea, esto es el valor ganado.

5.2.2. Establecimiento del Valor Planeado

En la plantilla de planeación de tareas:

- Sumar las horas del proyecto.

- Calcular el % de cada tarea respecto al total de horas.
- Registrar este % como el valor planeado para la tarea.
- Calcular el valor planeado acumulado para cada tarea.

En la plantilla de la agenda:

- Registrar el valor planeado acumulado para las tareas que sean completadas cada semana.

Ejemplo de Planeación de la Agenda

Luego, calcular el valor planeado:

Tarea	Horas	Horas acumuladas	Día	Valor planeado	Valor planeado acumulado
1	2	2	1	5.4	5.4
2	5	7	2	13.5	18.9
3	4	11	3	10.8	29.7
4	7	18	4	18.9	48.6
5	3	21	5	8.1	56.7
6	5	26	6	13.5	70.2
7	6	32	7	16.3	86.5
8	3	35	8	8.1	94.6
9	2	37	8	5.4	100.0

El valor planeado, lo que se conseguirá de avance al terminar la tarea 1 es 100*2/37=5.4, 2 el tiempo planeado para la tarea 1 y 37 el total del tiempo que se va a invertir en el proyecto.

Ejemplo de Planeación de la Agenda

Día	Horas	Horas acumuladas	Valor planeado acumulado
1	3	3	5.4
2	5	8	18.9
3	5	13	29.7
4	5	18	48.6
5	4	22	56.7
6	6	28	70.2
7	5	33	86.5
8	5	38	100.0

Luego, registrar el valor planeado acumulado para cada día:

Tomando la columna Fecha de la sección de Planeación de la Plantilla de Planeación de Tareas, se suman los valores planeados para cada fecha y se registra esta suma en la columna Valor planeado acumulado de la sección de planeación de la Plantilla para la planeación de la Agenda.

Con esto se completa la planeación del proyecto.

Ejecutando el Plan

¡Si se ha realizado un plan, hay que seguirlo, sino para que planeamos!

5.2.2.1 Dándole seguimiento al plan

A medida que cada tarea es completada, gana su valor planeado:

- Registrar este valor ganado.
- Registrar la fecha en que la tarea fue finalizada.
- Añadir el valor ganado a la fecha en la columna de valor ganado acumulado.

En la plantilla de la agenda, registrar el valor ganado acumulado para cada semana a medida que se completan.

Dar seguimiento al valor ganado respecto al valor planeado por semana.

Una tarea gana su valor solamente si es completada. Si al finalizar el día no se completó la tarea su valor no se contabilizará para ese día, sino hasta el día en que se termine. Al completar una tarea, su valor se registra al igual que el valor acumulado hasta ese día. El valor ganado nos sirve para saber el estado de nuestro proyecto, es decir, si estamos adelantados, atrasados o igual a como planeamos solucionar el proyecto.

5.3. Proyectando la finalización del proyecto

Se asume que el proyecto continuará ganando valor a la velocidad que lo ha ganado.

Extrapolar los resultados del proyecto extendiendo linealmente la función de valor ganado hasta que alcance el 100%.

Con la información anterior se determina la fecha de finalización proyectada, a menos que:

Esta es la fecha de finalización proyectada, a menos que:

- La velocidad del proyecto sea cambiada.
- El trabajo que falta para las tareas restantes pueda ser reducido por abajo de lo planeado.

En la lámina [8/8] del ejemplo de planeación de la agenda se explica la proyección (la extrapolación) para saber cuando finalizaríamos el proyecto.

Ejemplo de Planeación de la agenda

Durante el proyecto, registrar en la plantilla de planeación de tareas el día en que cada tarea es completada.

Tarea	Horas	Horas acumuladas	Día	Valor planeado	Valor planeado acumulado	Día de finalización
1	2	2	1	5.4	5.4	1
2	5	7	2	13.5	18.9	2
3	4	11	3	10.8	29.7	4
4	7	18	4	18.9	48.6	5
5	3	21	5	8.1	56.7	
6	5	26	6	13.5	70.2	
7	6	32	7	16.3	86.5	
8	3	35	8	8.1	94.6	
9	2	37	8	5.4	100.0	

Ejemplo de Planeación de la Agenda

También registrar en la plantilla de planeación de la agenda el valor ganado cada día.

Día	Horas	Horas acumuladas	Valor planeado acumulado	Valor ganado
1	3	3	5.4	5.4
2	5	8	18.9	18.9
3	5	13	29.7	18.9
4	5	18	48.6	29.7
5	4	22	56.7	48.6
6	6	28	70.2	
7	5	33	86.5	
8	5	38	100.0	

Ejemplo de Planeación de la Agenda

Día	Horas	Horas acumuladas	Valor planeado acumulado	Valor ganado	Valor ganado proyectado
1	3	3	5.4	5.4	5.4
2	5	8	18.9	18.9	18.9
3	5	13	29.7	18.9	18.9
4	5	18	48.6	29.7	29.7
5	4	22	56.7	48.6	48.6
6	6	28	70.2		58.3
7	5	33	86.5		68.0
8	5	38	100.0		77.8
9					87.5
10					97.2
11					100.0

Usando el valor ganado actual por día de 9.72, registrar el valor ganado por día para proyectar la finalización.

En el ejemplo en el día 5 del proyecto hemos alcanzado (avanzado) el 48.6% por lo que en promedio hemos logrado 9.72 (48.6/5), en una columna que llamaremos Valor ganado proyectado, podemos ver el comportamiento que tendrá nuestro proyecto (qué tanto vamos a avanzar en los días subsecuentes para poder saber cuándo vamos a finalizar el proyecto) si continuamos trabajando de la manera en que lo hemos venido haciendo. Así, a partir del 6 día le sumamos 9.72, es decir si continuamos la tendencia de sólo lograr 9.72% de avance diario finalizaremos el proyecto hasta el día 11 y no para el día 10 como estaba planeado.

Ejercicio de programación

Se tienen dos listas de datos de números enteros, las cuales no necesariamente se encuentran ordenadas ni tienen la misma cantidad de elementos. Se desea crear una tercera lista que contenga todos los números (sin repetición), de las dos listas iniciales, esta tercera lista estará en orden creciente o decreciente a preferencia del usuario. Al final, el programa debe desplegar las tres listas en el orden seleccionado por el usuario, el número de elementos que se repiten y la diferencia entre el elemento mayor y menor. Utilizar listas enlazadas en la solución.

Ejemplo

Entrada

l1 = {23, 12, 4, 56, 13, 9, 6}

l2 = {5, 43, 23, 6, 27, 8, 25}

Selección

Ordenamiento = Creciente

Salida

l1 = {4, 6, 9, 12, 13, 23, 56}

l2 = {5, 6, 8, 23, 25, 27, 43}

l3 = {4, 5, 6, 8, 9, 12, 13, 23, 25, 27, 43, 56}

Elementos repetidos = 2

Diferencia = 52.

Nota: Los alumnos deberán reportar el resultado de sus pruebas.

Orden de entrega de los documentos

- Forma de Registro de Compromisos.
- Plantilla de Planeación de Tareas.
- Plantilla para la Planeación de la Agenda.
- Forma de Registro de Actividades.
- Código Fuente del Programa.
- Estándar de codificación.
- Estándar de conteo.
- Pantallas de la interfase gráfica.
- Pantallas de los resultados.

Nota: El orden de entrega de la documentación es importante, si no se cumple, disminuir puntos de la calificación.

Forma de Registro de Compromisos

Nombre: _____ Fecha: _____
Proyecto:_____
Lenguaje de programación: _____

Tamaño del Progama (LOC)	**Planeado**	**Actual**	**A la fecha**	
Total de LOC físicas				
Tiempo en Fase (min.)	**Planeado**	**Actual**	**A la fecha**	**% a la fecha**
Compromiso y Solución				
Implementación				
Calidad y Liberación				
Total				
Productividad				
		Total de LOC físicas / Total tiempo	Total de LOC físicas / Total tiempo	

6. La Administración de los Defectos

Introducción

El trabajo de un desarrollador de software es entregar productos de software de calidad en los tiempos y al costo en que fueron planeados, para ello es imprescindible que el software esté libre de defectos. En esta unidad comenzaremos a recolectar información sobre los *errores* que cometemos, los clasificaremos y analizaremos para encontrar estrategias que nos ayuden a evitar el estar inyectándolos. Entenderemos que no es lo mismo un error que un defecto y pondremos al descubierto el costo que implica el no realizar un trabajo con calidad desde la primera vez.

Objetivo General

Al término de esta unidad el alumno será capaz de:

- Comprender el costo que tiene el cometer errores en el desarrollo de productos de software.

Objetivos Específicos

Al término de esta unidad el alumno será capaz de:

- Identificar y remover los defectos que inyectamos al producir productos de software.

¿Qué es la Administración de los Defectos?

Antes de contestar esta pregunta, debemos estar seguros de sabe qué es lo que se entiende por calidad del software.

Aquí el profesor comienza la discusión realizando la pregunta, ¿qué es la calidad del software?

La Calidad del Software

En términos sencillos, entendemos por calidad del software, desde el punto de vista del usuario que el software:

- Hace lo que tiene que hacer.
- Es fácil de instalar.
- Es fácil de usar.

Los defectos evitan que el software cumpla con los 3 puntos anteriores.

6.1. La Administración de Defectos

Es:

- Entender los defectos que uno inyecta.
- Encontrar y corregir eficientemente los defectos inyectados.
- Prevenir la inyección de defectos.

Tenemos que tener un mecanismo que nos permita darnos cuenta de los defectos que estamos inyectando en los diversos productos de software que desarrollamos, que nos permita entender el porqué los inyectamos, es decir, saber cuáles son las causas de inyección, que nos permita darnos cuenta lo más pronto posible de la presencia de los defectos y nos permita corregirlos de la manera más económica.

6.1.1. Pero, ¿qué es un defecto?

Un defecto es cualquier cosa que impida que el software haga lo que tenga que hacer.

Podemos mencionar como ejemplos, errores de sintaxis, error de interpretación de los requisitos, error al momento de teclear el código, etc.

6.1.2. Defecto vs. Error

Los errores son las acciones que las personas realizan que dan como resultado un defecto.

¡Las personas cometen errores (causa), los programas tienen defectos (resultado)!

Debemos cambiar la manera de trabajar para eliminar las causas de tal modo que los defectos puedan ser corregidos o eliminados de la manera más eficaz y eficiente posible.

6.1.3. Remoción vs. Prevención

Remoción de defectos:

- Encontrar y corregir.
- Proceso costoso.

Prevención de defectos:

- Reduce los errores que uno comete.
- Necesita cambios a la manera de trabajar.

Son dos conceptos diferentes pero complementarios para asegurar la calidad de los productos de software. Con la remoción asumimos que existen defectos, los cuales deben ser localizados y removidos, es costoso pues te nemos que buscar en todo el producto lo cual puede consumir mucho tiempo. No es tan fácil aprender, distinguir los errores que los inyectaron, pues muchas de las veces estamos más preocupados por eliminar los defectos que por entender el porqué de sus causas.

La prevención requiere tiempo de análisis, el averiguar las causas, motivos que dan pie a que se inyecten defectos, una vez entendidas las causas implica trabajar para eliminarlas, lo cual se traducirá en una manera diferente de trabajar.

6.1.4. La Calidad del Software y los Defectos

La remoción de defectos es un proceso que no termina, es decir, no podemos estar completamente seguros de que un defecto tiene ceros defectos.

La Administración de los Defectos es cara, impacta tanto a las ganancias como a la credibilidad de las organizaciones.

Nunca podremos estar seguros, completamente seguros, que un producto estará libre de defectos, pudiéramos probar que cierto tipo de defecto no está presente, pero el dominio de casos de prueba es tan grande que sería económicamente imposible probarlos todos.

6.1.5. Costos de Encontrar y Corregir Defectos

Algunos estudios indican que este costo se incrementa en una proporción de 10 a medida que un defecto avanza de fase.

Entonces sí tiene sentido corregir los defectos lo más pronto posible, más aún, prevenir la inyección

Por ejemplo, supongamos que se inyecta un defecto en la fase de requerimientos, si se descubriera en la fase de diseño tendría un costo de 10, en la fase de codificación de 100, y así sucesivamente.

6.1.6. ¿Quién debe remover los defectos?

Inicialmente quien los inyectó. Cada Ingeniero de Software es responsable de la calidad de su trabajo.

¿Qué significa esto?, pues que los ingenieros de software deben tener un mecanismo sistemático, un proceso para administrar los defectos.

Existen 3 técnicas reconocidas para encontrar defectos, las revisiones personales, las caminatas y las inspecciones. Las revisiones personales como su nombre lo indica, es el proceso que cada ingeniero de software debe seguir para remover los defectos que haya podido inyectar en sus productos. Más adelante en este curso se explicarán las 3 técnicas.

6.1.7. El Proceso de Administración de Defectos

Debemos:

1) Clasificar los defectos.
2) Darles seguimiento.
3) Priorizar los defectos que inyectamos.
4) Analizar las causas para prevenirlas.
5) Implementar las estrategias de prevención.

Aquí es bueno recordar el ciclo de mejora continua de la calidad estudiado en una lección anterior.

Debemos mantener un registro de cada defecto que encontremos.

Registrar la mayor cantidad de información para poder determinar sus causas. A medida que incrementamos nuestra base de datos de defectos, podremos darnos cuenta de cuáles defectos cometemos con mayor frecuencia y cuales cuestan más.

Analizar la información recolectada para determinar con certeza cuáles son los defectos que causan más problemas.

Finalmente encontrar maneras para descubrir y corregir los defectos en el producto y para evitar inyectarlos.

6.1.8. Clasificación de Defectos

Esta clasificación debe estar basada en la criticidad de las causas que los inyectan. La clasificación que seguiremos es:

Número de Tipo	Nombre del Tipo	Descripción
10	Documentación	**Problema**: La documentación, comentarios o mensajes no se entienden o están mal. **Corrección**: Corregir la documentación, los comentarios o los mensajes.
20	Sintaxis/Estática	**Problema**: Un defecto que puede usualmente ser detectado por el compilador (errores de sintaxis, declaraciones faltantes, errores de dedo, formatos de instrucciones). **Corrección**: Corregir la sintaxis o la semántica estática de las instrucciones defectuosas.
30	Construcción / Paquete	**Problema**: Errores en el control de versiones o en la administración del cambio, como por ejemplo en la reutilización o en las bibliotecas. **Corrección**: Crear o usar la versión correcta o corregir el cambio.
40	Asignación	**Problema**: Una sentencia contiene defectos (operadores incorrectos, expresiones incorrectas, asignación de objetos erróneos, nombres duplicados, límites, alcance de variables). **Corrección**: Corregir la sentencia.

Número de Tipo	Nombre del Tipo	Descripción
50	Interfase	**Problema**: Mal diseño o mal uso de las interfases (la interfase de clase, procedimiento o tipo de dato está incompleta o es errónea o no es utilizada apropiadamente). **Corrección**: Cambiar la interfase.
60	Chequeo	**Problema**: El manejo de excepciones y errores es manejado mal o no está presente. **Corrección**: Añadir o corregir el manejo de excepciones y/o errores.
70	Datos	**Problema**: La estructura o contenido de los datos es errónea. **Corrección**: Corregir la estructura o contenido de los datos.
80	Función	**Problema**: Defectos más allá de una sola sentencia en algoritmos o funcionalidad (algo hecho muy pronto o muy tarde, algo hecho en forma equivocada, algoritmo incorrecto, una funcionalidad mal diseñado o faltante, ciclos infinitos, recursión). **Corrección**: Añadir o corregir más de una sentencia.
90	Sistema	**Problema**: Problemas de temporización, sincronización, red, hardware, memoria, configuración o algo parecido. **Corrección**: Corregir el problema o cambiar la configuración.
100	Ambiente	**Problema**: Defectos en el ambiente de desarrollo o en los sistemas de soporte (compiladores, herramientas de diseño, datos para pruebas, etc.). **Corrección**: Corregir los defectos de los sistemas de apoyo o evitar ambientes de desarrollo defectuosos.

En un documento anexo se encuentra la clasificación de los defectos con la explicación y una posible manera de corregir cada uno de ellos.

A medida que vayamos entendiendo los defectos que cometemos podemos refinar nuestra clasificación.

¡No confundir las causas con los tipos de defectos!

Por ejemplo, de Sintaxis podemos abrir una clasificación para los que tienen que ver con errores de dedo y otra para los que denoten que no conocemos la sintaxis de cierta función propia del lenguaje o herramienta que estamos usando.

Entendiendo los Defectos

Las acciones para encontrar los posibles defectos que inyectamos y el hecho de encontrarlos son sólo el inicio de la mejora de nuestro proceso.

Pudieran escapársenos defectos, que posteriormente pudieran ser encontrados por otros.

Hay que aprender de los defectos que encontramos en nuestros programas, pero más importante, hay que aprender de los defectos que otros encuentran en nuestros programas, esto incrementará nuestra mejora.

¡La falla más grande es fallar en aprender de lo que hemos fallado!

Forma de Registro de Defectos

Tipos de defectos	
10 Documentación	60 Chequeo
20 Sintaxis	70 Datos
30 Construcción/Paquete	80 Función
40 Asignación	90 Sistema
50 Interfas e	100 Ambiente

Fases
1 Compromiso y Solución (Planeación y Diseño)
2 Implementación (Codificación)
3 Calidad y Liberación (Compilación, Pruebas y PostMortem)

Nombre:__________________________________Fecha: ______________________

Proyecto: _______________________________________

Fecha	Número	Tipo	Inyectado en Fase	Removido en Fase	Tiempo de Corrección	Defecto Corregido

Descripción: ____________________________________

Fecha	Número	Tipo	Inyectado en Fase	Removido en Fase	Tiempo de Corrección	Defecto Corregido

Descripción:

Fecha	Número	Tipo	Inyectado en Fase	Removido en Fase	Tiempo de Corrección	Defecto Corregido

Descripción: ____________________________________

Fecha	Número	Tipo	Inyectado en Fase	Removido en Fase	Tiempo de Corrección	Defecto Corregido

Descripción:

Fecha	Número	Tipo	Inyectado en Fase	Removido en Fase	Tiempo de Corrección	Defecto Corregido

Descripción: ____________________________________

Fecha	Número	Tipo	Inyectado en Fase	Removido en Fase	Tiempo de Corrección	Defecto Corregido

Descripción:

Fecha	Número	Tipo	Inyectado en Fase	Removido en Fase	Tiempo de Corrección	Defecto Corregido

Descripción: ____________________________________

Fecha	Número	Tipo	Inyectado en Fase	Removido en Fase	Tiempo de Corrección	Defecto Corregido

Descripción:

Fecha	Número	Tipo	Inyectado en Fase	Removido en Fase	Tiempo de Corrección	Defecto Corregido

Descripción: ____________________________________

Instrucciones para la Forma de Registro de Defectos

Propósito	Esta forma mantiene los datos de cada defecto que es encontrado y corregido. Los datos son utilizados para completar la Forma de Resumen del Plan del Proyecto.
General	• Registrar en esta forma todos los defectos encontrados en revisiones, compilaciones y pruebas. • Registrar cada defecto por separado y de forma completa. Si se necesita espacio adicional, utilizar otra copia de la forma.
Encabezado	Registrar los siguientes datos: • Tu nombre. • La fecha de hoy. • El nombre del proyecto. • El nombre del módulo/componente.
Fecha	Registrar la fecha en que es encontrado el defecto.
Número	Registrar el número de defecto. Para cada programa, este debe ser un número secuencial comenzando por ejemplo con 1 ó 001.
Tipo	Registrar el tipo de defecto tomándolo de la tabla de estándares de defectos, la cual se encuentra resumida en la parte superior de la Forma de Registro de Defectos.
Inyectado en Fase	Registrar la Fase en la cual fue inyectado el defecto. Utiliza tu mejor juicio.
Removido en Fase	Registrar la fase en la cual el defecto fue removido. Esto generalmente será la fase en la cual encuentras el defecto.
Tiempo de Corrección	Registrar el tiempo que te tomó corregir el defecto. Utiliza tu mejor juicio. Este tiempo puede ser determinado utilizando un cronómetro.
Defecto corregido	Si inyectaste este defecto mientras corregías otro, registra el número del defecto que estabas corrigiendo. Si no puedes identificar el número de defecto, registra una X.
Descripción	Escribe una descripción del defecto que sea clara para que posteriormente puedas utilizarla para saber porqué cometiste el error.

Fase de Inyección: fase en la cual se comete (inyecta) el defecto.
Fase de Remoción: fase en la cual se encuentra y remueve el defecto.

De ahora en adelante, no debemos posponer la solución, **¡defecto encontrado, defecto removido inmediatamente!**

En un archivo anexo se presenta la forma de registro de defectos, contiene la explicación de uso.

6.1.9. ¿Cómo saber qué registrar como defecto?

Una vez que se ha finalizado una tarea y se descubre que contiene defectos, cada uno de estos defectos debe ser contabilizado.

Una corrección que hagamos en una tarea que no hayamos terminado no se cuenta como defecto (una corrección al diseño mientras seguimos en el diseño no es un defecto).

Sin embargo si estamos en la codificación (implementación), y un defecto encontrado requiere un cambio en el diseño, entonces fue un defecto inyectado en el diseño (compromiso y solución).

Este punto es importante, está relacionado con entender las fases de desarrollo (requerimientos, diseño, codificación, pruebas, etc.).

¿Debo registrar todos los defectos?

No, si lo que quieres es aparentar, hacerte tonto a ti mismo.

Pero si estás verdaderamente comprometido con mejorar no debes evitar registrar todos y cada uno de los defectos que encuentres.

Tip: Piensa sobre el tipo de defecto más apropiado para clasificar tus defectos, más aún, describe a detalle el síntoma y la causa del error.
Sólo teniendo registrada toda la información podremos saber por dónde mejorar (tiempo, costo, calidad).

6.1.9.1. ¿Cómo encontrar los defectos?

Hasta ahora:

- En la fase de Implementación, utilizando el compilador.
- En la fase de Calidad y Liberación, realizando las pruebas.
- Después de la liberación, cuando los usuarios comiencen a quejarse.

El compilador sólo encontrará defectos de sintaxis, algunos que caigan en la clasificación de interfase y de ambiente. Este es un conjunto reducido de defectos, además, el hecho de dejar a una herramienta descubrir nuestros "h" errores, posibilita que no tomemos conciencia de estos tipos de defectos y evitemos encontrar estrategias para no cometerlos en el futuro.

Tradicionalmente la fase de pruebas se ha manejado como sinónimo de calidad, pero como hemos descrito, nunca estaremos seguros de que efectivamente un programa está libre de defectos pues del dominio de solución, datos que puede manejar un programa y caminos para solucionar, puede ser extremadamente caro y económicamente no factible.

6.1.9.2. Densidad de Defectos

La densidad de defectos es definida como el número de defectos removidos en un proyecto dividido por el tamaño del proyecto.

Ejemplo:

> 8 defectos removidos
> 120 LOC de tamaño final

Densidad de defectos = 8/120 = 0.06 defectos/LOC = 66.66 defectos/KLOC
Generalmente esta medida viene expresada por cada 1000 LOC = 1KLOC

6.1.9.3. Eficiencia de Remoción

La eficiencia de remoción está definida como el número de defectos removidos en cada fase dividido por el tiempo empleado en la fase.

Ejemplo:

> 5 defectos removidos en Calidad y Liberación.
> 30 minutos empleados en Calidad y Liberación.

ERD Calidad y Liberación = 5/30 = 0.16 defectos/minuto = 10 defectos/hora

El ERD generalmente se expresa en horas, multiplicar por 60

6.1.9.3.1. Rendimiento de la Remoción (yield)

Del proceso: El % de defectos removidos antes de la fase de Calidad y Liberación (propiamente antes de la compilación).

De una fase en particular: % de defectos removidos del producto en esa fase.

Objetivo: un yield del 100% (Todos los defectos removidos antes de la primera compilación).

La relación de defectos removidos en la fase analizada respecto al total de defectos removidos.

¿Hay una forma más económica y eficiente?

Sí, la prevención tema de la siguiente unidad.

Ejercicio de programación

Modifica el programa anterior para:

- Decir cuántos elementos estaban en la misma posición en las listas originales l1 y l2.
- En caso de existir elementos repetidos en l1 y l2, obtener la suma de estos elementos y guardar el total en l3 eliminando los elementos originales.

Entrada
l1 = {23,12,4,6,13,9,56}
l2 = {5,43,23,6,27,8,25}
Salida
l3 = {4, 5, 8, 9, 12, 13, 23, 25, 27, 43, 46, 56}

Elementos repetidos 6 y 23
Elementos repetidos en la misma posición 1 (6)
6+6 = 12
23+23 = 46

Sólo aparece una vez el 12 pues en l3 no se pueden repetir elementos

Nota: Los alumnos deberán reportar el resultado de sus pruebas.

Orden de entrega de los documentos

- Forma de Registro de Compromisos.
- Plantilla de Planeación de Tareas.
- Plantilla para la Planeación de la Agenda.
- Forma de Registro de Actividades.
- Forma de Registro de Defectos.
- Código Fuente del Programa.
- Estándar de codificación.
- Estándar de conteo.
- Pantallas de la interfase gráfica.
- Pantallas de los resultados.

Nota: El orden de entrega de la documentación es importante, si no se cumple, disminuir puntos de la calificación.

Forma de Registro de Compromisos Unidad 6

Nombre:_____Fecha:_____
Proyecto:_____
Lenguaje de programación: _____

Tamaño del Progama (LOC)	**Planeado**	**Actual**	**A la fecha**	
Total de LOC físicas				

Densidad de defectos		**Actual**		
Densidad de defectos				

Rendimiento de remoción de defectos		**Actual**		
Yield				

Tiempo en Fase (min.)	**Planeado**	**Actual**	**A la fecha**	**% a la fecha**
Compromiso y Solución				
Implementación				
Calidad y Liberación				
Total				

Productividad				
		Total de LOC físicas / Total tiempo	Total de LOC físicas / Total tiempo	

Defectos Inyectados		**Actual**	**A la fecha**	**% a la fecha**
Compromiso y Solución				
Implementación				
Calidad y Liberación				
Total en el Desarrollo				

Defectos Removidos		**Actual**	**A la fecha**	**% a la fecha**
Compromiso y Solución				
Implementación				
Calidad y Liberación				
Total en el Desarrollo				
Después del Desarrollo				