Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Guadalajara

Escuela de Graduados en Ingeniería y Arquitectura (EGIA)

Maestría en Ciencias Computacionales

áéíóúÁÉÍÓÚñÑüÜ

AUTOR: **Marco Antonio Rangel Bocardo**

ASESORES: Dr. Óscar Mondragón

Guadalajara (Jal), 11 de Septiembre de 2011

Dedicatoria

A mis padres,

a mis hermanos,

y a mis entrañables

amigos

Agradecimientos

Quizá no estaba en plenitud de mis sentidos en el momento que decidí iniciar una Maestría en Ciencias de la Computación. Definitivamente se necesita un poco de locura, y un tanto más de agallas para embarcarse en una maestría con enfoque científico, en la que tengas que crear y defender una tesis para obtener el grado. Esto se pone un poco más interesante si le agregamos el hecho de tener un trabajo de tiempo completo y otro de medio tiempo mientras se cursa la maestría. Cuando se ve esta perspectiva de una forma más general, se comprende que esto no lo hubiera logrado yo solo, y que, de alguna manera u otra, se ha llegado al objetivo gracias al apoyo de varias personas a las que quisiera agradecer personalmente y en esta dedicatoria.

Primero que nada quiero agradecer al Doctor Óscar Mondragón, el director de esta tesis, por todo el apoyo que me ha brindado durante este trabajo de un año. El decidió desinteresadamente apoyarme, y me ayudó a transformar una serie de ideas en un producto completo y palpable. Con sus conocimientos y experiencias personales y profesionales, fue una gran guía durante toda esta travesía.

También es muy importante mencionar a mis compañeros que elaboraron junto conmigo el sistema que es la base del estudio. Eduardo Campos y Humberto García, aparte de ser dos estudiantes sobresalientes, son aún mejores como personas, siendo destacados por tener valores como la responsabilidad, la honestidad, el trabajo duro y un sentido de compromiso como pocas personas en nuestro país.

Quisiera agradecer también a mis padres y hermanos, Antonio Rangel, Patricia Bocardo, Julia Rangel, Bryan Rangel y Ángel Rangel, por todo el apoyo que me brindan en cada uno de mis proyectos, y aunque no hayan cooperado de forma directa con el trabajo, su apoyo moral siempre ha sido muy importante para lograr lo que me propongo.

Cuento con la suerte de tener muchos y excepcionales amigos, así que no quería perder la oportunidad de mencionarlos, ya que ellos hacen que mi vida sea más plena y esté llena de experiencias divertidas. Agradezco mucho a: Guille Suro, Naiv Soto, Jafet Rodríguez,

Humberto García, Héctor Solís, Julián Hernández, Leonel Haro y Joaquín Soto; por todos los buenos momentos, las risas y las aventuras que hemos vivido.

Finalmente quisiera agradecer de forma especial a Alejandro Vázquez. Él nos brindó apoyo técnico y conocimientos durante la elaboración del sistema, todo de forma desinteresada, sin recibir nada a cambio, y siempre con una actitud de ayuda y servicio.

Resumen

Este trabajo de Tesis presenta el sistema de administración que se llamará "Bug Manager"(BM). Con este sistema se demuestra como la implementación de conceptos simples y claves de calidad dentro del proceso de desarrollo de software en empresas medianas y pequeñas, ayudan a mejorar sustancialmente los tiempos de entrega, la calidad final del producto y la satisfacción de los clientes. Algunos de estos conceptos son:

- Registro de actividades con su esfuerzo y tamaño dedicados;
- Registro y seguimiento de defectos encontrados;
- Realización de revisiones de código;
- Medición de la productividad personal y global de la empresa.

Con estos conceptos en mente, se propone la creación del BM, el cual será utilizado por cualquier empresa que busque implementar técnicas de control de calidad en su proceso de desarrollo de software. Este producto va enfocado inicialmente a pequeñas y medianas empresas, principalmente mexicanas. Esto último en virtud de que es el mercado preferente al cual se pretende llegar.

El BM permitirá que las empresas tener control sobre el avance y la calidad de sus proyectos, por medio de las siguientes estrategias:

- Guía en la elaboración del plan de calidad.
- Definición del ciclo de vida y actividades de desarrollo.
- Registro y seguimiento de actividades de aseguramiento de calidad.
- Registro y seguimiento de defectos.
- Generación de estadísticas personales, por proyecto, por equipo y por empresa.

Finalmente, durante el desarrollo del BM se pondrán en práctica las actividades de calidad mencionadas, y se hará un análisis del costo de la calidad para comprobar la efectividad de estas actividades en la mejora del proceso de desarrollo de software.

Contenido

Lista de Tablas

Lista de Figuras

CAPÍTULO 1

Introducción

1.1 Antecedentes

Para buen número de consultores (CITAR) de las empresas mexicanas de desarrollo de software, el común denominador de las empresas medianas y pequeñas, es no tener una administración de la calidad en el proceso de desarrollo.

En este tipo de situación, las empresas suelen asumir que las prácticas de calidad agregan trabajo extra, haciendo más lento y complicado el proceso de desarrollo. Esto provoca el atraso en los calendarios y la entrega tardía de los productos respectivos..

Estas decisiones provocan exactamente el efecto contrario. Al no tener un apropiado control de la calidad de sus productos, se ven envueltos en las siguientes situaciones:

- Cuando se llega a la fase de pruebas, el producto está plagado de defectos; lo que ocasiona que la fase tome la mitad del tiempo total de desarrollo, haciendo para muchos tortuosa esta etapa;
- Una vez que el sistema sale a producción, no está garantizado que el producto no tiene defectos. Muchos de estos fueron generados al momento de hacer las correcciones, o simplemente no se encontraron;
- Cuando los usuarios encuentran defectos en el producto final, lo usual es hacer la correción de estos defectos. Corregir un evento en etapa de pruebas cuesta normal-

mente diez veces más de lo que costaría en la fase de codificación; tanto como corregir un defecto en la fase mantenimiento cuesta cien veces más que hacerlo en la fase de codificación (ESCRIBIR LA CITA). Ambas generan altos costos de mantenimiento los cuales suelen ser absorbidos por la empresa que desarrolló el sistema.

- Peor aún que el incremento de los costos de mantenimiento, el cliente tiene un producto defectuoso, el cual no le permite realizar las actividades requeridas, generando desconfianza en la empresa de desarrollo, además una mala imagen y pérdidas de clientes en el futuro.

Todas estas situaciones pueden ser evitadas con una correcta administración de la calidad en el proceso de desarrollo de software.

Lo que muchas empresas no tienen en cuenta es que la calidad en el desarrollo debería ser la prioridad en el proceso de elaboración de productos de software. La calidad ayuda a que los productos en desarrollo sean predecibles en tamaño y calendario, fáciles de dar seguimiento y bajo costo de mantenimiento (CITA).

En vez de hacer más largo, complicado y costoso el proceso de desarrollo, la administración de la calidad recorta los tiempos de desarrollo, reduciendo considerablemente la fase de pruebas, disminuyendo el ciclo de vida del proyecto. La reducción del tiempo total de desarrollo se traduce en ahorro de costo en el proyecto. Si a lo anterior agregamos un menor número de defectos en las etapas de pruebas y producción, el cliente tendrá un producto de calidad, y la empresa de desarrollo mejorará imagen y su perspectiva de mercados a futuro.

1.2 Planteamiento del Problema

Normalmente las empresas medianas y pequeñas piensan que la administración de la calidad (y muchas veces hasta la más básica administración de software) es exclusiva de las empresas trasnacionales con grandes recursos y procesos bien definidos, normalmente con modelos de calidad mundialmente reconocidos. Tienen la idea de que la implementación de un plan de calidad, así como sus actividades pertinentes y propio seguimiento, es una tarea

que agrega más trabajo y complejidad al proyecto, lo que los haría salirse de calendarios y no cumplir con las fechas acordadas con el cliente.

Entonces la problemática de las empresas medianas y pequeñas de desarrollo de software es la falta de calidad en los productos que elaboran. Esta problemática se origina al no tener procesos definidos de desarrollo, COMPLETAR PROBLEMÁTICA

1.3 Propuesta de Solución

A partir de los antecedentes y la problemática descrita en el punto anterior, se pretende crear una herramienta llamada BM (por sus siglas en inglés Bug Manager), que ayude a las pequeñas y medianas empresas a la implementación y seguimiento de un plan de calidad, así como las actividades que se requieran realizar. Esta herramienta ataca el nicho de estas pequeñas y medianas empresas las cuales no cuentan con procesos de calidad.

El BM pretende ayudar a las empresas a:

- Establecer un ciclo de desarrollo.
- Elaborar un plan de calidad estableciendo objetivos y técnicas de detección de defectos para cada fase del desarrollo.
- Ayudar con plantillas que sirvan como guías de las técnicas de detección de defectos.
- Dar un seguimiento apropiado a los defectos encontrados durante el desarrollo del sistema.
- Generación de estadísticas y reportes los cuales mostrarán información valiosa acerca del desarrollo como: Productividad, Densidad de Defectos, Retorno de Inversión de las Actividades de Calidad, entre otras. Estas proporcionarán a las empresas información importante acerca de su proceso de desarrollo, mostrando cuáles son sus áreas fuertes y en cuáles hay oportunidad de mejora.

También se pretende que el BM genere una actitud de calidad total y mejora continua en las empresas que lo utilicen, y lograr un cambio cultural evolutivo:

- Promoviendo una cultura de calidad personal en el programador contra una cultura de calidad asignada a grupos organizacionales ajenos al desarrollo (pruebas, adherencia a procesos).
- Estableciendo una meta en el programador/grupo de desarrollo de cero defectos en pruebas de unidad contra número de componentes programados por hora.
- Promoviendo la prevención de defectos en lugar de la búsqueda de defectos durante las pruebas.
- Enfocando el esfuerzo de técnicas de detección de defectos al inicio del ciclo de vida en lugar de crecer los grupos dedicados a las pruebas al final del ciclo de vida.
- En resumen, promover un compromiso personal a la calidad del desarrollo de software y a las actividades asociadas para su mejora continua.

En la actualidad existen herramientas y programas de software que realizan tareas similares a las que realizará el sistema propuesto. Principalmente estos sistemas se dedican al registro y rastreo de defectos, así como al registro de las actividades realizadas dentro del ciclo de desarrollo, como una especie de bitácora. En general, consideramos que estos sistemas atacan una parte del problema y comúnmente carecen de la funcionalidad necesaria para agregar verdadero valor al proceso y método de desarrollo, debido a que se centran ya sea en el seguimiento de actividades o en el registro de defectos, pero no conjuntan ambas vistas de la problemática, aparte de que no generan estadísticas acerca de la información recabada.

El sistema propuesto pretende, al igual que las otras herramientas dar un seguimiento apropiado a los defectos, así como el establecimiento y la guía de un plan de calidad, finalmente generando estadísticas y reportes de todos los datos recabados.

1.4 Objetivos

Los objetivos del Trabajo de Tesis son:

- Realizar una investigación y análisis de los factores que determinan la calidad en el proceso de desarrollo de software, y demostrar el papel clave de la calidad en el desarrollo de software.
- Realizar una investigación y análisis del Costo de la Calidad en el proceso de desarrollo de software. Comparación del costo de la calidad contra el costo de la no calidad. Análisis del retorno de inversión de las distintas técnicas y prácticas de calidad.
- Realizar la propuesta de la herramienta BM, en las partes que conciernen al Costo de la Calidad.
- Construcción de la herramienta BM, en las partes que conciernen al Costo de la Calidad.
- Análisis de los resultados obtenidos en el proceso de construcción de la herramienta BM, que incluye el análisis de Costo de la Calidad contra Costo de la No Calidad, y análisis del Retorno de Inversión de las actividades y prácticas implementadas.
- Que la herramienta BM tenga las siguientes características mínimas:
  - Generar estadísticas y métricas de valor para la empresa y el personal en base a la información proporcionada por los usuarios del sistema.
  - Dar una guía en los procedimientos principales de aseguramiento de la calidad.
  - Optimizar y hacer más eficiente el proceso de desarrollo de software promoviendo actividades de prevención de defectos y análisis de datos.

1.5 Alcances

Los alcances del Trabajo de Tesis son:

- Investigación del Costo de la Calidad en el proceso de desarrollo de software.
- Propuesta y construcción de la herramienta BM, en las partes que conciernen el Costo de la Calidad.

1.6 Contribuciones

Las contribuciones del trabajo de tesis son:

- Proporcionar una herramienta flexible y efectiva para realizar las diferentes actividades de calidad, que permita eventualmente cambiar la manera en la que se elaboran este tipo de herramientas hasta el día de hoy.
- Que la herramienta elaborada colabore en la mejora continua de los productos de software desarrollados, por las diferentes empresas que adopten la herramienta como parte de su ciclo de desarrollo.
- Colaborar con la industria mexicana y latinoamericana de desarrollo de software, especialmente en las pequeñas y medianas empresas, a generar una cultura de calidad que ayudará a atraer más proyectos a la industria, generando así una mejora económica en la región.

CAPÍTULO 2

Marco Teórico

2.1 Nociones Básicas de Calidad de Software

En esta sección se darán las nociones básicas que se deben de conocer en el tema de la Calidad de Software antes de poder hablar de temas más específicos.

2.1.1 Definición de Calidad de Software

El concepto de Calidad de Software ha sido definido de varias formas por distintos autores. A continuación presento algunas de las definiciones más destacables:

- "Calidad significa cumplir con los requerimientos" - [1];
- "Calidad consiste en las características de los productos que cubren las necesidades de los clientes produciendo satisfacción gracias al producto" - [2];
- "Calidad consiste en la libertad de deficiencias" - [2];
- "Calidad de Software es: El grado en que un sistema, componente o proceso cumple con los requerimientos especificados" - IEEE 1991;
- "Calidad de Software es: El grado en que un sistema, componente o proceso cumple con las necesidades o expectativas del usuario" - IEEE 1991;
- "Como la belleza, todos tienen su idea de que es la calidad" - [3].

Tomando en cuenta estas definiciones, hay autores que se inclinan por definir Calidad de Software en relación al cumplimiento de requerimientos, y otros que prefieren relacionar la calidad con la satisfacción final del cliente. Finalmente las empresas deben de buscar un balance apropiado entre ambos enfoques para lograr el éxito en su labor.

2.1.2 Atributos de Calidad

Los atributos de calidad son características no funcionales de un sistema. También se conocen como requerimientos no funcionales y son factores que determinan la calidad interna y externa de un producto.

Los principales atributos de calidad son[4]:

- *Funcionalidad.* Este atributo se refiere a que el sistema cubra adecuadamente las necesidades funcionales del cliente. Sus características específicas son: Precisión, interoperabilidad, y seguridad;
- *Confiabilidad.* Este atributo se refiere a que tan confiable es el sistema, a si el sistema va a responder efectivamente cuando es necesario. Sus características específicas son: Madurez, tolerancia a fallas y la habilidad de recuperarse a éstas;
- *Usabilidad.* Este atributo se refiere a que tan fácil, intuitivo y usable es el sistema para el usuario. Sus características específicas son: La facilidad con la que el sistema es entendido, que tan fácil se aprende a usarlo, operatividad y atractividad;
- *Eficiencia.* Este atributo se refiere al desempeño del sistema. Sus características específicas son: Desempeño en el tiempo y utilización de recursos;
- *Mantenibilidad.* Este atributo se refiere a como el sistema puede ser mantenido y mejorado en el tiempo. Sus características específicas son: Analizabilidad, modificabilidad, estabilidad y facilidad para ser probado;
- *Portabilidad.* Este atributo se refiere a como el sistema puede ser instalado en distintos ambientes. Sus características específicas son: Adaptabilidad, facilidad para ser instalado, coexistencia y reemplazabilidad.

2.2 Enfoques de la Calidad de Software

Existen tres principales enfoques bajo los cuales la industria busca mejorar la calidad en el desarrollo de software. Estos son: Adherencia a procesos, elaboración de pruebas y revisiones de producto.

2.2.1 Adherencia a Procesos

Un enfoque que se tiene para dar calidad a los sistemas de software se hace mediante la definición y adherencia a procesos. La idea de este enfoque es que las organizaciones tengan procesos bien definidos, y los sigan religiosamente. Esto con el objetivo de siempre poder obtener resultados similares en sus diferentes procesos.

Dos grandes ejemplos de este enfoque son el ISO9000 y CMMI.

ISO9000 es una familia de estándares relacionados a la administración de la calidad de los sistemas, y están diseñados para ayudar a las organizaciones a asegurarse que cumplen con las necesidades de los clientes y las demás partes interesadas. Los estándares que componen al ISO900 son [5]:

- *ISO9000*. Es la introducción al ISO 9000, se refiere a la selección y uso de los ISO9001-9004;
- *ISO9001*. Modelo para asegurar la calidad en el diseño, desarrollo, producción e instalación en las organizaciones;
- *ISO9002*. Modelo para asegurar la calidad en la producción, instalación y servicio. Está basado en el ISO9001 pero también incluye la parte de creación de nuevos productos;
- *ISO9003*. Modelo para asegurar la calidad en las inspecciones finales y el proceso de pruebas;
- *ISO9004*. Modelo para asegurar la calidad a través de situaciones preventivas.

CMMI (siglas en inglés de Capability Maturity Model Integration) es un enfoque de mejora de procesos que tiene como meta ayudar a las empresas a mejorar su desempeño. Este modelo contiene los elementos esenciales de los procesos efectivos, y describe un proceso evolutivo de mejora tanto para mejorar procesos inmaduros como para mejorar procesos maduros. Cuenta con 5 niveles de madurez que son los siguientes[6]:

- *Nivel 1 (Inicial).* Procesos impredecibles, poco controlados y que reaccionan a las situaciones;
- *Nivel 2 (Administrado).* Proceso caracterizado por estar organizado en proyectos, normalmente es reactivo a las situaciones;
- *Nivel 3 (Definido).* La organización tiene procesos definidos y es proactiva. Los proyectos se rigen mediante los procesos de la organización;
- *Nivel 4 (Administrado Cuantitativamente).* Los procesos de la organización son controlados y medidos;
- *Nivel 5 (Optimización).* Se enfoca en la mejora de procesos.

2.2.2 Pruebas de Software

Un segundo enfoque es aquel que está basado en la realización de pruebas para asegurar la calidad del sistema.

Las pruebas de software son un proceso de verificación y validación de un sistema de software. Estas actividades se realizan con el objetivo de encontrar defectos en un sistema de software. Con esta definición surgen tres conceptos [7]:

- *Verificación.* Es asegurarse que el sistema hace lo que el usuario final necesita que haga.
- *Validación.* Es asegurarse que las operaciones que realice el sistema sean correctas.
- *Defecto.* Es una diferencia entre el resultado esperado y el resultado obtenido.

Las pruebas de software son realizadas para encontrar los defectos en el sistema de software antes de que estos lleguen al usuario final.

Existen distintos tipos de pruebas de software, los más comunes son [7]:

- *Pruebas Unitarias*. Son pruebas ejecutadas a una unidad de sistema. Dependiendo del tipo de proyecto, una unidad puede ser considerada, un método dentro de alguna clase, un archivo de código fuente completo entre otras. En estas pruebas solo se asegura la funcionalidad independiente de la unidad.

- *Pruebas de Sistema*. Es ejecutar todas las pruebas unitarias en un escenario real. Se refieren a utilizar el sistema bajo condiciones normales donde las unidades interactúen entre sí.

- *Pruebas de Estrés*. Es ejecutar pruebas de sistema pero bajo condiciones de estrés, con una carga de trabajo muy alta y fuera de lo normal.

- *Pruebas de Aceptación de Usuario*. También conocidas como pruebas Beta, tratan de darle una versión funcional, aunque no propiamente final, del sistema al usuario final, para que estos se aseguren de que el sistema hace lo que ellos necesitan y lo haga de forma correcta.

- *Pruebas de Regresión*. Estas pruebas se ejecutan cuando se realiza la corrección de algún defecto. Todos los componentes o unidades que son afectados por este cambio tienen que ser probados de nuevo para asegurarse de que la corrección del defecto original no introdujo nuevos defectos.

Sin embargo el enfoque de este Trabajo de Tesis es justamente evitar hacer pruebas extensivas. Como se mencionó en la sección 1.1 del presente Trabajo de Tesis, gran parte de la problemática actual se debe a que los productos del proceso de desarrollo de software llegan con una calidad tan baja a la etapa de pruebas, que a las organizaciones de desarrollo normalmente les toma la mitad del tiempo total del proyecto terminar esta etapa. El Trabajo de Tesis se enfoca en la revisión y prevención, con el último objetivo de que los productos del proceso de desarrollo de software lleguen libres de defectos a la etapa de pruebas.

2.2.3 Revisiones de Productos de Trabajo

Este es el enfoque del Trabajo de Tesis. Las revisiones de los productos de trabajo son evaluaciones y revisiones de estos productos por parte de uno o más compañeros del equipo de desarrollo calificados para hacerlo.

La idea principal detrás de las revisiones es detectar los defectos lo antes posible en el proceso de desarrollo de software. Esto evita que los errores permanezcan durante el proceso de desarrollo, donde se va haciendo más difícil encontrarlos y corregirlos mientras avanza el proyecto.

Los tipos de revisiones de productos más comunes son: Revisiones personales, revisiones entre colegas, caminatas e inspecciones.

En la subsección 3.4 del presente Trabajo de Tesis se esbozarán distintas estrategias para promover la cultura de la revisión y la prevención, y en la subsección 3.4.2 se ahondará en el tema de las técnicas de detección de defectos.

2.3 Importancia de la Calidad en el Desarrollo de Software

El software es una tecnología sorprendente. Es un producto totalmente intelectual por lo que no tiene costos de producción, aparte puede ser distribuido mundialmente en segundos, no se deteriora con el tiempo y es la forma más económica y sencilla de implementar casi cualquier función compleja.

En todos los campos de la ingeniería y la ciencia, más de la mitad del tiempo de cualquier profesional lo utiliza desarrollando, mejorando, manteniendo o usando un sistema de software. Es sin duda uno de los negocios más grandes e importantes de la actualidad[8].

Tomando en cuenta lo anterior, muchos ejecutivos de varias organizaciones vieron las oportunidades de negocio en el software, sin embargo implementar un departamento de software efectivo no es una tarea trivial, y la calidad en los productos que se desarrollen es clave para el éxito.

2.3.1 El software en las organizaciones

Para administrar una organización o un departamento dentro de una organización dedicado al software se necesita tener en cuenta los siguientes principios de administración:

1. Reconocer la importancia del software en el negocio;

2. La calidad en los productos de software debe de ser la principal prioridad. La calidad es una elección con consecuencias económicas, si no se paga por la calidad al principio del desarrollo, se pagará una cantidad mucho más alta después. Para realmente lograr que los proyectos cumplan con los calendarios y los costos, el trabajo tiene que realizarse de forma correcta desde el principio;

3. El software de calidad se desarrolla por personas disciplinadas y motivadas. Si los profesionales del software no están convencidos y entrenados en métodos de calidad, no van a seguir las prácticas requeridas y no producirán software de calidad.

2.3.2 ¿Por qué los proyectos de software fallan?

La razón principal por la que los proyectos fallan es administración inadecuada. Una buena administración requiere dos cosas: Estar enfocados en la calidad, e ingenieros motivados que realicen un trabajo disciplinado.

Las causas más comunes por las cuales los proyectos fallan son[8]:

- *Calendarios poco realistas.* Cuando un proyecto de software inicia con calendarios poco realistas, el proyecto será entrado después de lo que se hubiera hecho con un calendario realista. Esto es porque los ingenieros comienzan a hacer trabajo rápido y de poca calidad para alcanzar el calendario. El resultado de esto son productos de baja calidad, los cuales están llenos de defectos, lo que se traduce a una extensa etapa de pruebas;

- *Equipo inadecuado.* La única forma de terminar un proyecto de software de manera rápida y efectiva es asignar el número adecuado de personas y protegerlas de interrupciones y distracciones;

- *Cambio de requerimientos.* Es normal que los requerimientos cambien en las fases iniciales del proyecto, sin embargo llega un punto que esta situación es muy perjudicial. Se tiene que identificar este punto para evitar pérdidas de dinero y cambios que afecten en sobremanera el trabajo que se está haciendo;

- *Baja calidad en el trabajo realizado.* Si el trabajo que se realiza es de baja calidad, va a ocasionar que la fase de pruebas sea muy larga y se gaste mucho tiempo arreglando defectos durante el proceso de desarrollo;

- *Creer en la magia.* No existe la bala de plata. Muchas veces se cree que con una nueva tecnología o forma de desarrollo se resolverán todos los problemas, y lo que ocurre generalmente es que nos pueden llevar a serios problemas.

El costo de una baja calidad de software es difícil de ver hasta el final del proyecto. Comúnmente, la baja calidad le dará problemas aún a los usuarios finales ya que el proyecto se haya terminado. Los errores más comunes son: líderes de proyecto que toman compromisos irresponsables y no insisten en que el trabajo se haga de forma correcta. Todos los problemas mencionados anteriormente se pudieron evitar si la administración hubiera insistido en planear correctamente el trabajo y realizarlo de forma disciplinada.

2.3.3 La Calidad es Negocio

Hay tres razones fundamentales para insistir en que la Calidad de Software tiene que ser medida y administrada[8]:

1. La baja calidad en el software puede causar daños severos a la propiedad, y en algunos casos puede matar personas;

2. El trabajo de calidad ahorra tiempo y dinero;

3. Si la gerencia y el líder de proyecto no insisten en la administración de la calidad de software, nadie más lo hará.

Varios estudios han demostrado que incluso los ingenieros experimentados inyectan un defecto cada diez líneas de todo[8]. Esto no significa que los programadores sean incompetentes, simplemente son humanos.

El costo de encontrar y corregir los defectos se incrementa en cada fase del proceso de desarrollo de software. Entre más permanezca un defecto en el sistema, será más difícil removerlo.

Existen varias técnicas para remover defectos antes de llegar a la fase de pruebas. Varias de estas técnicas fueron mencionadas en la sección 3.2.3 del presente Trabajo de Tesis, y serán descritas a mayor profundidad en la sección 3.4.2. Una de estas técnicas por ejemplo, es la revisión de código, en las que se lee y revisa el código producido en busca de defectos. Estas actividades suelen tomar una pequeña cantidad de tiempo, y cada defecto encontrado ahorra una gran cantidad de tiempo en la fase de pruebas.

A diferencia de las revisiones de código, la fase de pruebas es una actividad de calidad mucho más larga. La razón de esto es que en las pruebas de software solo se revela los síntomas del defecto, mientras que en las revisiones se encuentra directamente el defecto.

La mejor manera de remover defectos de software es realizando revisiones de código, ya que estas son más baratas y más efectivas que las pruebas de software. El ingeniero que desarrolló el programa es el más indicado para encontrar y corregir sus propios defectos.

Aun conociendo la efectividad de las revisiones de código, muchas organizaciones no las utilizan. La razón es que para realizar revisiones de código efectivas se requieren métodos disciplinados.

Si los proyectos tienen la mayor parte de su tiempo dedicado a la fase de pruebas de software, va a ser casi imposible planearlo y darle seguimiento. Si se desea que los planes tengan compromisos precisos y realizables, se debe de insistir en realizar un trabajo de calidad.

2.4 Organizaciones con Procesos de Calidad

Existen varios factores que se tienen que tomar en cuenta en una organización que se dedique al desarrollo de software para que esta produzca sistemas de alta calidad.

2.4.1 3.4.1 Administración Racional

El principio de la administración racional es confiar que los miembros de un equipo de desarrollo de software son profesionales preocupados en el éxito del proyecto.

En la administración racional se necesita crear planes, seguir estos planes, y corregir los problemas antes de que se salgan de control. Sus cuatro elementos principales son[8]:

1. Establecer metas agresivas a largo plazo, pero descomponerlas en metas realísticas y medibles a corto plazo;

2. Insistir en la creación de planes e insistir que estos planes sean creados por las personas que harán el trabajo. Los planes tienen que ser detallados y completos, y tienen que ser revisados a conciencia en busca de omisiones y otros errores;

3. Se tienen que utilizar datos e información. Si se utiliza de forma objetiva los datos y la información del desempeño del equipo, se demuestra la confianza que se tiene en el equipo y la disposición que existe para escuchar sus problemas, planes e ideas;

4. Dar seguimiento al trabajo realizado y utilizar la información actual para anticipar y resolver problemas futuros. Los calendarios y planes se salen de control día a día.

Cuando el trabajo es planeado, medido y monitoreado, se puede analizar el desempeño del proyecto y del negocio.

2.4.2 Transformado las Organizaciones

Las organizaciones siempre quieren realizar sus actividades más rápido, de mejor manera y a menor costo. Un principio básico en la administración es: Lo que se mide puede ser administrado, y lo que es administrador puede realizarse. Al contrario, lo que no es medido comúnmente se ignora.

Las tres principales cosas en las que una organización se debe de concentrar para hacer el cambio de calidad son: Calendario, calidad y costo. Para lograr esto se tiene que cambiar la forma en que los ingenieros trabajan. Para acelerar el trabajo, te debes de enfocar a las actividades, como se hacen y que se necesita para acelerar el proyecto.

El primer requerimiento para ser más rápido, mejor y más económico es hacer planes detallados y comprensivos. Los ingenieros son quienes deben de realizar estos planes. Después la gerencia debe de asegurarse que los ingenieros tengan las habilidades y conocimientos necesarios para elaborar estos planes. Finalmente los jefes directos de los ingenieros, comúnmente los líderes de proyecto, deben de participar en la elaboración de los planes, y finalmente, la administración y la gerencia debe de revisar que estos planes sean completos.

El siguiente requerimiento es que los ingenieros deben de utilizar los planes que elaboraron para realizar el trabajo. Uno de los mayores beneficios de tener planes detallados es que los equipos pueden balancear las cargas de trabajo. La calidad es clave aquí, se necesita que todos los productos de trabajo tengan alta calidad, si no, eventualmente tendrán que ser corregidos y tendrán un costo extra.

Una meta de calidad en la organización debería ser que los ingenieros estén entrenados en las prácticas de calidad y que utilicen estas prácticas para realizar el trabajo.

La estrategia principal para reducir el costo de ingeniería es ayudar a los ingenieros a maximizar el tiempo trabajado en actividades que se encuentren dentro del plan de desarrollo. En el tiempo de actividades de desarrollo no se toma en cuenta el tiempo invertido en juntas, descansos, asesoría por parte de la gerencia u otras actividades requeridas dentro de los proyectos de ingeniería. En promedio los ingenieros invierten a las actividades planeadas entre diez y quince horas por semana.

Para maximizar el tiempo que se invierte en actividades planeadas existen cuatro elementos clave: Medir el tiempo invertido en las actividades planeadas, mantener a los ingenieros motivados, planear el tiempo a invertir en las actividades planeadas, y revisión y apoyo por parte de la administración.

Entonces para que una organización pueda mejorar su calidad las diferentes partes deben de cumplir con lo propuesto anteriormente. Los ingenieros deben de utilizar las prácticas de calidad, así como planear y dar seguimiento al trabajo realizado y finalmente medir y administrar la calidad del producto. Los líderes o administradores de proyecto deben iniciar y mantener este nuevo comportamiento, así como monitorear los planes y balancear efectivamente la carga de trabajo. La gerencia o líderes de la organización deben de proveer un ambiente disciplinado y atractivo de trabajo, establecer las metas a largo

plazo y dar seguimiento al trabajo realizado.

2.4.3 Cambiando el Comportamiento de los Ingenieros

La calidad del producto de software desarrollado está determinada por el proceso que se utilizó para crear el producto. La única forma de crear productos de software de calidad es implementar técnicas de calidad de software bajo un proceso disciplinado.

Para que los ingenieros sean capaces de crear planes detallados basados en información histórica, y sigan estos planes para hacer el trabajo, los ingenieros deben estar capacitados y convencidos que el realizar estas acciones va a fomentar el éxito en el proyecto.

La idea básica detrás de esto es que si las partes pequeñas del sistema no tienen alta calidad, el sistema completo tampoco será de calidad. Es por esto que los ingenieros deben de enfocarse en la calidad desde el principio del ciclo de desarrollo, teniendo cada componente por mínimo que sea, una calidad envidiable.

Todas las partes a cargo del desarrollo del proyecto son responsables de implementar los distintos métodos calidad. La gerencia y los líderes de proyecto son los responsables de implementar estos métodos de calidad y los ingenieros deben de seguir fielmente estos métodos. Toma el mismo tiempo escribir programas de alta calidad que lo que toma escribir programas plagados de defectos.

2.4.4 Construyendo Equipos Motivados

Los mejores resultados dentro de una organización se obtienen cuando la administración confía en sus ingenieros. En el contexto de negocios hay tres formas de motivar a las personas: Miedo, avaricia y compromiso. Solo en los trabajos más simples el miedo y la avaricia son motivadores efectivos.

Para crear equipos motivamos, se necesita que todos los ingenieros estén capacitados para realizar el trabajo que van a hacer. Después se tiene que convencer a los ingenieros de que el trabajo que van a realizar es importante y que se necesita su compromiso personal para que tenga éxito. Se tienen que plantear metas agresivas, pero crear planes detallados para alcanzarlas, si los planes creados por los ingenieros no cumplen la gerencia se tiene

que pedir a los ingenieros que justifiquen sus planes. Los planes tienen que ser revisados en busca de partes donde se pueda recortar tiempo, pero también deben de buscarse omisiones. Lo que se necesita es un compromiso realista que el equipo pueda cumplir, no una vana promesa.

El líder de proyecto debe de checar el progreso en el trabajo cada semana, y la gerencia al menos cada mes.

2.4.5 Beneficios del Trabajo de Calidad

Cuando se utilizan correctamente métodos disciplinados para planear, dar seguimiento y administrar el trabajo, los productos elaborados serán de alta calidad y dentro de calendario. Más importante aún, tendremos una noción bastante precisa de en qué parte del proceso de desarrollo se encuentra el proyecto, y una predicción fiable de cuándo se va a terminar.

El tiempo que se requiere para escribir programas de alta calidad es el mismo que se necesita para escribir programas de baja calidad. Los productos de calidad reducen el tiempo desarrollo. Entre menos defectos existan, el costo será menor y los estimados serán más efectivos. También el tiempo que se requiera para terminar la etapa de pruebas será mucho menor.

2.4.6 Pasos para Mejorar la Calidad

Esta subsección presenta un resumen rápido de la sección 3.4, dando los puntos principales para que una organización haga una transformación hacia la cultura de la alta calidad[8]:

1. Establecer una política de calidad. Se debe de tener claro que el trabajo de calidad es la primera prioridad. Las políticas de calidad deben de ser escritas y publicadas, haciendo énfasis en que hacer las cosas de forma correcta es la mejor forma de hacerlas;

2. La gerencia debe de tener en cuenta que son los responsables de la mejora de los procesos. Normalmente se opta por asignar una persona en particular para dar seguimiento a la mejora que se va teniendo;

Nivel de CMM	Defectos/Mil de Líneas de Código
1	7.5
2	6.24
3	4.73
4	2.228
5	8

Tabla 2.1: Número de Defectos por Nivel de CMM

3. Establecer metas precisas y medibles. Para alcanzar metas agresivas, las metas deben de ser claras y tener formas de medirlas para hacer comparaciones;

4. Los líderes de proyecto deben de ser responsables de la mejora de cada uno de los equipos a su cargo;

5. Se deben de proveer los recursos necesarios para realizar las mejoras;

6. Se deben de identificar las áreas de mejora, priorizarlas y enfocarse en las más importantes. Ya que se hayan mejorado las áreas de mayor prioridad se puede pasar a mejorar otras;

7. Se debe de dar seguimiento al proceso de mejora.

2.5 El Proceso Personal del Software

Los métodos tradicionales para asegurar la calidad del software en desarrollo son: Inspección de requerimientos, inspección de diseño, compilación, inspección de código y pruebas. La tabla 2.1 muestra los niveles de defectos promedio por las organizaciones según su nivel de CMM[9].

Para mejorar la calidad de productos de software, aparte de utilizar los métodos tradicionales de calidad, se debe de medir y dar seguimiento al trabajo personal. Si el objetivo es tener un sistema de software de alta calidad, cada parte del sistema debe de ser de alta calidad. La estrategia de PSP (por sus siglas en inglés Personal Software Process) es administrar los defectos contenidos en todas las partes del sistema. Con partes de alta calidad, el proceso de desarrollo de software se puede escalar sin perder productividad.

2.5.1 ¿Qué es la calidad de software?

La definición de calidad debería de basarse en las necesidades de los usuarios[9].

Los usuarios deben de contar con un producto funcional. Si el producto tiene muchos defectos este no se desempeñará con una consistencia razonable. Sin embargo esto no significa que los defectos son siempre la principal prioridad. Después de los defectos encontramos características de calidad como: Desempeño, seguridad, usabilidad, compatibilidad, funcionalidad, confiabilidad entre otras.

Normalmente se gasta un tiempo excesivo en la corrección de defectos, dejando muy poco tiempo dedicado a las otras características mencionadas anteriormente.

Aunque nos referimos que los defectos son solo una parte de la calidad del producto de software, son el principal objetivo del PSP, ya que la administración efectiva de estos es esencial para la administración de costo, calendario y los demás aspectos de la calidad de producto.

2.5.2 La Economía de la Calidad de Software

La calidad de software es un problema de calidad. Cada prueba realizada cuesta dinero y toma tiempo.

Entre más tiempo el defecto permanezca en el producto, el impacto es más alto. Por ejemplo encontrar un problema de requerimientos cuando el cliente ya está utilizando el producto puede ser demasiado costoso, en cambio encontrar el defecto durante una revisión de código será mucho menos costoso. El objetivo entonces es, remover los defectos lo antes posible en el proceso de desarrollo. Esto se logra haciendo revisiones e inspecciones de cada producto de trabajo lo más pronto posible de que se haya terminado el producto.

El proceso de pruebas puede ser muy efectivo para identificar problemas de desempeño, usabilidad y problemas operacionales, pero no es tan efectivo removiendo grandes cantidades de defectos. Los datos de estudios realizados demuestran que mientras más tarde en el proceso de desarrollo sea encontrado un defecto, es más difícil encontrarlo y removerlo.

Tipos de Defecto	Defectos/Mil de Líneas de Código
Requerimientos	2.3
Diseño	1.9
Codificación	0.9
Documentación	1.2
Correcciones Defectuosas	1.2
Total	7.5

Tabla 2.2: Tipos de Defectos por Mil Líneas de Código

2.5.3 Tipos de Defectos

Los tipos de defectos más comunes son: Errores en requerimientos, errores de diseño, errores de codificación, errores de documentación, correcciones defectuosas. La tabla 2.2 presenta el número de defectos cada mil líneas de código según el tipo de defecto[9].

2.5.4 Métricas de Calidad

Para mejorar la calidad, se debe de medir la calidad[9]. El PSP propone el uso de las siguientes métricas para medir la calidad de las organizaciones de software: El Yield de Calidad, el Costo de la Calidad (COQ), la Tasa de Revisiones, la Calidad de Radio de Fases y el Índice de Calidad de Proceso.

El Yield mide la eficiencia de cada fase removiendo defectos. El Yield de una fase es el porcentaje de defectos de producto totales que son removidos en esa fase. El Yield de proceso es el porcentaje de defectos que son removidos antes de la primera compilación. El Yield aumenta claramente cuando se comienzan a utilizar revisiones de diseño y de código.

El COQ tiene tres componentes principales:

1. *Costo de las Fallas (CF).* El costo total de las fases de compilación y de pruebas;
2. *Costos de Evaluación (CE).* El tiempo utilizado en revisiones de código y de diseño;
3. *Costos de Prevención.* El costo de identificar causas de defectos y acciones para prevenirlos en el futuro.

En la tabla 2.3 se muestran las métricas que se pueden obtener a partir del análisis del COQ[9].

Fórmulas COQ		
$CF = 100(\frac{TC+TT}{TTD})$	CF TC TT TTD	Costo de Fallas Tiempo de Compilación Tiempo de Pruebas Tiempo Total de Desarrollo
$CE = 100(\frac{TRD+TRP}{TTD})$	CE TRD TRP TTD	Costo de Evaluación Tiempo de Revisión de Diseño Tiempo de Revisión de Compilación Tiempo Total de Desarrollo
$TCOQ = CF + CE$	CE TTD	Costo de Evaluación Tiempo Total de Desarrollo
$CE\,\% = 100(\frac{CE}{TCOQ})$	CE TTD	Costo de Evaluación como % del TTD Tiempo Total de Desarrollo
$RF = \frac{CE}{CF}$	RF CE CF	Razón de Falla Costo de Evaluación Costo de Fallas

Tabla 2.3: Métricas del COQ

El Yield y el COQ son útiles, pero miden el trabajo que hiciste, no lo que estás haciendo. La Tasa de Revisión y al Radio de Calidad por Fase proveen una manera de dar seguimiento y control a los tiempos de revisión.

La tasa de revisión se utiliza principalmente para revisiones de código e inspecciones, y mide el número de líneas de código (LOC) o páginas revisadas por hora.

El Radio de Calidad por Fase, Es el radio de tiempo utilizado en dos fases de proceso. El radio para la revisión es el tiempo utilizado para realizar revisiones dividido entre el tiempo de desarrollo.

El Índice de Calidad de Proceso (PQI), es una métrica compuesta por cinco sub-métricas, la cual nos permite analizar el desempeño general de los procesos de una organización de software. El PQI se obtiene de multiplicar las cinco sub-métricas que se presentan en la tabla 2.4 y el objetivo es que el resultado sea 1.0.

Fórmulas PQI		
$PQI - DP = \frac{TD}{TP}$	PQI-DP TD TP	PQI de Diseño y Programación Tiempo de Diseño Tiempo Programación
$PQI - RD = 2(\frac{TRD}{TD})$	PQI-RD TRD TD	PQI de Revisión de Diseño Tiempo de Revisión de Diseño Tiempo de Diseño
$PQI - RP = 2(\frac{TRP}{TP})$	PQI-RP TRP TP	PQI de Revisión de Programación Tiempo de Revisión de Programación Tiempo de Programación
$PQI - DC = 20(10 + \frac{DC}{KLOC})$	PQI-DC DC KLOC	PQI para Defectos de Compilación por KLOC Defectos de Compilación Mil Líneas de Código
$PQI - DT = \frac{10}{5+\frac{DT}{KLOC}}$	PQI-DT DT KLOC	PQI para Defectos de Pruebas por KLOC Defectos de Pruebas Mil Líneas de Código

Tabla 2.4: Sub-Métricas del PQI

2.5.5 Administración de la Calidad del Producto

El proceso de pruebas ha crecido hasta que ahora consume cerca de la mitad del calendario de desarrollo. La estrategia consiste en solucionar esto removiendo tantos defectos como sea posible antes de entrar en pruebas. Esta estrategia mejora tanto la calidad del producto como la productividad en el desarrollo.

En los sistemas de millones de LOC debemos enfocarnos en tener menos de diez errores por cada MLOC.

2.5.6 Prácticas para la Mejora de Calidad del PSP

Existen seis principios que recomienda el PSP para la Mejora de la Calidad Personal[9]:

1. Para tener calendarios predecibles, se tiene que planear y dar seguimiento al trabajo personal;

2. Para hacer planes precisos y que se les pueda dar seguimiento, estos planes tienen que ser detallados;

3. Para hacer planes detallados, debes de basar los planes en datos históricos;

4. Para hacer un trabajo de alta calidad, debes de usar un proceso personal definido y medible;

5. Ya que el trabajo de mala calidad no es predecible, la calidad es un pre requisito para la predictibilidad.

2.5.7 Prevención de Defectos

Se debe de revisar los datos de los defectos que más se encuentran durante el desarrollo y las pruebas. Con esto se pueden establecer estrategias como enfocarse en los errores que:

- Se encuentran en el programa final o en el periodo de pruebas;
- Aquellos que ocurren más frecuentemente;
- Aquellos que son más difíciles o costosos de corregir;
- Aquellos en los que se pueden realizar acciones preventivas sencillas;
- Aquellos que más nos molestan.

2.5.8 Técnicas de Detección de Defectos

Las técnicas de detección de defectos pueden ser definidas como la evaluación y la revisión de un producto de trabajo por parte de uno o más compañeros de trabajo calificados para realizar la actividad[10].

Existen diferentes tipos de técnicas de detección de defectos, unas más formales y rígidas que otras. Evidentemente la diferencia radica en la forma de conducir la revisión del producto de trabajo. También existen diferentes clasificaciones acerca de las técnicas, pero creo que una clasificación coherente consiste en separar las técnicas que involucran a una sola persona y las técnicas en las que participan dos o más miembros del equipo. Los tipos más utilizados son: Revisión Personal, Revisión entre Colegas, Caminatas, Inspección

La revisión personal consiste en examinar el producto de trabajo antes de entregarlo a cualquier otro miembro del equipo, ya sea para su lectura, compilación, revisión, implementación o prueba.

Existen varias técnicas de detección de defectos que se realizan entre colegas y/o miembros de un equipo de desarrollo[11]. Estas técnicas se puede clasificar de acuerdo a su formalidad, de acuerdo con existen 6 diferentes tipos de técnicas:

- Revisión ad hoc;
- Revisión general;
- Revisión de parejas;
- Caminata;
- Revisión de equipo;
- Inspección.

La caminata es un tipo de revisión en el que precisamente una persona (preparada especialmente para ello) pasa a través del producto, exponiéndolo a una audiencia. Mediante esta técnica se pueden obviar muchos detalles, con lo cual se reduce el tiempo de revisión. Sin embargo, esto puede ser contraproducente si el objetivo es precisamente detectar defectos que residen en los detalles. Al mismo tiempo, el proceso de revisión está definido por el producto de trabajo siendo revisado, a diferencia de la inspección, en el que el proceso se determina por los puntos a revisar[12].

La Inspección es muy probablemente la técnica más formal de detección de defectos de software[11]. Consiste de los siguientes pasos[10]:

1. *Planeación*: Seleccionar dónde, cuándo y quiénes participarán;
2. *Resumen*: Revisión general para que los revisores se familiaricen con el producto de trabajo;
3. *Preparación*: Cada revisor lee el producto de trabajo e identifica posibles defectos. Todo esto generalmente mediante listas de chequeo. La persona que realiza el papel de líder, se prepara para la junta de revisión;

4. *Junta*: El líder modera la junta y realiza la revisión del producto. Los revisores lo interrumpen para discutir los defectos encontrados. Lo más importante es que NO se permite discutir posibles soluciones para ningún defecto;

5. *Re-trabajo*: El autor del producto realiza las correcciones pertinentes;

6. *Seguimiento*: El autor del producto notifica al líder de las correcciones y éstas son revisadas.

2.5.9 ¿Por qué Revisar los Programas?

Es la manera más rápida y más barata para encontrar y corregir problemas antes de diseñar una función equivocada o implementar un diseño incorrecto. Un poco de tiempo invertido revisando un programa puede ahorrar mucho tiempo durante las fases de compilación y pruebas. Aún más importante, cuando todos en el equipo de desarrollo hacen revisiones de diseño y de código a conciencia, el tiempo de integración y de pruebas de sistema es reducido en un factor de cinco a diez.

2.5.10 Principios de la Revisión

Los principios de las revisiones personales son los siguientes[9]:

- Revisar personalmente todo el trabajo propio antes de pasar a la siguiente fase de desarrollo;
- Intentar lo mejor posible corregir todos los defectos antes de dar el producto de trabajo a otra persona en el equipo de desarrollo;
- Utilizar una lista de chequeo personal y seguir un proceso estructurado de revisión;
- Seguir las buenas prácticas de la revisión: revisar en incrementos pequeños, hacer las revisiones en papel y hacerlas cuando estás descansado;
- Medir el tiempo de la revisión, el tamaño de los productos revisados y el número y tipo de defectos encontrados y perdidos;

- Usar los datos de las mediciones para mejorar el proceso personal de revisión;
- Diseñas e implementar los productos para que sean fáciles de revisar;
- Revisar los datos para identificar las formas de prevenir defectos.

2.5.11 Lista de Chequeo de Revisión de Código

Una lista de chequeo es una forma especializada que se usa en para hacer las revisiones. La lista de chequeo también te ayuda de disciplinar tu trabajo y te guía en los pasos de revisión.

Se necesita que cada persona desarrolle su propia lista de chequeo, para que cada persona se enfoque en los errores que más comete, o en los que quiere evitar según la estrategia que se siga, y que se completen todos los elementos de la lista. Para hacer un mejor uso de la lista de chequeo se tiene que dividir la lista en secciones con características similares para concentrarte en los mismos tipos de errores en cada pasada al código o al diseño. Para elaborar una lista de chequeo se puede tomar como base la tabla 2.5 [9].

Número de Tipo	**Nombre del Tipo**	**Descripción**
10	Documentación	Comentarios y mensajes.
20	Sintaxis	Ortografía, puntuación y tipos.
30	Paquete	Administración, librerías y versiones.
40	Asignación	Declaración, nombres duplicados y límites.
50	Interface	Procedimientos, referencias, I/O y formatos.
60	Chequeo	Mensajes de error y chequeos inadecuados.
70	Datos	Estructura y contenido.
80	Función	Lógica, apuntadores, ciclos, etc.
90	Sistema	Configuración, tiempo y memoria.
100	Medio Ambiente	Diseño, compilación y pruebas.

Tabla 2.5: Clasificiación de Defectos del PSP

La lista de chequeo es construida a partir de esta lista de defectos. Se deben proponer distintas actividades para enfocarse a remover los distintos tipos de defectos. La tabla 2.6 presenta un ejemplo de lista de chequeo[9]:

La lista de chequeo aparte de ser personalizada, tiene que ser actualizada constantemente. La persona al seguir los procesos de calidad tiene una mejoría notable en la forma

Lista de Chequeo para la Revisión de Código	
Nombre del Revisor	
Nombre del Programa	
Lenguaje	
Propósito	Ser una guía para la revisión de código
General	- Revisa el programa por cada categoría. - No intentar revisar dos categorías a la vez. - Cada que se complete un paso palomear el cuadro.
Documentación	- Los métodos están documentados correctamente.
Sintaxis	- Todas las líneas del programa terminan en ";". - Todas las llaves, corchetes y paréntesis tienen su pareja. - El código está correctamente indentado.
Paquete	- El programa se encuentra en la carpeta correcta. - Todas las librerías requeridas están presentes.
Asignación	- Las variables tienen el tipo de dato correcto. - Las variables siguen las convenciones de nombrado. - Todas las variables están inicializadas. - Las variables tienen el alcance adecuado.
Interface	- Los archivos leídos o escritos existen. - Las llamadas a funciones son correctas.
	- Las llamadas a funciones tienen los parámetros adecuados.
Chequeo	- Los métodos utilizados manejan excepciones. - Los mensajes de error mandados son útiles.
Datos	- La base de datos utilizada existe. - Las consultas a la base de datos son correctas.
Función	- La lógica de los métodos es correcta. - Los apuntadores están definidos correctamente. - Los ciclos respetan el tamaño de los arreglos.
Sistema	- El programa utiliza correctamente la memoria disponible.

Tabla 2.6: Lista de Chequeo para la Revisión de Código de PSP

de hacer su trabajo, por lo tanto los errores que se comenten en el tiempo van cambiando y la lista tiene que ser actualizada. También se tiene que tener en cuenta que las listas de chequeo cambian notablemente de un proyecto a otro, dependiendo del lenguaje de programación que se utilice entre otros factores.

La intención de la revisión de código es para asegurarte que todos los detalles son correctos. Una vez que hayas definido tus prácticas, debes de incorporarlas en tus estándares y checarlos en tus revisiones.

La estrategia de la revisión es:

- Para asegurarte que el código cubra todo el diseño, revisa cada método para asegurarte que todas las funciones requeridas están incluidas;
- Para checar las librerías a incluir, examina cada método para asegurarte que existan las inclusiones necesarias para cada función de la librería;
- Para checar problemas de inicialización, haz una caminata por la lógica de todos los métodos.

2.5.12 Evaluando las Revisiones Personales

Algunas medidas útiles para evaluar las revisiones son:

- El tamaño del programa revisado;
- El tiempo de revisión en minutos;
- Número de defectos encontrados;
- Número de defectos en el programa que se encontraron después, en otras palabras aquellos que no fueron encontrados en la revisión.

Las métricas formales que se proponen por PSP son: Yield de Revisión, Densidad de Defectos, Tasa de Defectos y Tasa de Revisión.

Yield de revisión es el porcentaje de defectos en el producto que fueron encontrados en la revisión. Un Yield alto es bueno, uno malo es pobre. La meta del Yield debe de ser

el 100 %. La meta de PSP es encontrar y corregir todos los errores antes de la primera compilación o las pruebas. Para hacer un estimado de Yield para cualquier fase, debes de asumir el Yield de la fase anterior.

La Tasa de Defectos es el número de defectos que se encontraron en el código por LOC.

La Tasa de Revisión es el número de líneas que se revisan por hora. El ideal es entre 200 y 400 líneas por hora.

La efectividad de los métodos de revisión es el radio de defectos removidos por hora en la revisión.

2.5.13 Efectividad de la Revisión

Las revisiones de códigos son inherentemente mejores que las pruebas[9]. Depuración es el proceso de encontrar código defectuoso que causa que el programa se comporte impropiamente. La cantidad de tiempo que se gasta realizando el debugging generalmente tiene poca relación con la complejidad del defecto[9].

La meta de la revisión es remover el máximo número de defectos, en otras palabras llegar a pruebas con cero defectos.

En una revisión, personalmente revisas tu propio programa. Una inspección es una revisión en equipo de un programa. Después de las revisiones personales, la inspección es la técnica más valiosa que un equipo de desarrollo de software puede utilizar.

2.6 Diseño de Software

La principal herramienta a disposición de los ingenieros de software son las abstracciones. Podemos crear abstracciones arbitrariamente y combinarlas en abstracciones más grandes. Si cumplimos con las capacidades de los sistemas que soportamos podemos también crear las estructuras lógicas que necesitemos.

El principal problema reside en la escala o el tamaño de los sistemas de software. La forma de subdividir los sistemas tiene que ser coherente y realmente ayudar a resolver los problemas de complejidad. Ya que si en el caso de un sistema de 1000000 LOC, creas 500 tipos de 10 LOC, entonces tendrías que diseñar un sistema de 100000 LOC, lo que parece

una ayuda, pero en realidad estás forzando a los diseñadores a relacionar coherentemente los 500 tipos distintos. Para que la escalabilidad sea útil, no solamente se debe de capturar los requerimientos físicos de la escalabilidad, si no capturar un nivel de funcionalidad significativa en las partes.

La forma más clásica es utilizar la regla de divide-y-vencerás. En otras palabras, el sistema se tiene que dividir en partes más pequeñas, las cuales deben de ser vistas como subsistemas coherentes o componentes. Si esto se hace correctamente, se reduce la complejidad en un factor de diez o más. Sin embargo hacer esto correctamente es una tarea difícil, a la que se llama diseño. El diseño de calidad tiene dos partes: la calidad del concepto de diseño y la calidad de la representación del diseño.

2.6.1 ¿Por qué Diseñar?

Producir un diseño claro, completo y libre de defectos hace más lento el proceso individual de desarrollo, sin embargo, si todo el equipo de desarrollo hace lo mismo, acelerará el trabajo en equipo ya que facilitará el trabajo de integración y pruebas de sistema.

2.6.2 El Proceso de Diseño

El diseño de software es una actividad creativa la cual no puede ser reducida a un proceso rutinario. Sin embargo, esta actividad, no tiene que carecer por completo de estructura. El diseño generalmente inicia revisando el propósito del producto, obteniendo datos relevantes, produciendo una vista general del diseño y llenando los detalles. Para diseños complejos, los buenos diseñadores siguen un proceso dinámico. Ellos trabajan en un nivel conceptual por un periodo de tiempo y después ahondan en cada parte.

El principio de incertidumbre de los requerimientos: Los requerimientos nunca van a ser completamente conocidos hasta que los usuarios utilicen el producto terminado. Tomando en cuenta este principio, el trabajo del diseñador es crear una solución funcional a un problema mal definido.

Ocasionalmente, el diseñador no va a ser capaz de especificar una función hasta que se haya diseñado, construido y probado. En estos casos se deben de construir prototipos para

estas funciones. Antes de desarrollar un prototipo, se debe de especificar su propósito y las preguntas que va a responder.

2.6.3 Niveles de Diseño

Los diferentes niveles de diseño son[9]:

1. *El diseño conceptual.* Es un concepto de planeación que se utiliza para estimar el tamaño del producto.

2. *Diseño a nivel de sistema (SLD).* Es la vista general del sistema completo de hardware y software. Si se está trabajando en frenos para autos, el nivel de sistema sería el auto, pero en cambio sí se trabaja con pistones, el nivel de sistema sería el motor.

3. *Diseño de alto nivel (HLD).* Este diseño tiene como objetivo dividir el sistema que se está desarrollando en partes más pequeñas o componentes.

4. *Diseño detallado (DLD).* Este nivel de diseño toma los componentes especificados por el HLD y define como construirlos.

5. *Diseño de implementación (ILD).* Toma el producto del DLD y lo refina de una forma en que el producto puede ser automáticamente construido.

2.6.4 Requerimientos y Diseño

Se puede argumentar que se necesita tener el conjunto completo de requerimientos antes de comenzar con el trabajo, pero esto es raramente posible debido a las siguientes razones[9]:

- La especificación de requerimientos es una habilidad especializada;
- Los requerimientos cambian conforme el proceso de desarrollo avanza;
- La solución que se desarrolla probablemente cambie el problema;
- El principio de incertidumbre de los requerimientos.

Los requerimientos deben de ser los más claro posible. Una vez que se tiene la definición completa se debe de comenzar con el diseño.

2.6.5 Estrategias de Diseño y Desarrollo

Cuando se está siguiendo una estrategia de desarrollo se deben de seguir las siguientes directrices[9]:

1. Cuando sea práctico, completar primero el HLD;
2. El diseño de un programa no se considera completo hasta que las especificaciones de todas sus abstracciones están completas;
3. Registre los supuestos para después asegurarte que son válidos;
4. Escriba notas acerca del diseño que expliquen la lógica en las decisiones de diseño;
5. Trate de resolver todas las incertidumbres antes de comenzar con el diseño;
6. Penetre en tantos niveles de diseño como sea necesario para resolver las incertidumbres.

Algunas estrategias de desarrollo son las siguientes[9]:

- *La estrategia progresiva.* Es una manera natural de desarrollar un sistema que consiste de funciones secuenciales. Si existe una estructura central, se puede dividir en pedazos verticales, implementando cada pedazo en un ciclo completo de desarrollo. Esta estrategia tiene la ventaja de ser fácil de definir e implementar. Tiene la desventaja de que no se puede probar exhaustivamente el comportamiento de cada pedazo.
- *La estrategia de mejoras funcionales.* Define una versión inicial del sistema y le agrega mejoras. Esta estrategia se sigue comúnmente en la actualización de grandes sistemas. En esta estrategia se construye un sistema funcional tan pronto como sea posible. Provee una base temprana para las pruebas de seguridad y desempeño. El principal problema de esta estrategia es que el primer paso puede ser muy grande.

- *La estrategia camino-rápido.* Es parecida a la estrategia de mejoras funcionales, con la diferencia de que la versión inicial del sistema está diseñada para demostrar el desempeño. Ya que se enfoca en el desempeño, la versión inicial del sistema solo debe de contener la lógica mínima para controlar las funciones sensitivas al desempeño. La ventaja principal es que expone problemas de tiempo en una etapa temprana. Para sistemas que tienen restricciones de tiempo o para versiones iniciales de un sistema operativo puede ser una estrategia muy atractiva. La principal desventaja es también el paso inicial.

- *La estrategia boba.* Sigue un enfoque de arriba hacia abajo. Inicia con las funciones de alto nivel del sistema y obvia el resto de las funciones con regresos predefinidos. Una vez que las funciones de alto nivel son probadas, implementa gradualmente las funciones obviadas. Esta estrategia tiene la ventaja de construir el sistema en pazos pequeños y provee considerable flexibilidad al momento de ordenar la implementación funcional. La principal desventaja de esta estrategia es que mucha de la funcionalidad útil no es vista hasta fases avanzadas del proceso de desarrollo.

2.6.6 Calidad en el Diseño

Los factores que afectan la calidad en el diseño son la precisión y la completez[9]. El diseño de software debe de contener una solución completa y precisa al problema.

Se necesita un diseño precisamente documentado, ya que si no al momento de implementación se tendrá que completar el diseño lo que es altamente propenso a errores.

Se puede desperdiciar tiempo sobre especificando el diseño, pero un diseño que no esté suficientemente especificado puede ser costoso y ser propenso a errores. Para ser más eficiente, se necesita definir específicamente el contenido que tendrá el diseño. Esto te ayudará a producir un diseño de alta calidad y te guiará al momento de hacer revisiones. Siempre y cuando cumplas con los criterios de salida, puedes utilizar los métodos de diseño que prefieras.

Las personas que utilizarán el diseño son: Programadores, diseñadores, documentadores y el equipo de pruebas. A continuación se describe:

- Administración de sistema: Registro de seguimiento de problemas, restricciones de implementación, descripción de la función del sistema, notas de aplicación, escenarios de uso y restricciones del sistema;
- Ingenieros de sistema: Una descripción de los datos y archivos relevantes, una descripción de los mensajes de sistema, condiciones especiales y razones por las que se tomaron decisiones en el diseño del sistema;
- Diseñadores de software: Una imagen de en qué parte del sistema encaja el programa, una imagen lógica del programa, una lista de las clases (clases, componentes o partes) relacionadas, una lista de las variables externas, una descripción precisa de todas las referencias y la descripción de la lógica del programa (pseudocódigo).

2.6.7 Plantillas de Diseño

La falta de un diseño preciso es la fuente de muchos errores de implementación. Para lograr una representación adecuada del sistema requieren 3 cosas: Producir un diseño coherente, registrar toda la información de diseño y registrar esa información en una forma precisa y entendible.

Una vez generado el diseño se debe analizar su corrección, completez y consistencia, para asegurarse que la notación lógica sea precisa. La notación usada para el diseño debe ser concisa y precisa, y debe de cumplir con las siguientes características[9]:

- Debe ser capaz de representar el diseño precisa y completamente;
- Debe ser entendida y usada por las personas que usan el diseño;
- Debe ayudar a verificar que el diseño hace lo que se pretende.

Un buen diseño debe tener mínima redundancia. En las plantillas de PSP, cada ítem solo se registra una vez, y se usan referencias cuando es necesario. Existen 4 categorías para agrupar los elementos de diseño en una plantilla de PSP[9]:

1. *Externo-estático*: Define las relaciones estáticas de una parte con otras partes o con elementos del sistema;

2. *Externo-dinámico*: Define interacciones de una parte con otras partes o elementos del sistema;

3. *Interno-estático*: Contiene una descripción estática de un módulo o parte, como su estructura lógica detallada;

4. *Interno-dinámico*: Define las características dinámicas de las partes.

Existen 4 plantillas de diseño diferentes en el PSP[9]:

- *Plantilla de Especificación Operacional (OST).* El OST es una forma simplificada de casos de uso que es utilizada para describir el comportamiento operacional de un programa. Ayuda a visualizar cómo el programa debe reaccionar en varios escenarios de uso. Cuando una decisión de diseño involucra a un actor, se produce un escenario para registrar cómo el actor percibe la acción. En comparación con el UML (Unified Modeling Language) es una combinación de diagramas de casos de uso, descripción de casos de uso, diagramas de actividades y diagramas de secuencia;

- *Plantilla de Especificación Funcional (FST).* El FST provee una forma simple de documentar la mayoría de los contenidos de un diagrama de clases de UML. Este tipo de plantilla describe las partes, incluyendo sus métodos, relaciones y limitantes (restricciones). Las partes pueden ser clases, módulos o incluso un sistema entero. El FST provee una descripción detallada de cada una de esas partes. En comparación con el UML es un diagrama de clases;

- *Plantilla de Especificación de Estados (SST).* Esta plantilla provee una forma simple y precisa de especificar el comportamiento de los estados de un sistema, el cual es determinado por sus entradas y el estado actual. Programas con múltiples estados son denominados máquinas de estados. En comparación con el UML es un diagrama de estados;

- *Plantilla de Especificación Lógica (LST).* Es un tipo de plantilla que usa un lenguaje simple de diseño de programación (PDL) para describir la lógica del programa.

Los PDLs también son conocidos como pseudocódigo. La plantilla contiene una descripción en pseudocódigo de un programa. El objetivo es describir de manera clara y concisa qué hará el programa y cómo. Se debe usar lenguaje humano ordinario, evitando usar expresiones de programación. De otra manera se estaría programando el código del programa, y no el diseño.

Las plantillas también son útiles al mejorar programas ya existentes. Ya que para producir mejoras sin defectos es necesario conocer el diseño, o es posible que haya errores de diseño, las plantillas ayudan a documentar el diseño de estos programas.

Estas plantillas se pueden usar de manera escalada. Es decir, en un sistema muy grande, se pueden usar plantillas de FST y OST para definir la funcionalidad del sistema en general, y usar plantillas separadas para definir la funcionalidad de los sub-sistemas o componentes.

2.6.8 Verificación de Diseño

Para programas largos y complejos, hasta los métodos de verificación de diseño que más consumen tiempo y más trabajo requieren son más efectivos y rápidos que las pruebas. Los siguientes métodos de verificación son relativamente rápidos de aprender:

- Estándares de diseño;
- Verificación por tabla de ejecución;
- Verificación por tabla de seguimiento;
- Verificación de máquinas de estado;
- Verificación analítica.

Los programas tienen que ser verificados porque los desarrolladores tienen la creencia que pocos defectos en un programa son aceptables mientras el programa se ejecute normalmente bien[9]. Pero qué pasa cuando 1 o 2 defectos por cada 100 líneas de código se convierten en 10,000 a 20,000 en 1,000,000 de líneas de código.

Es recomendable utilizar técnicas de verificación de diseño para reducir el número de defectos y aumentar la calidad del programa. El problema no es que las pruebas sean

malas, sino que no son suficientes. Imagina analizar todas las posibilidades de ejecución de cierto programa, eso nos llevaría demasiados años. Por eso es importante utilizar técnicas de verificación para reducir el número de defectos, que además se traduce en la reducción del costo para encontrar esos defectos, ya que se encuentran en una etapa temprana de desarrollo.

Aunque no es una técnica de verificación en sí, es importante para comparar el diseño de un programa contra las bases de los estándares. Todos los estándares definen convenciones para el producto, estándares de diseño del producto y estándares para reutilización de código.

Las convenciones para el producto final incluyen las interfaces de usuario, la manera de atacar errores, convenciones de nombres, procedimientos de instalación y la ayuda. El estándar de diseño del producto abarca desde las convenciones hasta la arquitectura del sistema. Los estándares para reutilización de código precisan que las partes a reutilizar estén completamente definidas, sean de la mayor calidad y estén soportadas adecuadamente.

Las distintas técnicas de verificación son[9]:

- *Verificación por Tabla de Ejecución.* El propósito de esta técnica es verificar el flujo lógico de ejecución del programa. Esta técnica puede consumir bastante tiempo, sin embargo, es bastante confiable, sencilla y es mucho más rápida para identificar problemas complejos de diseño que las pruebas. Es importante seguir un proceso para realizar esta verificación, para de esta manera maximizar su efectividad sin tener que desarrollar demasiados casos o escenarios. Se recomienda comenzar identificando los ciclos y rutinas complejas para su revisión. Posteriormente definir cómo se hará el análisis. Si de abajo hacia arriba o de arriba hacia abajo, esto depende de cuál de los niveles (alto o bajo) sea el más sólido en el diseño. Si el nivel alto se ve sólido, se recomienda comenzar con el nivel bajo y viceversa.

- *Verificación por Tabla de Seguimiento.* Es muy similar a la tabla de ejecución, solamente que aquí se realiza la verificación de una manera más general. Con la tabla de ejecución, básicamente se emula el funcionamiento de una computadora. Con la tabla de seguimiento, primero se consideran todos los posibles casos lógicos y se selecciona

un escenario para cada caso. Aunque son muy similares, en general las tablas de seguimiento son mucho más eficientes que las tablas de ejecución. El método para ejecutar esta técnica de verificación es el mismo que para las tablas de ejecución. Durante el análisis mediante esta técnica, es bueno considerar la robustez del programa. Como por ejemplo, cuáles serían los límites de ejecución y si el diseño los soportará, cuáles son las excepciones que se pueden presentar y si el diseño las manejará correctamente. Es importante considerar que si la lógica del diseño es demasiado compleja para ser analizada por una tabla de seguimiento, lo mejor sería rediseñar esta parte.

- *Verificación de Máquinas de Estado.* Si el programa incluye máquinas de estado, es importante verificar que estas máquinas hayan sido diseñadas correctamente. De manera general, se considera que una máquina de estado es correcta si se puede alcanzar el estado de retorno (si es que lo hay) desde cualquier otro estado y si las transiciones entre los estados son completas y ortogonales.

- *Una función completa es aquella que incluye todas las condiciones lógicas posibles.* Existen técnicas, como los mapas de Karnaugh, para determinar si una función es completa o no. Además, un conjunto de funciones es ortogonal si ninguno de sus miembros tiene ninguna condición en común. Si una máquina de estado cumple con estas dos condiciones, se puede decir que es una máquina de estado correctamente diseñada.

- *Verificación Analítica.* Dentro de las verificaciones analíticas existe un caso especial que es la verificación de ciclos. La verificación de ciclos nos ayuda a asegurarnos que el programa no caerá en ningún ciclo infinito, no se quedará atorado y no presentará ningún comportamiento extraño. En general, la verificación de ciclos consiste en verificar que las precondiciones del ciclo se cumplan siempre, que se garantice la terminación del ciclo para cualquier argumento que afecte al ciclo y que se mantenga la identidad cuando se cumple la condición de terminación, es decir, que no se ejecute una o más veces de más.

Así mismo, existen otras técnicas de verificación analíticas que ayudan a verificar la lógica del programa. Para seleccionar alguna técnica en particular o alguna herramienta

que realice esta técnica, es importante considerar lo siguiente:

- Si la técnica puede conducir a resultados erróneos;
- Si la técnica podrá encontrar todos los defectos de una categoría en especial o solamente algunos;
- La profundidad con la que podrá realizar el análisis;
- La eficiencia respecto al tiempo que tomará el análisis;
- Si se necesita que el producto esté finalizado completamente o no.

El tipo de productos que puede verificar. Si es de propósito general o de algún dominio en específico.

Las herramientas que proveen una verificación automática puede ser muy útiles, pero también peligrosas si no se usan adecuadamente. Siempre es necesario tener en cuenta que es muy poco probable que estas herramientas muestren el 100 % de errores cometidos en el diseño. Por lo que se recomienda realizar siempre revisiones personales al diseño y otras técnicas de verificación antes de utilizar herramientas para realizar una verificación automatizada.

Cuando se diseñan programas complejos, siempre se cometen errores. Por lo tanto, siempre será importante realizar revisiones completas a todos los diseños que se hagan. Además, es importante contar con una estrategia de verificación que permita maximizar los resultados de la verificación. Esta estrategia puede no ser la misma para diferentes tipos de programas.

Sin embargo, el mejor momento para realizar la verificación y los análisis siempre será mientras se produce el diseño. Mantener el registro de resultados actualizado y seguir procedimientos para la revisión del diseño serán dos tareas que maximizarán los resultados de la verificación.

2.7 Seguimiento de Defectos

2.8 Costo de la Calidad en el Desarrollo de Software

Bibliografía

[1] P. B. Crosby, *Quality Is Free: The Art of Making Quality Certain: How to Manage Quality - So That It Becomes A Source of Profit for Your Business.* McGraw-Hill Companies, 1979.

[2] J. M. Juran and F. M. Gryna, *Juran's quality control handbook.* McGraw-Hill, 1988.

[3] E. Davies and M. Whyman, "Iso 9000:2000-new iso, new responsibilities for top management," *Engineering Management Journal*, pp. 244 – 248, 2000.

[4] R. Dhiman, C. Sigel, and J. Dörr, *ISO/IEC 9126 Standard*, ISO/IEC Std., 2005.

[5] D. Stelzer, W. Mellis, and G. Herzwurm, "Software process improvement via iso 9000? results of two surveys among european software houses," in *Proceedings of the Twenty-Ninth Hawaii International Conference on System Sciences*, 1996, pp. 703 – 712.

[6] C. Yoo, J. Yoon, B. Lee, C. Lee, J. Lee, S. Hyun, and C. Wu, "An integrated model of iso 9001:2000 and cmmi for iso registered organizations," in *Software Engineering Conference, 2004. 11th Asia-Pacific*, 2005, pp. 150–157.

[7] J. E. Bentley and W. Bank, "Software testing fundamentalsŮconcepts, roles, and terminology."

[8] W. S. Humphrey, *Winning with Software, An Executive Strategy, How to Transform Your Software Group into a Competitive Asset.* Addison - Wesley, 2002.

[9] ——, *PSP A Self-Improvement Process form Software Engineers*, P. Education, Ed. Addison - Wesley, 2005.

[10] K. Owens, "Software detailed technical reviews: Finding and using defects," in *Proceedings Wescon*, 1997.

[11] L. Harjumma, "Peer reviews in real life - motivators and demotivators." *International Conference on Quality Software*, 2005.

[12] G. M. Freedman and D. P. Weinberg, "Reviews, walkthroughs and inspections," *IEEE Transactions on Software Engineering*, 1984.