**Bug Manager**

Sistema de Administración de Calidad de Proyectos de Software

Anteproyecto de Tesis

Rev 0.1

Fecha: 21 de marzo de 2011

Autor: MR, HG, EC

|  |  |  |  |
| --- | --- | --- | --- |
| **Autoridad** | **Nombre** | **Firma** | **Fecha** |
| Director de Tesis | Óscar Mondragón (OM) |  |  |
| Tesista | Marco Rangel (MR) |  |  |
| Tesista | Eduardo Campos (EC) |  |  |
| Tesista | Humberto García (HG) |  |  |

# Contenidos

|  |  |
| --- | --- |
|  | [Contenidos 2](#_Toc292379069)  [1. Control del Documento 3](#_Toc292379070)  [1.1. Propósito y alcance 3](#_Toc292379071)  [1.2. Audiencia 3](#_Toc292379072)  [1.3. Organización del Documento 3](#_Toc292379073)  [1.4. Definiciones 3](#_Toc292379074)  [1.5. Control de Versiones 4](#_Toc292379075)  [2. Resumen 4](#_Toc292379076)  [3. Problema 4](#_Toc292379077)  [3.1. Planteamiento del Problema 4](#_Toc292379078)  [3.2. Justificación y Áreas de Oportunidad 5](#_Toc292379079)  [4. Objetivos 6](#_Toc292379080)  [4.1. Generales 6](#_Toc292379081)  [4.2. Específicos 7](#_Toc292379082)  [5. Alcances y Contribuciones 7](#_Toc292379083)  [6. Investigación Preliminar 8](#_Toc292379084)  [6.1. Antecedentes 8](#_Toc292379085)  [6.2. Marco Teórico 10](#_Toc292379086)  [6.2.1. Winning With Software 10](#_Toc292379087)  [6.2.2. Ténicas de Detección de Defectos 15](#_Toc292379088)  [6.2.3. Personal Software Process 20](#_Toc292379089)  [6.2.4. Clasificación Ortogonal de Defectos 36](#_Toc292379090)  [7. Metodología 38](#_Toc292379091)  [7.1. Investigación 38](#_Toc292379092)  [7.2. Desarrollo de Software 39](#_Toc292379093)  [8. Resultados Esperados 41](#_Toc292379094)  [9. Bibliografía 41](#_Toc292379095)  [10. Anexos 42](#_Toc292379096)  [10.1. Concepto de Operaciones 42](#_Toc292379097) |

# Control del Documento

## Propósito y alcance

El presente documento refiere al anteproyecto de tesis para el sistema de administración de calidad de proyectos de software, cuyo nombre clave es “Bug Manager” (BM).

## Audiencia

Este documento está dirigido a:

* Oscar Mondragón (Director de tesis).
* Luis Eduardo Falcón (Director de la maestría).
* Tesistas: Eduardo Campos, Humberto García y Marco Rangel.

## Organización del Documento

|  |  |
| --- | --- |
| Sección 1 |  |
| Sección 2 |  |
| Sección 3 |  |
| Sección 4 |  |
| Sección 5 |  |
| Sección 6 |  |
| Sección 7 |  |

## Definiciones

|  |  |
| --- | --- |
| **Término** | **Significado** |
| BM | Bug Manager |

Tabla1: Definiciones

## Control de Versiones

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Versión** | **Fecha** | **Editor** | **Comentarios** | **Aprobación** |
| 0.1 | Abril 21, 2011 | MR | Versión inicial del Anteproyecto. |  |
| 0.2 | Mayo 5, 2011 | MR, HG, EC | Versión para revisión. |  |

Tabla2: Control de Versiones

# Resumen

El BM será utilizado por cualquier empresa de desarrollo de software, 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.

El BM permitirá que las empresas que lo utilicen puedan tener control sobre el avance y la calidad de sus proyectos por medio de lo siguiente:

* 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.

# Problema

## Planteamiento del Problema

Actualmente gran parte de las pequeñas y medianas empresas de desarrollo de software en México no dan un apropiado seguimiento a la administración de la calidad durante su proceso de desarrollo de software.

Muchas de estas empresas simplemente piensan que las prácticas de calidad agregan trabajo extra, haciendo más lento y complicado el proceso de desarrollo, lo que provoca atrasarse en los calendarios y no entregar los productos a tiempo.

Estas decisiones provocan exactamente el efecto contrario. Las empresas al no tener un apropiado control de la calidad de sus productos, ocasionan las siguientes situaciones:

* Una vez que se llega a la etapa de pruebas del producto de software, este está tan plagado de defectos que esta etapa se vuelve demasiado larga, muchas veces toma hasta la mitad del tiempo total de desarrollo.
* Una vez que las empresas terminan la tortuosa etapa de pruebas y el sistema sale a producción, los clientes se encuentran con una gran cantidad de defectos los cuales no se encontraron en la etapa de pruebas, o simplemente se inyectaron al momento de hacer las correcciones de otros defectos.
* Cuando los usuarios finales encuentran defectos en el producto final, la empresa tiene que hacer las correcciones de estos defectos. Corregir un evento en etapa de pruebas cuesta normalmente diez veces más de lo que costaría en codificación, y corregir un defecto en mantenimiento cuesta cien veces más que hacerlo en codificación. Lo que provoca altos costos de mantenimiento los cuales deben de ser absorbidos por la empresa.
* Peor aún que los costos de mantenimiento, el cliente es incapaz de realizar las tareas que requería con el sistema propuesto, lo que genera desconfianza en la empresa de desarrollo, una mala publicidad y pérdidas de clientes en el futuro.

Todas estas situaciones pudieron 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 toda empresa que se dedique al desarrollo de software. Esto es debido a que la calidad ayuda a que los productos sean predecibles en tamaño y calendario, que sean fáciles de dar seguimiento y que tengan un bajo costo de mantenimiento. Al contrario de hacer más largo y complicado el proceso de desarrollo, las actividades y administración de la calidad recortan los tiempos de desarrollo, debido a que el tiempo de la etapa de pruebas se reduce considerablemente. También el costo del proyecto se reduce automáticamente al momento de reducir el tiempo de desarrollo, agregando lo que se ahorra debido a la poca cantidad de defectos que llegan a la etapa de pruebas y a producción. Finalmente y lo más importante, el cliente tendrá un producto de calidad, la empresa de desarrollo tendrá una buena imagen y más y mejores proyectos en el futuro.

## Justificación y Áreas de Oportunidad

Normalmente las empresas medianas y pequeñas piensan que la administración de la calidad 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.

Viendo estas razones y la problemática descrita en el punto anterior, se pretende crear una herramienta 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.

La herramienta 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 sistema 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 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.

# Objetivos

## Generales

Podemos definir como objetivos generales de la realización de la tesis los siguientes:

1. Exposición del estado del arte en referencia a las herramientas de apoyo del proceso de calidad: Actualmente existen varias decenas de herramientas que ayudan en las actividades de calidad para el desarrollo de productos de software. Algunas se enfocan en actividades de seguimiento, otras simplemente como bitácora y así sucesivamente. El objetivo es exponer las más importantes y sus funcionalidades actuales.
2. Desarrollo del documento de tesis: Obviamente es un objetivo primordial el desarrollar un documento acorde al tema de investigación, que contenga la información suficiente y adecuada para exponer el tema y explicar correctamente el motivo de la realización de la misma, que contenga el marco teórico adecuado, la metodología de elaboración y los resultados obtenidos en la realización de la tesis.
3. Desarrollo de la herramienta como tal: Construcción de un producto de software conforme a lo establecido en el documento de tesis y en la documentación de diseño del mismo, que satisfaga las necesidades presentadas, así como las funcionalidades requeridas. El desarrollo de este producto es tan importante como el documento de tesis, por lo que es vital que en el producto final se encuentren todas las funcionalidades especificadas para alcanzar el éxito en el proyecto.

## Específicos

Debido al tamaño de la herramienta que se desarrollará, se presentará un panorama general como base teórica y se desarrollarán tres temas de investigación a profundidad. Estos temas de investigación se reflejarán directamente en la funcionalidad de la herramienta. Por lo tanto podemos definir como objetivos específicos los temas de investigación a desarrollar:

1. El costo de la calidad:
2. Comparación y efectividad de técnicas de detección de defectos: El objetivo es la correcta implementación de técnicas de detección de defectos, que sean lo suficientemente efectivas en las diferentes fases de desarrollo de productos de software, y al mismo tiempo que cuenten con la suficiente flexibilidad para adaptarse a proyectos diversos. También es importante que la información requerida por el sistema y que introducirán los usuarios de la herramienta sea fácil de cuantificar para evitar el cálculo erróneo de estadísticas. Por último, los reportes y estadísticas que se presentarán referentes a las técnicas de detección permitirán visualizar un panorama completo que permite tomar las decisiones adecuadas para proyectos presentes y futuros.
3. Caracterización, clasificación y seguimiento de los defectos:

# Alcances y Contribuciones

Los alcances de la herramienta están descritos completamente en el Anexo 1 del presente documento. Como contribuciones se pretende lo siguiente:

1. 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.
2. 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.
3. 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.

# Investigación Preliminar

## Antecedentes

Actualmente gran parte de las pequeñas y medianas empresas de desarrollo de software en México no suelen utilizar sistemas de administración de calidad en su proceso de desarrollo. El sistema BM pretende cubrir esta necesidad, mediante la instalación del producto y la adopción del mismo, así como del nuevo proceso implícito en la adopción del sistema.

En la actualidad, la mayoría de las empresas cuentan con sistemas de administración de proyectos o tareas donde se controla principalmente el avance del calendario y en algunos casos el esfuerzo real de las tareas. La mayoría de estos sistemas de administración carece del registro, administración y/o planeación de las actividades de calidad asociadas al desarrollo de software.

Algunas empresas cuentan con herramientas que les ayudan a dar seguimiento a los defectos encontrados. Estos sistemas se usan principalmente en las fases de pruebas o durante el mantenimiento de los sistemas. Los defectos no son analizados por las personas que los cometen y esto dificulta que los programadores de manera consciente eviten los errores que cometen de manera frecuente.

Algunas otras empresas que usan listas de chequeos para revisiones entre colegas, usan sistemas de seguimiento de corrección de defectos. No es frecuente que estás empresas realicen análisis de los errores cometidos, tampoco son asociados los errores a los programadores que los cometen.

Los sistemas arriba mencionados están enfocados al registro y seguimiento de defectos careciendo de un enfoque de prevención y análisis de datos.

El BM no sustituirá a ninguna aplicación o sistema en específico, siendo su principal objetivo el de optimizar y hacer más eficiente el proceso de desarrollo de software en pequeñas y medianas empresas, principalmente mexicanas, que deseen incorporar actividades de mejora continua en el ciclo de vida de desarrollo.

Sin embargo, 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.

Hay más de 50 sistemas de registro y seguimiento de defectos, con diferentes tipos de licenciamiento y diferentes funcionalidades. A continuación presentamos una descripción de los sistemas que consideramos más significativos y similares en cuanto a funcionalidad respecto al sistema propuesto.

* *Bugzilla*: Es una herramienta de propósito general basada en tecnología web, pensada como un sistema de rastreo de defectos. Fue desarrollada para el proyecto Mozilla. Aunque su diseño soportaría ser una herramienta de administración de proyectos y actividades, el equipo de desarrollo ha decidido mantenerlo como un sistema de rastreo de defectos. Es prácticamente un sistema de tickets, en el que cualquiera puede levantar un nuevo ticket, que corresponde a un defecto, y este ticket se asigna a un desarrollador. Este sistema se utiliza tanto para proyectos propietarios como de open source.
* *PSP*: La herramienta de PSP permite el registro de las actividades realizadas para el desarrollo del producto, así como el registro de defectos (por tipo) inyectados y removidos en cada actividad. Además permite la estimación del proyecto de desarrollo. En base a estos registros, calcula una serie de métricas referentes a los defectos, al plan de estimación, al proceso y a la calidad. Todas estas métricas proporcionan información valiosa que posteriormente permite tomar decisiones para mejorar continuamente el proceso de desarrollo.
* *BugTracker .NET*: Esta herramienta cuenta con funcionalidades interesantes, como la de capturar una pantalla y registrarla como un defecto, poniendo las características del mismo como comentarios. También tiene la capacidad de registrar los correos electrónicos recibidos como defectos. Una parte importante es la facilidad para filtrar y ordenar la lista de defectos y sus columnas conforme a las necesidades del usuario. Además se pueden definir los estatus de defectos y el proceso a seguir para el seguimiento de los mismos.
* *BugNET*: Esta herramienta contiene un módulo para administración del proyecto, así como un módulo de administración de defectos. Dentro de las funcionalidades interesantes podemos mencionar el soporte para múltiples proyectos, el cálculo y reporte de métricas, la capacidad de crear un resumen con los defectos por proyecto, así como la creación de una bitácora con el histórico de cada defecto. Además de la posibilidad de agregar archivos a cada defecto, realizar comentarios por defecto, notificación de cambios en los estatus y la capacidad de configurar roles y permisos para cada usuario.
* *Bontq*: Es una de las herramientas más recientes para la administración de proyectos y seguimiento de defectos. Dentro de sus funcionalidades más destacables se encuentran la integración con Google Docs, la posibilidad de grabar videos (para mejor demostración de un defecto), capturar pantallas y agregar archivos a cada defecto. Además provee la capacidad de manejar los proyectos por compañías, es decir, el nivel más alto de administración es la empresa y no el proyecto, por lo que se puede contar con la administración de varias empresas con múltiples proyectos cada una.

También permite la posibilidad de registrar defectos, solicitudes de nuevas funcionalidades, avisos, actividades y casos de prueba. Además permite la configuración de la clasificación de defectos por su impacto, así como el envío de notificaciones vía correo electrónico.

Con esto nos podemos dar una clara idea del estado en el que se encuentran las herramientas para administración y seguimiento tanto de proyectos como defectos. Algunas herramientas cuentan con ciertas funcionalidades y otras se concentran en otro tipo de funcionalidades. Todo esto nos da un panorama de lo que ya está hecho y de cómo lo podemos combinar para lograr un sistema completo y eficiente.

## Marco Teórico

El material revisado se encuentra resumido en los puntos:

* 6.2.1 comprende la lectura y resumen del libro “Winning With Software” [13].
* 6.2.2 comprende una investigación acerca de las “Técnicas de Detección de Defectos” [1-11].
* 6.2.3 comprende la lectura y resumen del libro “Personal Software Process” [11].
* 6.2.4 comprende la investigación acerca de la “Clasificación Ortogonal de Defectos” [12].

### Winning With Software

1. **EVERY BUSINESS IS A SOFTWARE BUSINESS**

Regardless of the industry you are in, you almost certainly use software in just about every part of the business.

To manage a software intensive business, you must observe three management principles:

1. Recognize that you are in the software business.
2. Quality must be the top priority
3. Quality software is developed by disciplined and motivated people.

There are several principles involving software management that every business must be aware of:

1. Maintain control of product uniqueness. If you do not recognize that you are in the software business and do not own the software in your products and services, you will lose control of your product’s uniqueness. Then you will likely lose control of business revenue and profit.
2. Quality is more important than schedule. Quality is an economic choice; pay a little now or a fortune later. To truly accelerate development work and optimize time to market, your people must do their jobs the right way the very first time.
3. In software, what must happen often does not. Software managers and professionals who are not trained in quality methods will not believe that quality is important and will not follow the disciplined practices required to build quality products.
4. **WHY PROJECTS FAIL**

The main reason why the projects fail is poor management. Proper management requires only two things: a dedication to quality and motivated engineers who do disciplined work.

The five most common causes of project failure are:

1. Unrealistic schedules. When software projects start with unrealistic schedules, they are invariably delivered much later than they would have been with a rational plan, because the engineers start doing a fast low quality work who translates in testing time.
2. Inappropriate staffing. The only way to complete an engineering project rapidly and efficiently is to assign an adequate number of people and then protect them from interruptions and distractions.
3. Changing requirements. While the requirements normally change during the early phases of the job, there is a point beyond where changes are damaging. After this point, changes waste time and money and disrupt the work.
4. Poor quality work. If the work done has low quality, it will translate into a lot of time in testing to find and fix the defects in the project.
5. Believing in magic. There is not silver bullet. An unquestioning belief in a miraculous answer can lead to serious problem. It generally costs as much to test and fix a defective product as it took to develop it in the first place.

The cost of poor quality is rarely visible until the end of the project. Often, in fact, the poor quality will continue causing customer problems long after the project is over. The costs of poor quality continue for a long time. Common errors are: managers made undisciplined commitments, and they did not insist that the job be done properly. All the problems mentioned above could have been avoided if management insisted on planned and disciplined work.

1. **RATIONAL MANAGEMENT**

The principle of rational management is that your people are loyal and thinking professionals who would like to support you in addressing organization’s problems.

Rational management means making plans, following these plans, and fixing problems before they get out of control.

There are four elements of rational management:

1. Set aggressive long term goals, but break these goals into realistic and measurable short term goals.
2. Insist on plans and insist that these plans be made by the people who will do the work. The plans have to be detailed and complete, and you must review them for fats and omissions.
3. Use facts and data. By objectively using the team’s facts and data, you demonstrate trust in your people and a willingness to listen to their problems, plans, and ideas.
4. Monitor the work, and use current data to anticipate and resolve future problems. Schedules slip a day at a time.

When the work is planned, measured, and tracked, you can regularly monitor business performance.

1. **WHY QUALITY PAYS**

There are three basic reasons to insist that software quality has to be measured and managed:

1. Poor quality can cause major property damage and even kill people.
2. Quality work saves time and money.
3. If you don’t manage software quality, nobody else will.

Even experienced engineers inject a defect about every ten lines of code. The programmers are not incompetent or lazy, they are just human.

Cost of finding and fixing defects increases at every step in the development process. The longer a defect stays in the product is more expensive to remove it.

There are several techniques that help engineers to remove defects in the development process. Code reviews are done by reading and analyzing their programs. In inspections, several engineers review a program to find its problems. In unit testing, each programmer test the small program modules that he or she developed and fixes all of the defects found. In system testing, all of these small program modules are combined and tested as a complete system.

The system testing is the quality activity that takes longer. The reason is that a test reveals only the symptoms of the problem.

The cheapest and most effective way of removing software defects is by doing code reviews. The engineer who developed the program is best able to find and fix its defects. But even knowing that code reviews are so effective, a lot of people do not use them. The reason is that good code reviews require disciplined methods and special training.

Any project with a large amount of testing will be nearly impossible to plan and track. Therefore, if you want accurate plans and reliable commitments you must insist on quality work.

1. **LEADERSHIP GOALS**

All the organizations who want to succeed need to set improvement goals. Many companies aim to make things faster, better and cheaper. A principle in management is that what gets measured gets managed and what gets managed gets done. On the other hand, what does not get measured and managed often gets ignored.

There are three steps to fixing the software work in an organization:

1. Decide that you need to make changed.
2. Decide what kind of software business you want.
3. With a defined goal, you can start making changes.

In order to establish a goal, you basic objectives concern three principal topics: schedule, quality and cost. The real goal must be to change the way that engineers work. To actually get faster, better and cheaper you need specific goals. Sign these goals and track them to ensure that they are met.

To actually accelerate the work, you must focus on the tasks, how they are done, and what it takes to accelerate an engineering project.

The first requirement for being faster, better and cheaper is to make detailed and comprehensive plans. Second the engineers must make the plans. Third, the managers must ensure that the engineers are all trained to make detailed and accurate plans. Fourth, the immediate managers must participate with their teams in making the plans, and finally, more senior managers must review these plans for detail and completeness.

The next requirement is to have the teams use their plans to do the work. One of the major benefits of having detailed plans is that teams can do workload balancing. Quality is a key point, because if you have parts or pieces with low quality, you will spend a lot of time fixing them. After quality the next key is motivation.

A quality goal in an organization should be: that the engineers be trained in quality methods and that they use these methods on the job.

The principal thing you can do to reduce the engineering costs is to help the engineers maximize the time spent actually working on the project. Task time is the time that engineers spend working on the scheduled tasks. Task time does not include time spent in meetings, on breaks, asking management guidance, or any of other normal and necessary activities involved in engineering projects. Engineers work between 10 to 15 task hours a week.

There are four key elements for improving task time: Task time measures, engineer motivation, task time plan, and management review and support.

In order to be faster, better and cheaper; the engineers must use disciplined methods, plan and track their work, and measure and manage product quality; the managers must initiate and maintain this new behavior; and company leaders must establish goals and track the work.

The managers must manage the work, develop the workforce, and maintain an effective and attractive workplace. The managers must also guide the teams’ planning, monitor their plans, track task time performance and oversee product quality.

The team must provide a disciplined environment, ensure that everyone does quality work, dynamically balance workload, monitor weekly task time, and track and report on project status.

The engineers must make detailed and accurate plans; record their time, size and defect data; measure, manage, and report their task time; and be disciplined in using quality methods in all of their work.

1. **CHANGING ENGINEERING BEHAVIOR**

If your objective is “faster, better, cheaper” that requires changing engineering behavior. If you want your engineers to make detailed plans to base these plans on historical performance data, and to adjust their plans as they do the work, you must provide them with the proper training. You must also convince them that planning, data gathering, and tracking are important and personally helpful.

You can only get quality in software by implementing disciplined software practices. The quality of the product is determined by the quality of the process that produced the product.

The rationale for focusing on the quality of each engineer’s personal work was that if every small program in a big system is not of high quality, the system cannot be of high quality either.

With typical practices, engineers spend about half their development time compiling and testing. The resulting programs generally are defective. If engineers concentrate on quality from the beginning they reduce compile and test defects by five or more times and cut the time by three or more times.

The managers and engineers must agree with disciplined methods and support their use. The managers must be responsible for implementing the changes in the methods. The engineers must believe that the new methods will work for them. It takes no longer to predictably write high quality programs that it took them to write defective programs.

1. **BUILDING MOTIVATED TEAMS**

The best results are got when the management trusts the engineers to do their best. In a business context there are only three ways to motivate people: fear, greed, and commitment. For any but the simplest work, fear and greed are not effective motivators. That leaves you with commitment.

To build motivated teams, first train all the engineers. Next, convince the engineers that the job they are doing is exciting and important that need their personal commitment. Give them aggressive goals, but also have them make the plans for meeting the goals. Whether or not the plan meets the management goals, have the engineers describe it and why they believe is right. Show that you need this product and that the date is important, but also demonstrate your need for a meaningful commitment. Look for any fat in the plan, but also look for holes and omissions. You don’t need a promise; you need a realistic commitment that the team will meet.

In order to motivate engineers, you need to build a trusting environment. If you do trust the engineers, they will keep thinking and, once they better understand the job, they are likely to devise creative ways to accelerate their work.

If you push a team to meet an unrealistic schedule, you will likely get a disaster, regardless of the process you teams follow. With an unrealistic date, teams take longer to produce poor quality products that do not meet user’s needs, because they start cutting the quality practices thinking they will save time. Then they spend more time fixing the product that it would have taken to do the job properly in the first place.

Review team performance monthly or at least quarterly and have management follow project progress every week. Show continued interest in the work and regularly ask about quality.

1. **THE BENEFITS OF TEAMWORK**

TSP introduction program shows that you will fully recover your costs within the first two years with an average five year return of over 300%. PSP and TSP, organizations have significantly improved the predictability and cycle time performance of their software groups. They have also reduced development costs, improved product quality, and reduced employee turnover.

When engineers are properly trained and use disciplined methods to plan, track, and manage their work, they consistently deliver quality products on schedule. What is more, they know precisely where they stand and can predict when they will finish.

The time required to write high quality programs is no more than the time to produce defective ones, quality work sharply cuts development time. Fewer defects mean lower costs and more accurate cost estimates. Defects in completed TSP systems are at or very near zero.

Engineers like the personal control that the TSP provides them. They also feel proud when they deliver quality products on time. They consider that developing software is fun. A reputation as a good place to work is money in the bank.

1. **NEXT STEPS**

There are seven steps you need to follow to transform an organization culture.

1. Establish a quality policy. Tell you people that quality work is you first priority. Write and issue a policy saying that the right way to do a job is always the best way.
2. Identify an improvement champion. Process improvement cannot be the engineers or managers top concern. You need a champion keep track of the improvement processes and the organization transformations.
3. Set precise and measurable goals. To achieve difficult objectives, people must have clear goals and measurable benchmarks.
4. Hold line managers responsible. To improve the performance of you organization, you must improve the way that the people work. To make the transformation happen, the line managers must be responsible for the introduction of the changes and improvements in their own departments.
5. Provide improvement resources.
6. Establish priorities. The key to process improvement is to identify a few areas for improvement and concentrate on those. Once these areas are under control, move on to the next.
7. Provide continuing oversight. Hold your top managers personally responsible, regularly review status, and do not accept excuses, and you will get rapid results.

Software is an important part of your business, and it will be more important in the future. If your software people do not consistently meet their commitments with quality products, your business will suffer, and it could even fail. To run your business and to maintain a competitive position in your market place, you must have a superior software capability.

### Ténicas de Detección de Defectos

La gran mayoría de las personas que alguna vez han estado involucradas en el proceso de desarrollo de software, se han visto en la situación de estar bajo presión por el calendario de entrega del proyecto. El primer pensamiento y la primera reacción es acelerar el proceso de codificación para terminar más rápido y dejar más tiempo para la fase de pruebas. Sin embargo, esta decisión (también la gran mayoría de las veces) tiene como consecuencia que el producto final no sea de la calidad necesaria y en última instancia provoque la desconfianza del cliente en el sistema.

Evidentemente, este no es un escenario deseado para cualquier compañía dedicada al desarrollo de software, ya que puede traer repercusiones para futuros proyectos (como su cancelación). Es principalmente por este escenario el que la mayoría de la gente involucrada en desarrollo de software, tanto a nivel operativo como gerencial, se pregunta si es posible realizar el desarrollo igual o más rápido y con una mayor calidad. La respuesta es contundente: sí se puede.

Las técnicas, formales y no formales, de detección de defectos que se encuentran en la literatura son herramientas valiosas que lograrán aumentar de manera significativa la calidad total del producto final. La primera técnica de detección concebida como tal de la que se tenga conocimiento surgió en 1976 gracias a Michael Fagan, por lo que llevan prácticamente 35 años en el mundo del software. Ahora bien, lo importante es saber qué son, cómo se implementan, quién las implementa, cuándo se implementan y lo más importante, porqué es bueno implementarlas.

Las técnicas de detección de defectos se pueden definir como la evaluación y revisión de un producto de trabajo por parte de uno o más compañeros calificados para tal actividad [1]. Esta definición nos da una idea de quiénes realizan las revisiones. Es importante escoger adecuadamente el número de personas involucradas para tal actividad y el perfil de las mismas. Pueden ser desde una hasta la cantidad que los recursos y el tiempo permitan, aunque más adelante se verá que solamente el agregar más y más personas tampoco es lo mejor necesariamente. También es importante señalar que aunque la gerencia no participa directamente en las revisiones, es muy importante su apoyo activo para la realización de estas actividades.

Una de las grandes ventajas de implementar técnicas de detección de defectos es que no solamente se limitan a la revisión de código, sino que se pueden aplicar para revisiones de documentos de especificación, de diseño, etc. Por lo que se pueden aplicar prácticamente en todo el ciclo de vida de desarrollo. Sabemos que cada compañía de desarrollo (de cualquier tamaño) tiene sus particularidades en cuanto a la adopción de diferentes procesos, y las técnicas de detección de defectos no son la excepción. Aunque estas técnicas están bien definidas en la literatura, siempre es importante preguntarse si la implementación al pie de la letra es lo más adecuado o si es mejor adaptarlas a las características de la empresa. La mayoría de las veces la respuesta más acertada es lo segundo.

En este punto nos podemos preguntar porque implementar este tipo de técnicas o procesos traerá ventajas al proceso mismo de desarrollo y aumentará la calidad de los productos. Podemos enumerar como ventajas importantes las siguientes:

1. Son efectivas en cuanto a costos se refiere. Es por todos conocido que entre más tarde en el ciclo se encuentre un defecto, más costará su corrección. Por lo que al aplicar estas técnicas en etapas tempranas (requerimientos, diseño, codificación) el costo disminuye considerablemente, inclusive hasta 100 veces menos [2]. Además, el no detectar defectos en etapas tempranas puede causar que en etapas posteriores se originen más defectos derivados de uno solo. Por ejemplo, un defecto que se escapa en diseño, puede provocar 5 o más defectos en codificación y pruebas.
2. Se pueden aplicar en etapas tempranas del ciclo. Como se ha comentado anteriormente, no solo se puede revisar código, sino cualquier producto de trabajo generado del proceso mismo de desarrollo.
3. Bien implementadas, pueden llegar a ser mucho más eficientes que las pruebas mismas. Es decir, las técnicas puede detectar más defectos por unidad de esfuerzo que las diferentes etapas de pruebas.
4. Son medibles. El esfuerzo requerido para generar métricas e información valiosa de la implementación de las técnicas puede llegar a ser despreciable. Estas mediciones son de gran importancia, ya que permiten conocer qué tan eficientes son las técnicas, cuándo es importante realizar cambios en las mismas y cuáles son más eficientes para determinadas situaciones.

De acuerdo a [3], es importante considerar algunos factores para la implementación de técnicas de detección, que pueden maximizar la efectividad de las mismas. Estos factores son:

1. Mientras mayor sea el esfuerzo de revisión, mayor será el número de defectos detectados. Es una aseveración que a simple vista tiene sentido y que se puede lograr de dos maneras. La primera es aumentando el número de revisores y la segunda es aumentando el esfuerzo individual de cada revisor. Laitenberger y otros [3] definen la probabilidad de encontrar un defecto como:



Donde k es el número de revisores y p la probabilidad de encontrar un defecto, que para este caso, se asume que es la misma probabilidad para todos los revisores. Es importante considerar que existe un efecto techo en el que agregar más revisores no aumentará el número de defectos detectados.

1. Mientras más grande sea el producto de trabajo revisado, mayor será el número de defectos detectados. Se espera que los productos más grandes sean más complejos y tengan más defectos, pero se observa el mismo efecto que en el punto 1. Esto también debido al tiempo de revisión, ya que la efectividad del ser humano se ve afectada negativamente después de un tiempo de realizar una misma actividad que requiera concentración.
2. Entre mayor sea el producto de trabajo realizado, mayor será el esfuerzo de revisión. El comportamiento de este punto es idéntico al observado en el punto 2. Ya que la principal restricción es la efectividad de revisión a medida que el tiempo pasa.

**TIPOS DE TECNICAS**

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.

**REVISION PERSONAL**

De acuerdo a [11], la técnica más importante de detección de defectos es la revisión personal. El proceso de software personal como tal fue desarrollado por Watts Humphrey. En este proceso se hace un énfasis muy marcado en la evaluación, seguimiento y mejora de la calidad personal como elemento vital de la calidad total del producto de software. 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.

Humphrey realza la necesidad de realizar las revisiones personales de cualquier producto con una metodología definida y bajo ciertas condiciones, por lo que se pueden considerar como principios de la revisión personal los siguientes:

1. Revisar personalmente todo el trabajo propio antes de pasar a la siguiente fase de desarrollo.
2. Intentar lo mejor posible corregir todos los defectos antes de dar el producto de trabajo a otra persona en el equipo de desarrollo.
3. Utilizar una lista de chequeo personal y seguir un proceso estructurado de revisión.
4. Seguir las buenas prácticas de la revisión: revisar en incrementos pequeños, hacer las revisiones en papel y hacerlas cuando estás descansado.
5. Medir el tiempo de la revisión, el tamaño de los productos revisados y el número y tipo de defectos encontrados y perdidos.
6. Usar los datos de las mediciones para mejorar el proceso personal de revisión.
7. Diseñar e implementar los productos para que sean fáciles de revisar.
8. Revisar los datos para identificar las formas de prevenir defectos.

**REVISION ENTRE COLEGAS**

Existen varias técnicas de detección de defectos que se realizan entre colegas y/o miembros de un equipo de desarrollo. Estas técnicas se puede clasificar de acuerdo a su formalidad, de acuerdo con [4] existen 6 diferentes tipos de técnicas:

1. Revisión ad hoc.
2. Revisión general.
3. Revisión de parejas.
4. Caminata.
5. Revisión de equipo.
6. Inspección.

**CAMINATA**

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. Como se explica en [5], 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.

[5] expone como ventajas de la implementación de una caminata el que el presentador se prepare explícitamente para realizar la revisión, lo que acelera el proceso. También, al tiempo que no requiere de una preparación para nada exhaustiva de los participantes por adelantado, la audiencia de la presentación se puede familiarizar rápidamente con el producto. Si la audiencia es lo suficientemente diversa, se puede afirmar con alto grado de certeza que la revisión será altamente efectiva.

Al mismo tiempo que se presentan estas ventajas, precisamente ciertas desventajas emanan de ellas. Siendo la principal desventaja el que al no requerirse preparación previa, el grado de entendimiento adquirido por cada participante al momento de la presentación no sea el mismo, causando la falta de profundidad deseada en la revisión.

**INSPECCION**

Es muy probablemente la técnica más formal de detección de defectos de software [4]. De acuerdo a [1] consiste de los siguientes pasos:

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.

La inspección no es una técnica que se pueda limitar solamente a la revisión de código o a una etapa dentro del ciclo de desarrollo, sino que es perfectamente aplicable a los productos de trabajo generados desde el principio y hasta el final del ciclo. En [8] se mencionan 5 diferentes tipos de inspecciones:

1. inspección de arquitectura: Inspección de documentos de requerimientos y especificación.
2. inspección de diseño: Inspección de los documentos de diseño. Que de acuerdo a [6] es la que más reduce el costo de detección de defectos (44%), inclusive por encima de la inspección de código (con un 39%).
3. inspección de código: Inspección de código que aún no ha sido ejecutado.
4. inspección de casos de prueba: inspección aplicada a los casos de prueba diseñados.
5. Inspección de implementación de casos de prueba: Se aplica a los casos de prueba ya implementados o ejecutados.

La implementación de esta técnica de revisión no es algo que se deba tomar a la ligera. Primero que nada es importante decidir de manera adecuada los revisores que estarán involucrados. Si los revisores no son los mejores posibles, puede ser que la efectividad de la técnica se vea reducida considerablemente. De igual manera, al saber que esta técnica es la más formal y rigurosa, es muy probable que sea también la que más tiempo consuma. Esto nos obliga a decidir sobre cuáles productos de trabajo se realizará una inspección. Es prácticamente imposible realizar inspecciones a todos los productos de trabajo, y es necesario escoger los más críticos para el proyecto.

**IMPLEMENTACION DE LAS TECNICAS DE DETECCION**

La implementación de estas técnicas no representa la panacea sobre el aseguramiento de la calidad en productos de software. Como cualquier otra actividad y/o proceso, estas técnicas llevan consigo ventajas y desventajas en su aplicación. En un estudio realizado por Harjumma y otros [4] en el que se consideraron 11 empresas de distintos tamaños (desde 5 empleados hasta varios miles), se realizaron entrevistas estructuradas acerca de las técnicas de detección que implementaban y cuáles identificaban como los principales motivadores y desmotivadotes para llevarlas a cabo.

Se identificaron como los principales motivadores los siguientes:

1. Encontrar más defectos.
2. Encontrar defectos en etapas tempranas.
3. Compartir el conocimiento.
4. Compartir información del proyecto.
5. Mejor administración y seguimiento del proyecto.

Se observa una marcada tendencia sobre el hecho de detectar defectos y hacerlo de manera temprana. En cierto sentido esto contribuye a una mayor calidad del producto de software.

Se identificaron como los principales desmotivadotes lo siguientes:

1. Falta de tiempo.
2. Falta de recursos.
3. Solamente el autor entiende el producto de trabajo.
4. La complejidad de las tareas.
5. Equipos geográficamente distribuidos.

La falta de tiempo y recursos se identifican como los mayores obstáculos para realizar este tipo de actividades. Estos dos obstáculos representan el mayor reto para las empresas, algo que no es nada fácil de resolver.

A su vez, en [8] se definen y establecen situaciones no favorables para la ejecución de revisiones entre colegas. Estas situaciones abarcan la aceptación emocional, problemas de ejecución, tipo de proyecto, calendario, recursos humanos, pequeñas empresas y la organización actual de las mismas. Todas estas desventajas expuestas en [8] hacen replantear la idea sobre la efectividad de las inspecciones creadas originalmente por Fagan y la latente necesidad de modificar su implementación para incrementar su efectividad.

### Personal Software Process

1. **CALIDAD DE 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.

DENSIDAD DE DEFECTOS POR NIVEL DE CMM

|  |  |
| --- | --- |
| **Nivel de CMM** | **Defectos/KLOC** |
| 1 | 7.5 |
| 2 | 6.24 |
| 3 | 4.73 |
| 4 | 2.28 |
| 5 | 1.05 |

Para mejorar la calidad de los productos de software, aparte de utilizar los métodos tradicionales de calidad debemos de medir y dar seguimiento al trabajo personal.

**LA ESTRATEGIA DE CALIDAD DE PSP**

Para producir un sistema de alta calidad, cada parte del sistema debe de ser de alta calidad. La estrategia de PSP 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.

**¿QUÉ ES CALIDAD DE SOFTWARE?**

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

**Calidad de Producto**

El producto debe funcionar. 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.

**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 DEFECTOS**

DEFECTOS POR KLOC EN ETAPAS DE DESARROLLO

|  |  |
| --- | --- |
| **Tipos de Defecto/Origen** | **Defectos/KLOC** |
| Requerimientos | 2.3 |
| Diseño | 1.9 |
| Codificación | 0.9 |
| Documentación | 1.2 |
| Correcciones defectuosas | 1.2 |
| Total | 7.5 |

**PRÁCTICAS PERSONALES DE CALIDAD**

En el PSP, se propone el uso de revisiones de diseño y de código para encontrar defectos antes de entrar en la etapa de pruebas. Cuando se pasa al PSP, también se utilizarán inspecciones. Cuando se realizan las revisiones e inspecciones de forma correcta, los equipos de TSP encuentran en promedio cerca del 99% antes de iniciar con las pruebas de sistema.

**MEDIDAS DE CALIDAD**

Para mejorar la calidad, debes de medir la calidad.

**El “Yield” de la calidad**

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 Costo de la Calidad (COQ)**

Tiene tres componentes principales:

* Failure Costs. El costo total de las fases de compilación y de pruebas.
* Appraisal Costs. El tiempo utilizado en revisiones de código y de diseño.
* Prevention Costs. El costo de identificar causas de defectos y acciones para prevenirlos en el futuro.

Utiliza las siguientes fórmulas:

* Failure COQ: 100 \* (compile time + test time)/(total development time)
* Appraisal COQ: 100 \* (design review time + code review time)/(total development time)
* Total COQ: Appraisal COQ + Failure COQ.
* Appraisal as a % of Total Quality Costs: 100\*(Appraisal COQ)/(Total COQ)
* Appraisal to Failure Ratio (A/FR): (Appraisal COQ)/(Failure COQ)

**Tasa de Revisiones**

Yield y COQ son útiles, pero miden el trabajo que hiciste, no lo que estás haciendo. La tasa de revisión y al radio de fase provee 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 LOC o páginas revisadas por hora.

**Calidad de Radio de Fases**

Es el radio de tiempo utilizado en dos fases de proceso. El radio para la revisión es el tiempo dividido por el tiempo de desarrollo.

**El índice de Calidad de Proceso**

El criterio del índice del proceso de calidad (PQI) es el siguiente (rangos de 0.0 a 1.0):

• Diseño/Codificación.

• Tiempo de revisión de Diseño = 2\*Tiempo de Revisión de Diseño/Tiempo de Diseño.

• Defectos de Compilación/KLOC = 20/(10 + Defectos de Compilación/KLOC)

• Defectos de Pruebas Unitarias/KLOC = 10/(5 + Defectos de Pruebas Unitarias/KLOC)

El PQI se obtiene multiplicando los cinco valores. La meta del PQI es 1.

**ADMINISTRACIÓN DE LA CALIDAD**

Desde un punto de vista de la administración del Yield, el número final de defectos es el residuo de los Yields de todas las fases. El trabajo con mala calidad tiene efectos duraderos, ya que tarde o temprano los errores aparecerán y tendrán que ser corregidos a un alto precio. El trabajo de calidad no solo ahorra tiempo y esfuerzo, si no que produce ahorros económicos durante todo el ciclo de desarrollo.

**ADMINISTRACIÓN PERSONAL DE LA CALIDAD**

Para administrar efectivamente la calidad el software, debes enfocarte tanto en remover defectos como en no inyectarlos. Para el software, el proceso de cambio es la actividad que más puede provocar errores. Otra actividad que puede provocar muchos errores es interpretar los requerimientos y tomar decisiones de diseño. La estrategia para esta situación es reconocer que cada acción tiene la probabilidad de inyectar un defecto. El reto consiste en identificar las fuentes de defectos y prevenirlas, o realizar acciones preventivas para consistentemente detectar y remover los defectos.

El PSP recomienda que la estrategia de administración personal sea usar los planes y los datos históricos para guiar el trabajo. Para empezar se inicia intentando cumplir las guías del PQI, primero produciendo diseños completos.

Finalmente como no se puede esperar un yield del 100% en inspecciones y revisiones de código, también debe de incluirse la parte de pruebas en la estrategia de calidad de software.

**ADMINISTRACIÓN DE LA CALIDAD DE 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.

**PRÁCTICAS PARA LA MEJORA DE PSP**

Existen seis principios:

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.

**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.

1. **REVISIONES DE DISEÑO Y DE CÓDIGO**

**¿QUÉ SON LAS REVISIONES?**

Una inspección es una revisión en equipo estructurada de un producto de software.

Una caminata es un poco menos formal que una inspección, sigue el formato de una presentación, con una audiencia identificando problemas y realizando preguntas.

En una revisión personal, cada persona examina el producto que desarrollo antes que ninguna otra persona. Tu objetivo es encontrar y corregir tantos defectos como puedas, antes de que cualquier implemente, compile, revise, pruebe o incluso lea el programa.

**¿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.

**LOS PRINCIPIOS DE LA REVISIÓN**

Los principios de las revisiones personales son los siguientes:

1. Revisar personalmente todo el trabajo propio antes de pasar a la siguiente fase de desarrollo.
2. Intentar lo mejor posible corregir todos los defectos antes de dar el producto de trabajo a otra persona en el equipo de desarrollo.
3. Utilizar una lista de chequeo personal y seguir un proceso estructurado de revisión.
4. Seguir las buenas prácticas de la revisión: revisar en incrementos pequeños, hacer las revisiones en papel y hacerlas cuando estás descansado.
5. Medir el tiempo de la revisión, el tamaño de los productos revisados y el número y tipo de defectos encontrados y perdidos.
6. Usar los datos de las mediciones para mejorar el proceso personal de revisión.
7. Diseñas e implementar los productos para que sean fáciles de revisar.
8. Revisar los datos para identificar las formas de prevenir defectos.

**EL PROCESO DE REVISIÓN DE CÓDIGO DE PSP**

Seguir el “Code Review Scrip” (falta agregarlo) apoyado por la lista de chequeo de revisión de código.

**LA 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.

**Construyendo la lista de chequeo**

Se necesita que cada persona desarrolle su propia lista de chequeo y que se completen todos los ítems de la lista. Para elaborarla te debes de basar en la tabla de defectos de PSP.

|  |  |  |
| --- | --- | --- |
| **Número de Tipo** | **Nombre del Tipo** | **Descripción** |
| 10 | Documentación | Comentarios, mensajes |
| 20 | Sintaxis | Ortografía, puntuación, tipos, formatos de instrucción |
| 30 | Construcción, Paquete | Cambio de administración, librerías, control de versiones |
| 40 | Asignación | Declaración, nombres duplicados, alcance, límites |
| 50 | Interface | Llamados a procedimientos y referencias, I/O, formatos |
| 60 | Chequeo | Mensajes de error, chequeos inadecuados |
| 70 | Datos | Estructura, contenido |
| 80 | Función | Lógica, apuntadores, ciclos, recursión, computación, defectos en funciones |
| 90 | Sistema | Configuración, tiempo, memoria |
| 100 | Medio Ambiente | Diseño, compilación, pruebas, otros problemas de soporte de sistema |

**Actualizando la lista de chequeo**

Se necesita hacer una distribución de Pareto para tus defectos y actualizarla cada cinco programas para asegurarte que te estás concentrando en el tipo de defectos que pierdes más frecuentemente en las revisiones.

**Revisar contra los estándares de codificación**

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.

**Utilizando la lista de chequeo**

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.

**La estrategia de revisión**

* 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.

**REVISIONES DE DISEÑO**

Los problemas de diseño son muy difíciles de encontrar en código fuente. Para hacer productos de alta calidad, se tiene que revisar el diseño antes de implementar el código.

**PRINCIPIOS DE LA REVISIÓN DE DISEÑO**

L siguientes recomendaciones:

* Producir diseños que puedan ser revisados.
* Seguir una estrategia de revisión.
* Revisar el diseño en etapas.
* Si el diseño está incompleto, revisar el código.
* Verificar que la lógica implemente correctamente los requerimientos.
* Checar problemas de seguridad.

**MEDIDAS DE REVISIÓN**

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.

**Estimados de Yield**

Para hacer un estimado de yield para cualquier fase, debes de asumir el yield de la fase anterior.

**Tasa de revisión**

LOC/hora es una métrica útil e instantánea.

**Efectividad de la revisión**

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

**PROBLEMAS EN LA REVISIÓN**

Para obtener el máximo beneficio de las revisiones personales estas deben de hacerse de la forma adecuada, ya que hasta los desarrolladores más experimentados inyectan alrededor de 100 defectos por KLOC.

**Prácticas de revisión**

* Tomar un descanso después de programar.
* Imprimir el programa. Revisar en la pantalla es una pérdida de tiempo porque te centras en una parte muy pequeña del programa lo que hace que pases por alto muchos errores.
* Revisar cuidadosamente cada ítem de la lista de chequeo.
* Actualizar periódicamente las listas de chequeo de diseño y de código.
* Desarrollar y utilizar una lista de chequeo diferente para cada lenguaje de programación.

**Eficiencia de la revisión**

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

**Objetivos de la revisión**

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

**La relación entre las revisiones y las inspecciones**

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.

1. **DISEÑO DE SOFTWARE**

**¿QUÉ ES EL DISEÑO?**

**El poder de las abstracciones**

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 problema con las abstracciones**

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.

**Resolviendo el problema de las abstracciones**

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.

**¿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.

**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.

**La incertidumbre de los requerimientos**

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.

**Prototipado**

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.

**NIVELES DE DISEÑO**

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 si 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.

**Definición de requerimientos**

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:

• 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.

Así que se debe de conocer tanto como se pueda de los requerimientos. Una vez que se tiene la definición completa se debe de comenzar con el diseño.

**La especificación del diseño**

Se necesita un esfuerzo considerable para subdividir las funciones del sistema en componentes coherentes y especificar sus relaciones. Si las decisiones de diseño no son especificadas cuando se tomaron, tendrán que re crearse después. Intentar documentar estas decisiones después es difícil, propenso a errores y muy tardado. Cada que se haya aclarado algún aspecto de los requerimientos, debe de ser rigurosamente especificado. Si la especificación tiene conflictos con especificaciones previas, se deben de hacer cambios lo antes posible, cada retraso puede retrasar el trabajo de diseño o provocar errores. Entre más precisa sea la especificación, es más probable identificar errores en los requerimientos, omisiones e inconsistencias.

**Diseño de alto nivel (HLD)**

Aquí se crea el diseño general. Esto involucra decisiones acerca de reusar componentes existentes, definir clases y especificar relaciones entre las clases. El diseño de alto nivel debe de considerar dos sistemas: El sistema de aplicación y la librería de componentes reusables. Aquí también se toman decisiones acerca del contenido del producto, el costo de desarrollo y el calendario. Aunque no desarrolles un sistema con todo lo que el usuario desea, debes de proveer un sistema el cual sea lo que el cliente necesite y pueda pagar.

**Diseño detallado (DLD)**

Aquí se reduce el HDL a una forma implementable detallando funciones, especificando máquinas de estado y produciendo el diseño terminado.

**Diseño de implementación (ILD)**

La tarea de este nivel de diseño es escribir el código. Aunque normalmente no se considera escribir código como una actividad de diseño, hacer esto nos puede generar preguntas acerca del diseño, algunas de estas provocando cambios en el diseño. En cierto sentido es la primera prueba real del diseño.

**ESTRATEGIAS DE DISEÑO Y DESARROLLO**

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

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

**Estrategias de desarrollo**

Algunas estrategias de desarrollo son las siguientes: Progresiva, mejoras-funcionales, fast-path y dummy.

**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 fast-path**

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 dummy**

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.

**CALIDAD EN EL DISEÑO**

El diseño de software debe de contener una solución completa y precisa al problema.

**Precisión en el diseño**

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.

**Completez del diseño**

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 necesidades del usuario**

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 que 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).

1. **PLANTILLAS DE DISEÑO**

**REPRESENTACIÓN DE DISEÑO**

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

Notación lógica precisa

Una vez generado el diseño se debe analizar su corrección, completez y consistencia. La notación usada para el diseño debe ser concisa y precisa, y debe de cumplir con las siguientes características:

* 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.

**LAS PLANTILLAS DE DISEÑO**

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:

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 PSP .

**OPERATIONAL SPECIFICATION TEMPLATE (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.

**FUNCTIONAL SPECIFICATION TEMPLATE (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 (constraints). Las partes pueden ser clases, módulos o incluso un sistema entero. El FST provee una descripción detallada de cada una de esas partes.

**STATE SPECIFICATION TEMPLATE (SST)**

Esta plantilla provee una forma simple y precisa de especificar el comportamiento de los estados de un sistema, el cuál es determinado por sus entradas y el estado actual. Programas con múltiples estados son denominados máquinas de estados.

**LOGIC SPECIFICATION TEMPLATE )(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.

**USANDO LAS PLANTILLAS DE DISEÑO DE PSP**

El objetivo principal de las plantillas es el registrar el diseño, no ayudar a producirlo. Estas plantillas definen qué hará el programa y cómo se implementará. Se recomienda que al adoptar estas plantillas, aún se haga el diseño como se solía hacer, y a la vez registrándolo en las plantillas, hasta que la persona esté familiarizada con las plantillas.

**Comparación con UML**

OST: Combinación de diagramas de casos de uso, descripción de casos de uso, diagramas de actividades y diagramas de secuencia.

SST: Diagrama de estados.

FST: Diagrama de clases.

LST: No tiene equivalente en UML.

Rediseño

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.

**USANDO LAS PLANTILLAS EN DISEÑOS A GRAN ESCALA**

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.

1. **VERIFICACION 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

**¿PORQUÉ VERIFICAR PROGRAMAS?**

Normalmente los desarrolladores tienen la creencia que pocos defectos en un programa son aceptables mientras el programa se ejecute normalmente bien. 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.

**ESTANDARES DE DISEÑO**

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.

**VERIFICACION POR TABLA DE EJECUCION**

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.

**VERIFICACION 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.

**VERIFICACION DE MAQUINAS 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.

**VERIFICACION 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.

**CONSIDERACIONES EN LA VERIFICACIÓN**

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.

### Clasificación Ortogonal de Defectos

Normalmente, los defectos representan el aspecto no deseado en el manejo de calidad del software. Una herramienta muy útil en este aspecto es el *Root Causal Analysis (RCA)*, ya que provee un gran nivel de detalle en cada defecto, pero normalmente este tipo de análisis toma mucho tiempo y esfuerzo. También existe el *Growth Modeling,* el cual provee una forma fácil y rápida de detectar patrones y tendencias, pero no tiene un modelo robusto para la captura de defectos.

Una herramienta que nos permite atacar las debilidades de los modelos antes mencionados es la Clasificación Ortogonal de Defectos, la cual está diseñada para capturar todos los atributos de los defectos y permitirnos hacer análisis matemáticos sobre ellos.

La clasificación ortogonal de defectos nos permite clasificar 8 tipos de atributos diferentes, que están divididos en 2 clases:

***Opener Section:***

Estos atributos son registrados cuando se encuentra el defecto, y antes de resolverlo:

* Actividad: Se refiere a la actividad que se estaba realizando cuando se encontró el defecto. Un ejemplo de estas actividades sería inspecciones de código, pruebas unitarias, pruebas de sistema, etc.
* *Trigger:* Se refiere al ambiente o condición existente que ocasiona que el defecto aparezca, aquella condición necesaria para reproducir el defecto. El método ya tiene algunos activadores definidos para cierto tipo de actividades, aunque también se pueden usar los propios.
* Impacto: Aquí se selecciona el impacto que el defecto podría haber tenido en el cliente final si no se arreglara. En caso de que el defecto lo haya detectado un cliente, se pone el impacto actual que el defecto tuvo en el cliente.

***Closer Section:***

Estos atributos son registrados después de que el defecto ha sido arreglado.

* *Target:* Representa la entidad de alto nivel que se arregló. Por ejemplo, el diseño o el código del programa.
* Tipo de defecto: Representa la naturaleza de la corrección que se realizó. Para este campo existen valores predeterminados, por ejemplo: asignación o inicialización, que se refiere a que el valor de una variable fue corregido; algoritmo/método, significa que se implementó o corrigió un algoritmo; tiempo/serialización, que nos indica que se tuvo que implementar métodos de serialización en un recurso compartido.
* *Qualifier:* Aplica al tipo de defecto, y captura el calificativo que describe al defecto. Para este atributo existen 3 valores posibles:
  + Omisión*:* Indica que el defecto se debió a una omisión.
  + Incorrecto: Indica que valores incorrectos se usaron.
  + Extraño: Indica que el defecto se debe a algo no relevante al código.
* Fuente: Identifica el origen del *target* que tenía el defecto. Para este atributo también existen ciertos valores predeterminados:
  + Desarrollado internamente: Nos dice que el defecto fue introducido por el equipo de desarrollo
  + Reusado de una librería: Nos indica que el defecto pertenecía a una librería de reuso.
  + *Outsourced:* Nos indica que el defecto fue introducido por un *vendor****.***
  + *Ported:* Nos indica que el defecto tiene que ver con una parte que fue validada en un ambiente diferente.
* Edad: Identifica el historial del *target* que tenía el defecto. Para este campo existen los siguientes valores:
  + Base: Nos indica que el defecto está en un aparte del producto que no ha sido modificada en el proyecto actual, y no es parte de una librería de reuso. Es un defecto latente.
  + Nuevo: El defecto fue introducido en el proyecto actual.
  + Re-escrito: El defecto fue introducido por re-diseñar o re-escribir una función con el objetivo de mejorar su diseño o calidad.
  + Re-arreglado: El defecto fue introducido al proveer una solución a un defecto previo.

# Metodología

La tesis constará de dos partes básicas. Una herramienta de software con las funcionalidades propuestas en el Concepto de Operaciones (ver Anexo 1), y tres documentos de tesis, los cuales contendrán la base teórica que sustentará conceptualmente el diseño de la herramienta.

El grupo de tesistas; Eduardo Campos, Humberto García y Marco Rangel; sostendrá una reunión semanal con el director de tesis, Óscar Mondragón, para presentar los avances hechos durante la semana en los rubros de investigación y elaboración del sistema.

En las reuniones semanales, los tesistas presentarán el avance realizado durante la semana, se harán las revisiones pertinentes y se decidirá el trabajo por realizar a futuro. A continuación se detalla la metodología específica del proceso de investigación y del proceso de desarrollo.

## Investigación

La investigación tendrá tres temas principales, los cuales serán cubiertos por cada tesista, estos temas son:

* El costo de la calidad.
* Comparación y efectividad de técnicas de remoción de defectos.
* Caracterización, clasificación y seguimiento de los defectos.

Cada tema será presentado de una forma práctica, esto significa que la investigación se utilizará para sustentar la elaboración de un sistema de software, y se aprovechará el proceso de desarrollo de software para comprobar la práctica de la aplicación propuesta. El proceso a utilizar será el siguiente:

1. **Establecimiento del Marco Conceptual.** Se establece el marco teórico, las teorías y métodos que apoyan el tema práctico, y se dan las definiciones y los conceptos. En otras palabras es la investigación preliminar.
2. **Elaboración de la propuesta básica del tema.** Se aplican los métodos y conceptos teóricos tanto en el proceso de desarrollo del sistema de software, así como en el diseño y los componentes que el sistema contendrá. Se elabora la propuesta práctica y los logros que se pretenden tener con esta.
3. **Presentación del caso práctico a comprobar.** Se concentra la propuesta práctica y se comprueba la efectividad de su aplicación.
4. **Logros y conclusiones.** Se llegan a las conclusiones.

## Desarrollo de Software

La metodología utilizada durante el desarrollo de software será una metodología en cascada, agregando actividades de calidad en la parte de implementación y pruebas. Es importante destacar que en el proceso de desarrollo se seguirán estas técnicas de calidad con los siguientes objetivos:

* Hacer una clasificación ortogonal de los defectos encontrados durante el proceso de desarrollo.
* Hacer un análisis de la efectividad de las técnicas de detección de defectos: Revisiones personales de código y caminatas.
* Hacer un análisis del costo de la calidad, tomando en cuenta el esfuerzo de las técnicas de detección de defectos y el esfuerzo de la remoción de defectos.

Para el proceso de desarrollo se hará un reporte semanal de los avances realizados, así como un reporte mensual en el cual se pueda observar la información acerca de en qué punto del proceso de desarrollo se encuentra el sistema, así como el esfuerzo que se requiere para que este sea completado.

Los pasos de la metodología con sus productos de trabajo son los siguientes:

Este proceso se repetirá de forma cíclica para cada paquete de trabajo.

# Resultados Esperados

De la realización del presente proyecto se esperan los siguientes resultados:

1. Un documento de tesis completo y correcto para los temas de investigación desarrollados. El documento deberá contener:
   1. Un planteamiento del problema a tratar. Este planteamiento explicará suficientemente el estado en el que se encuentran las principales herramientas de apoyo para actividades de calidad en el desarrollo de software.
   2. La justificación de la realización y desarrollo de la tesis, la herramienta y los temas de investigación.
   3. Un marco teórico sobre las actividades de calidad a implementar en la herramienta y sobre los temas de investigación escogidos.
   4. Exposición de los resultados obtenidos en la realización del proyecto.
2. La documentación de diseño y especificación de la herramienta:
   1. Documento de concepto de operaciones
3. Una herramienta de software con las funcionalidades especificadas en la documentación. La herramienta deberá contener (todo especificado del documento de concepto de operaciones):
   1. El ambiente en el que se verá la herramienta.
   2. Los diferentes módulos.
   3. Las actividades de desarrollo y calidad que podrán realizar los usuarios.
   4. Los modos de operación especificados.
   5. Los escenarios de operación especificados.
   6. Las funcionalidades especificadas para cada modo de operación.
   7. La generación de estadísticas y reportes.

# Bibliografía

[1] K. Owens, “Software Detailed Technical Reviews: Finding and Using Defects”, in *Conference Proceedings Wescon/97*, 1997, doi: [10.1109/WESCON.1997.632327](http://0-dx.doi.org.millenium.itesm.mx/10.1109/WESCON.1997.632327)

[2] Boehm, Barry, *Software Engineering Economics*, 1981.

[3] O. Laitenberger et al, “Quantitative Modeling of Software Reviews in an Industrial Setting”, in *Sixth international Software Metrics Symposium*, 1999, doi: [10.1109/METRIC.1999.809752](http://0-dx.doi.org.millenium.itesm.mx/10.1109/METRIC.1999.809752)

[4] L. Harjumma et al, “Peer Reviews in Real Life – Motivators and Demotivators”, in *Fifth International Conference on Quality Software*, 2005, doi: [10.1109/QSIC.2005.48](http://0-dx.doi.org.millenium.itesm.mx/10.1109/QSIC.2005.48)

[5] G.M. Weinberg and D.P. Freedman, “Reviews, Walkthroughs, and Inspections”, *IEEE Transactions on Software Engineering*, vol. SE-10, 1984.

[6] C. Wohlin et al, “Software Inspection Benchmarking – A Qualitative and Quantitative Comparative Opportunity”, in *Eighth IEEE Symposium on Software Metrics*, 2002, doi: [10.1109/METRIC.2002.1011331](http://0-dx.doi.org.millenium.itesm.mx/10.1109/METRIC.2002.1011331)

[7] O. Laitenberger et al, “Software Inspections, Reviews & Walkthroughs”, in *Proceedings of the 24rd International Conference on Software Engineering*, 2002, doi: [10.1109/ICSE.2002.146316](http://0-dx.doi.org.millenium.itesm.mx/10.1109/ICSE.2002.146316)

[8] G. Chroust and H. Lexen, “Software Inspections – Theorie, New approaches and an Experiment”, in *Proceedings, 25th EUROMICRO Conference*, 1999, doi: [10.1109/EURMIC.1999.794793](http://0-dx.doi.org.millenium.itesm.mx/10.1109/EURMIC.1999.794793)

[9] T. Hall et al, “Towards Implementing Successful Software Inspections”, in *Proceedings, International Conference on Software Methods and Tools*, 2000, doi: [10.1109/SWMT.2000.890428](http://0-dx.doi.org.millenium.itesm.mx/10.1109/SWMT.2000.890428)

[10] P. Runeson et al, “What Do We Know about Defect Detection Methods?”, in *IEEE software*, vol. 23, pp. 82-90, 2006

[11] W. S. Humphrey, *PSP A Self-Improvement Process for Software Engineers*, Westford, Addison – Wesley, 2009.

[12] IBM Research Center for Software Engineering*, Orthogonal Defect Classification,* Febrero 1, 2002.

[13] W. S. Humphrey, *Winning with Software An Executive Strategy,* Westford, Addison – Wesley, 2002.

[14] Ljubomir Lazic, Amel Kolasinac, Deznan Avdic, *The Software Quality Economics Model for Software Project Optimization*, University of Novi Pazar, 2009

# Anexos

## Concepto de Operaciones

El presente documento refiere al concepto de operaciones para el sistema de administración de calidad de proyectos de software, cuyo nombre clave es “Bug Manager” (BM). Contiene:

* Los objetivos, antecedentes y políticas del sistema actual.
* Las áreas de oportunidad del sistema actual y justifica la necesidad de realizar uno nuevo.
* Las características, metas, restricciones y políticas del BM.
* Los escenarios operacionales del BM.
* El impacto organizacional y operacional del BM.

