HACIA LA CONSOLIDACIÓN DE UN MÉTODO UNIVERSAL DE CALIDAD DE SOFTWARE

JUAN CARLOS MARÍN RINCÓN

TRABAJO DE GRADO

NORMAN DANILO CASTRO TELLES
COORDINADOR MAESTRÍA EN INGENIERÍA DE SISTEMAS

INSTITUCIÓN UNIVERSITARIA POLITÉCNICO GRANCOLOMBIANO
FACULTAD DE INGENIERÍA Y CIENCIAS BÁSICAS
MAESTRÍA EN INGENIERÍA DE SISTEMAS
BOGOTA
2014

1. Dedicatoria.

Este trabajo va dedicado a todas las personas que me han apoyado y motivado durante el duro camino de una maestría.

A mis padres que con su amor, esfuerzo y paciencia me han dado el apoyo moral y económico necesario para llevar a cabo el sueño de hacer y terminar una maestría. Gracias a ustedes entre a recorrer este camino y por ustedes termino de recorrerlo.

A mi novia Lorena quien también me ha apoyado moralmente en los momentos más complicados de mi vida, gracias a Dios por ponerte en este hermoso camino que es la vida y espero poder compartir el resto de mi vida contigo.

Y por último a todos mis amigos quienes me han ayudado con ideas, grandes palabras y sobre todo apoyo moral en los momentos más difíciles.

Sin ustedes esto un hubiese sido posible…

2. Agradecimientos.

Una tesis de maestría no se hace todos los días, es un esfuerzo y sacrificio grande, pero al conseguir el objetivo, es donde todo el sacrificio y esfuerzo se ve totalmente recompensado.

Primero quiero agradecerle a Dios por regalarme el don de la vida, dame salud y un buen trabajo y gracias a ello puedo cumplir con esta gran meta que me he planteado en mi vida.

A mis padres que han tenido toda la paciencia para sacarme adelante y me han aconsejado y apoyado en todas las decisiones que he tomado.

A mi novia Lorena que ha sido un gran apoyo moral para salir adelante en los momentos más difíciles de mi vida. Gracias a su apoyo y amor incondicional, el esfuerzo ha sido más gratificante.

A todos los profesores de la maestría que me han brindado su conocimiento durante mi paso por la universidad. Gracias a ustedes soy mejor persona.

A mi director de tesis, Danilo Castro, por su apoyo incondicional y guiarme durante este largo camino de la calidad de software.

A mis amigos cuyo apoyo y consejos han sido muy importantes para poder recorrer este camino.

A todos ustedes muchas gracias, sin ustedes esto no hubiese sido posible.

3. Contenido.

4. Introducción.

Todo proyecto de software tiene cuatro aspectos fundamentales: el tiempo, los costos, el alcance y la calidad del producto. En la mayoría de los casos se tienen en cuenta tres de estos cuatro aspectos y normalmente se excluye parcialmente la calidad del producto con el fin de terminar el proyecto en el plazo y costos determinados inicialmente. Sin embargo esta decisión deriva en atrasos que traen consigo aumento de costos debido a la baja calidad de un producto que no satisface las expectativas funcionales del cliente lo que muchas veces implica el cobro de pólizas de incumplimiento de contratos.

La calidad de un producto de software tiene diferentes perspectivas: la perspectiva de usuario enfocada a la usabilidad del producto y los beneficios que le produce a su operación; la perspectiva del fabricante se enfoca en los defectos de la aplicación con respecto a las especificaciones acordadas (Kitchenham & Pfleeger, 1996).

Aunque muchas empresas cuentan con un grupo de desarrolladores altamente capacitados y aplican procesos de aseguramiento de calidad por medio de pruebas unitarias y pruebas de tipo funcional ejecutadas por el equipo de pruebas funcionales, el resultado sigue siendo, en la mayoría de los casos, un producto con muchos defectos. Este factor crea la necesidad de aplicar diversas técnicas durante todo el ciclo de desarrollo de software con el fin de poder eliminar defectos en etapas tempranas.

En la actualidad existen varios métodos y herramientas para el aseguramiento de la calidad, una de estas es Squid, basada en los estándares ISO 9126 y está descrita por Boeng, Depanfilis, Kitchenham y Pasquini en su artículo (Boegh, Depanfilis, Kitchenham, & Pasquini, 1999). También existen modelos de estimación de calidad como CART, S-PLUS y la regresión logística, entre otros (Khoshgoftaar & Seliya, 2004). Esto nos hace pensar en la necesidad de encontrar un modelo estándar de calidad aplicable con el fin de mejorar los atributos de calidad de un producto de software. Dentro de los modelos para evaluar la calidad de un producto encontramos el modelo de McCall, Boehm, FURPS, ISO 9126, Dromey entre otros (Ortega, Pérez, & Rojas, 2003).

Existen diversos factores que afectan la calidad de software tanto en las fases de diseño por medio de métricas de diseño orientado a objetos (Brito e Abreu & Melo, 1996), como por medio de diseños racionales (Briand, Wüst, Daly, & Porte, 2000) (Tang, Tran, Han, & van Vliet, 2008) y el desarrollo distribuido en grandes distancias (Bird, Nagappan, Devanbu, Gall, & Murphy, 2009).

5. Objetivos e Hipótesis.

5.1. Objetivo General.

Fundar las bases para consolidar un método universal que permita asegurar la calidad de un producto de software antes de este ser entregado al cliente o usuario final.

5.2. Objetivos Específicos.

Dentro de los objetivos específicos tenemos:

- Investigar sobre las técnicas de la metodología ágil, junto con sus ventajas y desventajas.
- Proponer algunas técnicas de ámbito gerencial para mejorar la calidad de un producto de software.
- Aplicar las técnicas de inspecciones de código y desarrollo dirigido por pruebas en un equipo de trabajo experimental.
- Analizar los datos recolectados para fundar las bases necesarias de consolidación de la metodología.

5.3. Hipótesis.

La aplicación del diseño de software, junto con la experiencia en programación de los desarrolladores, que componen un equipo de desarrollo, juega un papel determinante en la calidad final del producto.

6. Marco Teórico.

El axioma fundamental de un producto de software de calidad es un producto tangible con características y propiedades internas, definidas por los desarrolladores, que determinan sus atributos externos de calidad (Geoff Dromey, 1996).

López, en su tesis de grado "*Hacia la Identificación de un Método Universal de Calidad de Software"* (López, 2013), propone un proceso de desarrollo de software que involucra varias prácticas pertenecientes a la metodología ágil en donde la frecuencia de las prácticas de aseguramiento de calidad ocurren más a menudo con respecto a los métodos tradicionales como la cascada y también ocurren en momentos más tempranos del ciclo de desarrollo (Huo, Verner, Zhu, & Babar, 2004). Dicha investigación incluye las prácticas de inspecciones de código, la programación en parejas, el desarrollo dirigido por pruebas y el modelo de pruebas en V. Adicionalmente a estas prácticas pertenecientes al área técnica, se propone incluir y evaluar el impacto de otras prácticas como lo son la motivación del equipo de trabajo y la especialización de trabajo:

6.1. Moral del Equipo de Trabajo.

Mantener alta la moral del equipo es muy importante dado que al bajar, el rendimiento y la productividad del equipo disminuyen afectando la calidad de los artefactos realizados. Por este motivo es necesario mantener la moral alta, informando al equipo las decisiones tomadas, los inconvenientes que tenga la empresa, ofreciendo posibilidades de capacitación a los empleados y manteniendo un buen ambiente de trabajo en las mejores condiciones posibles (McConnell, Code Complete. A Practical Handbook of Software Construction, 2004).

Esta técnica tiene está muy ligada a la empresa y su organización, existen varias métricas a tener en cuenta (Nagappan, Murphy, & Basili, 2008):

- **Número de ingenieros activos en la compañía:** entre más alto sea el número de ingenieros que realizan modificaciones conjuntas sobre el mismo fragmento de código, la probabilidad de inyectar defectos es mayor y se necesitará una mejor organización y comunicación por parte del equipo de trabajo. Según Brooks (Brooks, 1995) el número de comunicaciones teóricas para cada uno de *N* ingenieros se calcula con la fórmula:

$$\frac{(N * (N-1)}{2}$$

Formula 1: Número de comunicaciones para cada ingeniero

Teniendo en cuenta esta fórmula, podemos ver que a medida que el equipo es más grande, se pueden presentar más problemas de comunicación entre los ingenieros ocasionando errores en el diseño, daños en los fragmentos de código de los otros ingenieros y problemas de entendimiento de los artefactos de diseño.

- **Número de ex – ingenieros:** esta medida nos da el número total de ingenieros que trabajaron en un fragmento de código y que han abandonado la empresa. Esta medida toma importancia en el momento de la realización de la transferencia de conocimiento, dado que el nuevo recurso que retome el trabajo del ingeniero que se retira, puede no estar familiarizado con los fundamentos de diseño, con los razonamientos de ciertas correcciones y/o con las relaciones con otros fragmentos de código.

  Para mitigar un poco el impacto de la renuncia de un ingeniero, las empresas deben tener planes de contención y no recargar el conocimiento de un módulo a un solo ingeniero, de esta manera la transferencia de conocimiento sea menor y mucho más sencilla que realizar la transferencia a una persona que no conoce nada del módulo o de la aplicación. Otra ventaja es que la solución de incidencias o realización de controles de cambio es mucho más sencilla dado el entendimiento del ingeniero sobre el módulo asignado.

- **Frecuencia de edición:** este indicador nos presenta la cantidad de veces que un código fuente es modificado por parte de los desarrolladores, comenzando cuando es descargado o actualizado desde el repositorio de integración continúa hasta que las modificaciones son confirmadas en el repositorio. Esta medida nos muestra una posible inestabilidad en determinado fragmento de código y muestra la distribución de las modificaciones por parte del equipo de trabajo.

Muchos gerentes de proyectos mencionan que la clave en la motivación del equipo de trabajo está en: la gente que está sentada en diferentes ubicaciones del sitio de trabajo, debe conocerse cara a cara con las otras personas, deben sentir que no hubiesen podido lograr los objetivos logrados sin el trabajo en equipo y que en algún momento recibieron el apoyo del grupo completo en situaciones de ignorancia con respecto a temas técnicos o de negocio (Cockburn, 2000).

Sin embargo existen otros factores que incrementan la motivación del equipo de trabajo, dado que un equipo de trabajo motivado tiene un rendimiento superior y la probabilidad de inyección de defectos es menor (McConnell, Rapid Development. Taming Wild Software Schedules, 1996):

- **Buen ambiente de trabajo:** la mejor manera de motivar a un desarrollador es proveerle un buen ambiente que le permita enfocarse y cumplir con las asignaciones que esté tenga.

- **Apropiación:** las personas trabajan más duro para conseguir sus propias metas. Por esto la apropiación de las responsabilidades por parte del desarrollador es importante para la consecución de los objetivos.

- **Metas bien establecidas:** este es un factor muy alto de productividad del equipo de desarrolladores, los desarrolladores van a cumplir con los objetivos que se les plantea, ni más, ni menos, siempre y cuando este objetivo este bien planteado conforme con el tiempo que se asigna al cumplimiento de este. En un experimento llevado a cabo por Weinberg y Schulman, se le dio a 5 equipos de desarrollo el mismo problema pero con objetivos principales diferentes entre ellos, el resultado mostro de 4 de los 5 equipos terminaron primero el objetivo primario y ningún equipo realizó consistentemente los objetivos primarios y secundarios, por lo consiguiente se determinó que los equipos de desarrollo tienen una alta motivación a la consecución del objetivo.

  Uno de los principales defectos de los líderes de equipo es diluir los esfuerzos del equipo de desarrollo con la asignación de demasiadas prioridades. Para obtener mejores resultados, el líder de equipo debe seleccionar uno de los objetivos como el más importante y no varios.

- **Posibilidades de crecimiento:** la motivación por las posibilidades de crecimiento nacen por la misma naturaleza cambiante de este campo de la ingeniería. Una empresa puede mostrar interés en el crecimiento de los desarrolladores de las siguientes formas: proveer una restitución de matrícula para el desarrollo profesional, dar el tiempo libre para asistir a las clases o para estudiar, proporcionar un reembolso por la compra de libros profesionales, asignación de los desarrolladores a proyectos que ampliaran sus habilidades, asignación de mentores a los desarrolladores nuevos y evitando la presión por cronograma y horarios.

  Los costos a invertir en los desarrolladores no tienen límites superiores, Nissan invirtió aproximadamente $30.000 USD por persona en entrenamientos de inducción para abrir su fábrica en Smyrna, Tennesse.

- **Vida personal:** este es uno de los factores motivacionales más difíciles de entender, dado que al aumentar las responsabilidades del desarrollador, se tienen a apretar los cronogramas haciendo que este pierda paulatinamente la vida personal lo cual terminará afectándolo motivacionalmente y llevándolo a pensar que el premio que le dio el gerente de proyectos o el líder de equipo es más un castigo. Para esto el líder de equipo o gerente de proyecto debe

respetar las vacaciones del desarrollador y ser sensible a las peticiones ocasionales para ausentarse en el transcurso del día laboral.

6.2. Especialización del Trabajo.

El impacto de la especialización del trabajo recae sobre la calidad del producto que se está desarrollando. Cuando una empresa dedicada a la creación de productos de software tiene un equipo de trabajo con determinadas características y conocimientos técnicos e inicia la etapa de arquitectura de determinada aplicación, es extremadamente importante tener en cuenta los conocimientos técnicos del equipo para determinar las tecnologías a utilizar en la construcción de dicha aplicación.

Según datos de Cocomo II, los desarrolladores que han trabajado por 3 años o más en un lenguaje de programación determinado, son aproximadamente un 30% más productivos que trabajando en un lenguaje que no conocen. Adicionalmente en un estudio realizado por IBM (McConnell, Code Complete. A Practical Handbook of Software Construction, 2004), los programadores con una amplia experiencia en un determinado lenguaje de programación son 3 veces más productivos que los que no tienen mucha experiencia en el mismo lenguaje.

Los desarrolladores que trabajan en lenguajes de alto nivel adquieren una mayor productividad y calidad en sus desarrollos que los desarrolladores que trabajan en lenguajes de bajo nivel. Lenguajes como C++, Java, Smalltalk, y Visual Basic han sido acreditados como lenguajes de alta productividad debido a su sencillez, fiabilidad y su alto nivel de comprensión por parte de las personas que los aprenden en factores de 5 a 15.

Por otro lado, los lenguajes de alto nivel son más expresivos que los lenguajes de bajo nivel. La tabla 1 muestra las proporciones típicas de varios lenguajes de alto nivel con respecto al lenguaje C, en donde cada nivel indica que cada línea de código en el lenguaje hace más que una línea de código en C.

Lenguaje	Nivel relativo a C
C	1 a 1
C++	1 a 2.5
Fortran 95	1 a 2
Java	1 a 2.5
Perl	1 a 6
Smalltalk	1 a 6
SQL	1 a 10
Visual Basic	1 a 4.5

Tabla 1: relación de expresividad de líneas de código de lenguajes de alto nivel contra C. Tomada del libro de Steven McConnell (McConnell, Code Complete. A Practical Handbook of Software Construction, 2004).

El tener un equipo de trabajo altamente productivo implica un aumento en la calidad del producto a desarrollar debido al alto conocimiento técnico del equipo en el lenguaje que se está trabajando sumado con la cantidad de defectos que se pueden inyectar por desconocimiento del lenguaje. Esto muestra claramente que es necesario tener en cuenta los conocimientos técnicos del equipo de trabajo en el momento de diseñar la arquitectura de una aplicación.

Aunque muchas veces se utilizan tecnologías nuevas en el desarrollo de una aplicación, es muy importante tener en cuenta en las estimaciones de tiempo de desarrollo, la curva de aprendizaje sobre estas nuevas tecnologías. En muchos casos esto no se tiene en cuenta en las estimaciones ocasionando atrasos en las entregas o entregas incompletas con bajos niveles de calidad por la ausencia de pruebas por parte del equipo de calidad del proyecto.

6.3. Programación por Parejas.

Según Kent Beck en su libro (Beck, 1999), la programación en parejas (pair programming en inglés) su objetivo es que toda la producción de código se realiza entre dos personas sentadas delante de una pantalla con un teclado y un ratón.

En esta técnica uno de los integrantes toma el control del teclado y el ratón y se encarga de realizar la mejor solución al problema (conductor), mientras el otro integrante se encarga de tomar una postura más estratégica en donde se cuestiona si el planteamiento propuesto va a funcionar, que casos de prueba podrían fallar y de qué manera se podría simplificar la solución (navegante). De esta manera, el conductor y el navegante participan en un constante dialogo transversal a todas las etapas del ciclo de desarrollo (diseño, implementación y pruebas unitarias) tomando las decisiones pertinentes para llegar a la solución del problema.

Las características principales de esta técnica son:

- **Su carácter dinámico:** si dos personas se emparejan en la mañana, en la tarde podrían emparejarse con otras personas sin ningún problema, en caso que el programador no tenga conocimiento de negocio para resolver su actual asignación, este puede hacer pareja con alguien que tenga el conocimiento suficiente para cumplir con el objetivo. Esta rotación no solo implica realizar el cambio de compañero, sino también de rol (Bryan, du Boulay, & Romero, 2006).

- **Compañerismo:** la programación en parejas puede eliminar las disputas entre los integrantes del equipo y aumentara la comunicación entre ellos, si se encuentran frescos y descansados.

- **Alineamiento:** las parejas pueden alinear su entendimiento del problema antes de comenzar con la implementación mediante la codificación conjunta de los casos de prueba.

- **Depuración:** cuando un programador codifica solo, está expuesto a cometer más errores y a sobredimensionar el diseño, sobre todo cuando se está trabajando bajo presión.

- **Conocimiento distribuido:** el trabajar en parejas, las dos personas comparten el conocimiento de la actividad realizada (Raymond, 2013).

Esta técnica no implica una sesión de tutoría, muchas veces las parejas están conformadas por un programador con menos experiencia que el otro lo que genera retrasos en la finalización de la tarea, dado que el programador con menos experiencia comenzara a realizar muchas preguntas y probablemente cometerá errores simples como ausencia de caracteres como paréntesis y puntos y comas.

Esta técnica es bastante útil para encontrar errores simples que se pueden convertir en defectos, en el momento de arranque de un proyecto o para la solución de problemas de alta complejidad. También ayuda a generar un código más limpio y legible a otros programadores (Hulkko & Abrahamsson, 2005). Adicionalmente, esta técnica tiene un alto impacto en la identificación de defectos en la fase de desarrollo. Jones afirma que el porcentaje de eficiencia que un desarrollador solitario encuentre y remueva los defectos de su código es inferior al 50% (Jonnes, Software Quality in 2012: A Survey of the State of the Art, 2012).

6.4. Modelo en Cascada.

El modelo en cascada (Boehm, 1981) habla de la evolución del producto a través de una secuencia de fases lineales en donde se permiten iteraciones al estado anterior, dentro de las fases más comunes se encuentran:

- **Análisis de requerimientos del sistema:** en donde se realizan los análisis correspondientes a los requerimientos funcionales y no funcionales del cliente.
- **Diseño preliminar:** se realiza un diseño de alto nivel de la solución a los requerimientos analizados en la fase anterior.
- **Diseño detallado:** en metodologías agiles se adiciona la creación de un pseudocódigo.
- **Codificación y pruebas:** se realiza la codificación y las pruebas correspondientes a la solución con base al diseño realizado anteriormente.
- **Mantenimiento:** se realizan las correcciones a los defectos encontrados en las pruebas realizadas y a los defectos encontrados mediante las pruebas del cliente.

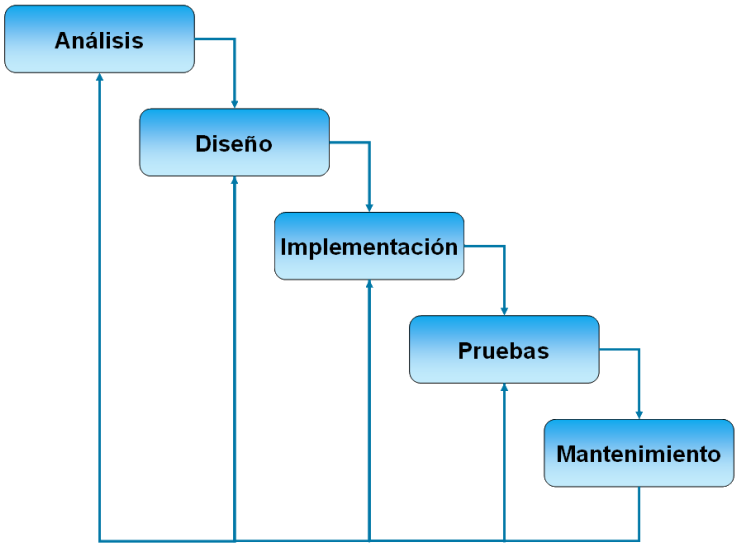

Figura 1: Modelo del ciclo de vida en cascada. Tomada de http://metodologiaencascada.blogspot.com/

6.5. Modelo de pruebas en "V".

El modelo de pruebas en "V" es muy utilizado en la actualidad, se considera como una extensión del modelo de desarrollo en cascada presentando una relación entre cada fase del ciclo de desarrollo y su correspondiente fase del ciclo de pruebas (Hilburn & Towhidnejad, 2000) (Mathur & Malik, 2010). La imagen 1 muestra como es el modelo de pruebas en V con sus correspondientes fases.

El modelo inicial contemplaba los siguientes niveles (Nguyen, Perini, & Tonella, 2007):

- **Pruebas unitarias:** son el nivel más pequeño de pruebas a realizar, es la encarga de realizar las pruebas sobre una funcionalidad específica con respecto a sus objetivos.

- **Pruebas de integración:** es el siguiente nivel, encargado de realizar las pruebas al conjunto integrado de funcionalidades validadas con las pruebas unitarias. Su objetivo es verificar que una funcionalidad trabaje correctamente con las otras funcionalidades integradas anteriormente. Comúnmente se trabaja con objetos falsos que simulan una funcionalidad que aún no se ha terminado de desarrollar.

- **Pruebas de sistema:** las pruebas de sistema se encargan de validar el funcionamiento correcto de todas las funcionalidades del sistema después de ser integradas en búsqueda de problemas de funcionamiento conjunto de los objetos desarrollados, problemas de seguridad, abrazos mortales entre componentes, etc.

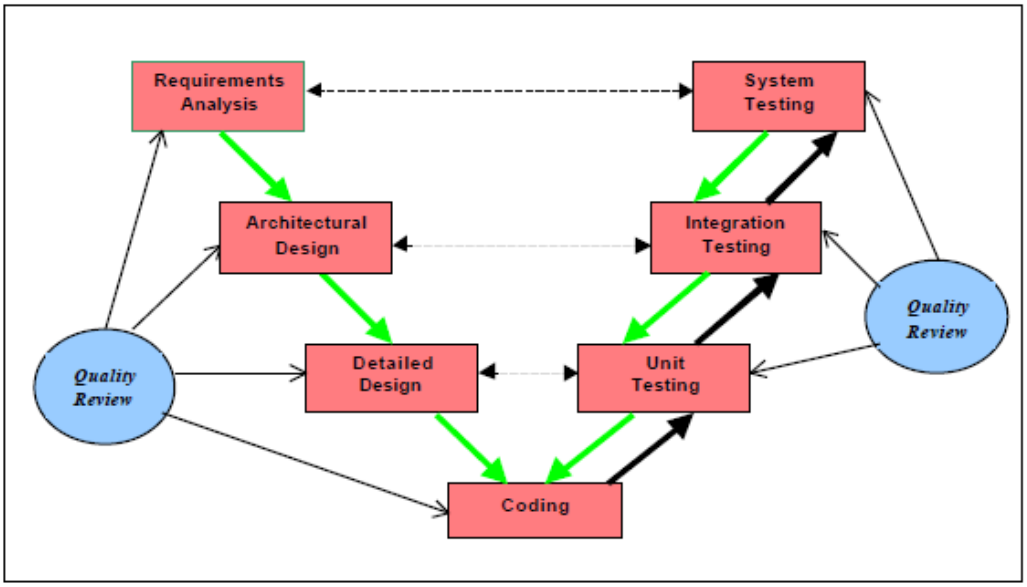

Figura 2: Modelo de pruebas en V. Tomado de Software Quality: A Curriculum Postscript? (Hilburn & Towhidnejad, 2000).

Existen diversas modificaciones al modelo de pruebas en V como el modelo en V múltiple el cual incluye un ciclo de desarrollo en V para cada uno de los artefactos presentados en la aplicación (modelos, prototipos y producto final) incluyendo el diseño, la construcción y las pruebas correspondientes (Notenboom). La esencia de este modelo es que determinadas propiedades del sistema no se podrán probar en el ciclo normal, si no que se podrán probar en los prototipos o incluso en ciertas condiciones se podrán probar en el producto completamente desarrollado y terminado.

Otra propuesta es el modelo avanzado en V que adiciona las actividades de mantenimiento a las de desarrollo y de pruebas con el fin de conseguir una mayor eficiencia y confiabilidad del sistema. En la figura 2 se muestra el modelo avanzado en V.

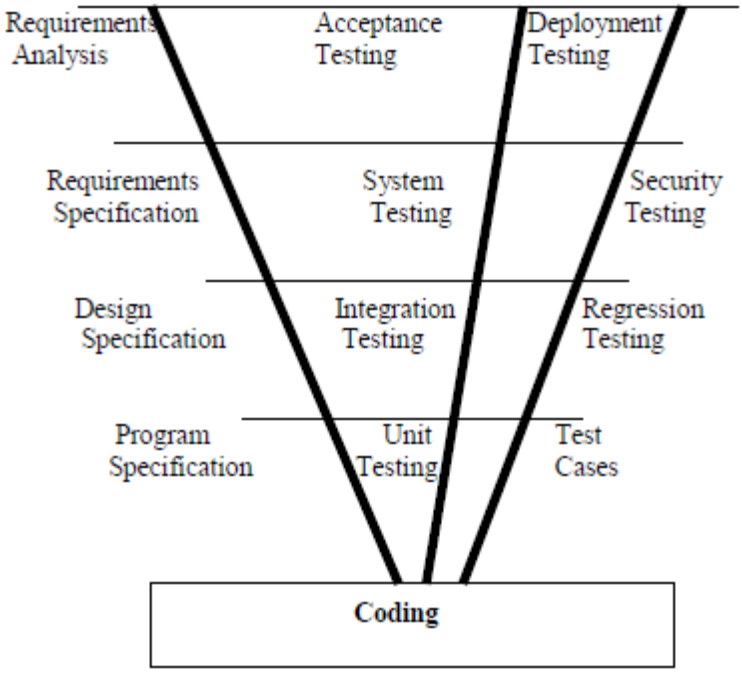

Figura 3: Modelo en V avanzado. Tomado de Advancements in the V – Model (Mathur & Malik, 2010).

Este modelo propone realizar pruebas sobre determinadas actividades que normalmente no se tienen en cuenta como los despliegues en el ambiente de producción del cliente, detalles de seguridad como los accesos no autorizados, o pruebas de regresión con el objetivo de encontrar posibles inyecciones de defectos después de adicionar nuevas funcionalidades a la aplicación.

Indudablemente esta técnica es de gran importancia para obtener un producto de calidad, permitiendo que cada una de las etapas del ciclo de desarrollo tenga sus correspondientes pruebas que garanticen la calidad de los entregables y de la aplicación en general. La aplicación concienzuda de cada una de las etapas de pruebas es vital para la calidad del producto que se está desarrollando.

6.6. Inspecciones de Código.

Las inspecciones de código es un proceso en el cual un grupo de personas sigue un proceso formal de revisión de artefactos con el fin de encontrar defectos en etapas tempranas disminuyendo los costos de corrección en etapas más tardías del proyecto (Casallas, 2009).

Fue introducido por Michael Fagan a mediados de los años 70 en IBM con el objetivo de revisar lógica en hardware y posteriormente se utilizó para revisar diseños, código, planes de pruebas y documentación. El proceso consta de un equipo con roles bien definidos, familiarizados con el artefacto a inspeccionar y con el proceso en sí de inspección, se reúnen y discuten con el fin de identificar defectos en medio de la etapa correspondiente del artefacto. A continuación, la lista de defectos se entrega al autor del artefacto para que este haga las correcciones pertinentes (Aurum, Petersson, & Wohlina, 2002) (Gately, 1999).

Los objetivos de las inspecciones son los siguientes (Zamiriano, 2007):

- Encontrar tempranamente los defectos de un artefacto en construcción.
- Prevenir el malfuncionamiento de los procesos o planes establecidos.
- Proporcionar mejoras en el software.
- Aportar con el mejoramiento continuo del proceso de desarrollo.
- Aportar con el conocimiento compartido entre los programadores o ingenieros involucrados recién llegados al proyecto.

Para que la inspección sea más exitosa el autor del artefacto hace una revisión al terminar el desarrollo del mismo, con el fin de encontrar defectos “obvios” y que no lleguen al proceso de inspección (Humphrey, 2000).

El modelo de inspecciones más conocido es el propuesto por Fagan, el cual consta de las siguientes etapas (Porter, Siy, & Votta, A Review of Software Inspections, 1995) (Fagan, 1976):

- **Planeación:** en esta etapa el artefacto a ser inspeccionado se chequea con el fin de verificar que cumpla con ciertos criterios. Si los cumple, se conforma el equipo de inspección comúnmente conformado por programadores que han trabajado en artefactos similares partiendo del hecho que esta familiaridad con artefactos del mismo tipo hará que la inspección sea más efectiva.

- **Revisión:** el autor del artefacto se reúne con el equipo de inspección y da un resumen del propósito de su creación incluyendo las relaciones con otros artefactos.

- **Preparación:** cada miembro del equipo de inspección analiza independientemente el artefacto y la documentación relacionada a este, en busca de defectos potenciales para registrarlos.

- **Inspección:** el equipo de inspección se reúne y analiza el artefacto para encontrar defectos partiendo del hecho que el trabajo en equipo podrá encontrar defectos que individualmente son más difíciles de encontrar.

  Después de realizar la revisión en equipo, se nombran los roles como el moderador encargado de encaminar la reunión y el lector encargado de leer el contenido del artefacto al equipo (Fagan, 1976). A medida que el lector lee el artefacto, se van descubriendo los defectos preguntando si han sido descubiertos por otros miembros del equipo. Estos defectos se anotan para que el autor tenga el registro y proceda a realizar las correcciones. Se debe procurar que las reuniones de inspección no duren más de 2 horas.

- **Correcciones:** el autor realiza las correcciones de los defectos encontrados en la reunión de inspección.

- **Seguimiento:** la corrección de cada uno de los defectos reportados es verificada por el moderador. El moderador tiene la potestad de solicitar una re inspección del artefacto dependiendo de la cantidad y/o calidad de las modificaciones realizadas por el autor.

Dentro de las métricas de las inspecciones de código (Suma & Gopalakrishnan, 2009), las dos a tener en cuenta son las siguientes:

$$Número\ total\ de\ Defectos = A + B - C$$

***Formula2*: *número total de defectos*.**

$$Número\ total\ de\ defectos\ Estimados = \frac{A * B}{C}$$

Formula 3*: *Número total de defectos estimado

Donde ***A*** y ***B*** son el número total de defectos encontrados por el inspector 1 y 2, y ***C*** es el número total de defectos encontrados en común por los inspectores.

El modelo de Fagan indica que el número perfecto de participantes en un proceso de inspección debe ser cuatro (4). Adicional a este modelo, existen otros modelos de inspecciones de software en donde el número de participantes se ve afectado (Aurum, Petersson, & Wohlina, 2002):

- **Modelo de revisión activa de diseño:** propone múltiples revisiones pequeñas enfocadas a diversas partes del artefacto (Parnas & Weiss, 1985). Para diseños de alto nivel se ejecutan técnicas avanzadas de revisión y de lectura buscando determinadas características de calidad en el documento. Estas técnicas de lectura pueden ser enfocadas a la búsqueda de defectos en diagramas especializados, en el lenguaje natural que acompaña los modelos en las interfaces de usuario adicionales (Travassos, Shull, Fredericks, & Basili, 1999).

- **La inspección en parejas:** propone eliminar el rol de moderador lo cual tiene beneficios inmediatos en la calidad y productividad del programador (Bisant & Lyle, 1989).

- **Modelo de Inspección por N equipos:** propone ejecutar la inspección por varios equipos conformados por varios revisores, el autor y son coordinados por un único moderador con el fin de recolectar toda la información producida por los equipos. Esto con la premisa que muchos equipos encontrarán más defectos que un único equipo (Martin & Tsai, 1990). Experimentalmente se mostró que el éxito de este modelo radica en el nivel de experiencia de los revisores y el número de equipos (Knight & Myers, 1993).

- **Inspección por fases:** este modelo combina ciertos aspectos de los anteriores modelos y plantea realizar las inspecciones en varias fases con un orden secuencial impidiendo que se ejecute la siguiente fase sin terminar las correcciones propuestas por la fase anterior (Knight & Myers, 1993).

La importancia de este método dentro en la calidad de un producto radica en el alto costo de corregir defectos en fases de desarrollo diferentes a la del artefacto, el costo de corregir un defecto en la especificación de un requerimiento en la fase de pruebas es mucho más alto que corregirlo en la misma fase de requerimientos. Este método ayuda a la construcción de la calidad del producto durante el proceso de desarrollo y con esto, el trabajo en las últimas etapas del proyecto disminuye mejorando los tiempos en los cronogramas del proyecto y evitando los atrasos fatales (Casallas, 2009).

Dentro de los principios para realizar las inspecciones de código se encuentran los siguientes (Wiegers, 1995):

- **Dejar los egos afuera:** para un desarrollador no es fácil exponer el producto que ha desarrollado con mucho cuidado a un grupo de inspectores que están empeñados en encontrar todos los errores posibles. Esto causa que el desarrollador se ponga a la defensiva y se produzcan discusiones acaloradas en las reuniones de inspección. Un ambiente cordial y tranquilo es el más recomendable para realizar esta tarea, dado que es mejor que los inspectores encuentren un defecto que lo encuentre el cliente.

- **Criticar los productos no a los productores:** el propósito de la inspección no es comparar la inteligencia del inspector contra la del desarrollador, si no generar un producto con alto estándar de calidad. El moderador debe estar atento a estos inconvenientes y actuar inmediatamente.

- **Encontrar los defectos durante la revisión:** el autor se debe encargar de resolverlos después de la reunión de inspección, no durante la misma dado que se puede entrar en un círculo vicioso tratando de encontrar la mejor solución al problema.

- **Limitar la reunión de inspección a máximo dos horas:** en reuniones muy largas, la atención de los participantes disminuye bastante y con esto disminuye la efectividad encontrando defectos. En caso que no se cubra todo el código en las dos horas, se debe agendar otra reunión para terminar la revisión. Después de esta reunión se procede a realizar las correcciones correspondientes.

En otro experimento realizado por Porter, Siy, Toman y Votta en AT&T sobre una aplicación con aproximadamente 53.000 líneas de código de las cuales 8.000 fueron reutilizadas de los prototipos implementados, implementando combinaciones sobre el tamaño del equipo y el número de reuniones de inspecciones demostraron (Porter, Siy, Toman, & Votta, 1997):

- El modelo de 2 sesiones con 2 personas es un 33% más efectivo que el de 1 sesión con 4 personas.

- Las revisiones realizadas por unas sola personas es menos efectiva que realizadas por 2 o 4 personas, siendo el equipo con una reducción significativa de esfuerzo el conformado por 2 personas.

- Varias reuniones de inspección con 2 personas utilizando un método especial de detección de defectos es más efectivo que varias reuniones con un solo grupo grande sin utilizar una técnica especial de detección.

- Realizar las correcciones de defectos no tienen un efecto significativo en la tasa de detección de defectos.

El proceso de inspecciones también se aplicó en la NASA, en donde los resultados no fueron los deseados: el efecto en la detección de defectos fue muy pequeño. En los análisis realizados se tuvo la hipótesis que los inspectores requerían técnicas especializadas en la detección de defectos. Como respuesta a esta hipótesis, se formuló el proceso “Cleanroom” en donde se les daba a los desarrolladores técnicas especializadas y motivación en lectura. Esto dio como resultado un aumento en el 25% en la tasa de descubrimiento de defectos y en un

30% de la productividad (Ciolkowski, Laitenberger, Rombach, Shull, & Perry, 2002). Mientras que en otro experimento realizado por Gately, el índice de eficiencia de remoción de defectos de esta técnica está entre el 80% y 85% (Gately, 1999).

6.7. Desarrollo Dirigido por Pruebas.

Muchos desarrolladores piensan que realizar pruebas unitarias sobre código fuente es un proceso muy aburrido que genera una pérdida de tiempo. Sin embargo la importancia de estas pruebas radica en encontrar diversos defectos en el código antes de pasar por la etapa de pruebas del sistema realizadas por el equipo de calidad o incluso antes de las pruebas de aceptación realizadas por parte del cliente.

El desarrollo dirigido por pruebas (TDD por sus siglas en inglés) es una técnica que lleva aplicándose desde hace varios años exitosamente en lugares como Estados Unidos y el Reino Unido. Para Carlos Blé es una técnica de diseño e implementación de software incluida dentro de la metodología XP y conocido dentro de esta como Diseño Dirigido por Ejemplos y que responde a preguntas fundamentales en el diseño como: ¿Cómo lo hago? ¿Por dónde empiezo? ¿Cómo sé qué es lo que hay que implementar y lo que no? ¿Cómo escribir un código que se pueda modificar sin romper funcionalidad existente? Lo cual convierte a esta técnica en una herramienta poderosa de diseño de software en donde se debe pensar en ejemplos que eliminen la ambigüedad del lenguaje natural de los casos de uso y en donde la propia implementación de los artefactos va definiendo la arquitectura de la aplicación (Blé, Beas, Gutiérrez, Reyes, & Mena, 2010).

La técnica se puede describir de la siguiente manera: primero se debe escribir la prueba unitaria de un requisito y hacer que esta falle retornando un valor contrario al que se espera, después se debe escribir la cantidad mínima necesaria de código para que la prueba pase, a continuación se debe ordenar el código y se repite el ciclo tantas veces como casos de prueba se tenga. Lo ideal es que no se escriba código de la aplicación sin tener la prueba completamente codificada y que esta falle la primera vez que se ejecuta, si ésta se ejecuta exitosamente se debe revisar la codificación de la prueba unitaria ya que probablemente esté mal implementada y no serviría de nada.

Para aplicar esta técnica se tienen varios requisitos como el entendimiento y comprensión del problema antes de comenzar a desarrollar las pruebas unitarias y el código de la aplicación; la aplicación que se está desarrollando tiene que ser lo suficientemente flexible como para permitir que sea probada automáticamente; cada prueba debe ser lo suficientemente pequeña como para que permita determinar si el código probado aprueba o no la verificación que está le impone.

Capers Jones afirma en su libro (Jonnes, Software Engineering Best Practices: Lessons from Successful Projects in Top Companies, 2010) que la eficiencia de TDD para eliminar defectos en el código fuente es superior a otras técnicas como codificar primero y después probar llegando hasta un 85% y combinando esta técnica con otras como son la inspección de código y de casos de prueba se puede alcanzar el 95% de eficiencia sin olvidar el 7% correspondiente a la inyección accidental de errores producida por la corrección de otros errores dejando un cifra del 88% de eficiencias que en mi concepto es suficiente para garantizar un producto de muy buena calidad lo cual implica la satisfacción del usuario que utilizará la aplicación que se ha construido.

Dentro de las ventajas de esta técnica (Informática, 2012) encontramos: el aumento en la calidad de la aplicación que se está desarrollando dado que se ejecutan pruebas unitarias que garantizan que la aplicación no falle la primera vez que se ejecuta adicionalmente que la cantidad de defectos que se encuentran en la etapa de pruebas del sistema es menor gracias a la detección temprana por las pruebas

El diseño de la aplicación se enfoca en las necesidades del usuario haciendo que las necesidades principales del usuario sean solventadas en primer lugar haciendo que el trabajo posterior a la entrega sea menor. Adicionalmente, el diseño de la aplicación es mucho más simple por el enfoque dado a las necesidades principales del usuario, disminuyendo la complejidad, las clases multipropósitos o el código demasiado acoplado.

Otra ventaja es el aumento de la productividad del equipo, aunque inicialmente no se ve de esa manera dado a la cantidad de casos de prueba a escribir y las constantes modificaciones y refactorizaciones sobre el código, en las etapas finales del proyecto no se presentaran grandes cantidades de errores (gracias a la continua revisión del código por la pruebas) siendo un factor motivante para el equipo haciendo que se aumente la producción de soluciones a los errores presentados.

Una de las más grandes desventajas de esta técnica es la dificultad para realizar las pruebas unitarias de ciertos fragmentos de código dependientes de componentes externos o clases que están siendo desarrolladas en paralelo. Para esto se deben construir objetos de simulación para suplir la funcionalidad en cuestión, sin embargo, cuando se trata de componentes de la aplicación desarrollados en paralelo no es un tema complicado, pero en los casos de componentes externos como bases de datos, sistemas de correo electrónico, sistemas de colas, etc., la complejidad de realizar los objetos de simulación es muy alta lo que aumenta el tiempo de desarrollo del componente.

En un experimento realizado por Janzen y Saiedian (Janzen & Saiedian, 2008) se reunieron varios grupos de desarrolladores con características técnicas similares, en donde uno de los grupos implementará una aplicación siguiendo el flujo tradicional de desarrollo (diseño de alto nivel, diseño detallado, implementación, pruebas unitarias y pruebas de sistema) y el otro grupo lo hará con un flujo de desarrollo que incluye TDD (diseño de alto nivel, prueba unitaria, codificación, refactorización y pruebas de sistema).

Dentro de los resultados se encontró que los grupos que trabajaron en el flujo que incluye TDD, realizaron módulos y métodos más pequeños en líneas de código que la contraparte; también se identificó que el grupo que implementó TDD escribió una menor cantidad de métodos por clase, con lo que se concluye que los programadores que utilizan TDD tienden a escribir clases y métodos más pequeños y simples los cuales son mucho más fáciles de comprender.

La técnica del desarrollo dirigido por pruebas es una herramienta muy poderosa para garantizar la calidad de una aplicación, sobre todo nos brinda la posibilidad de reducir la cantidad de defectos en las etapas finales de un proyecto.

7. Metodología.

Para la realización de este experimento, se seleccionaron 10 estudiantes de pregrado con conocimientos en programación, a los cuales se les solicitó diligenciar un formato de hoja de vida con el objetivo de indagar sobre sus conocimientos en programación y experiencia, tanto a nivel académico como a nivel profesional. Según las respuestas de los estudiantes, se realizó la organización de los grupos de trabajo, dividiéndolos en tres grupos: un grupo de cuatro estudiantes y dos grupos de tres estudiantes. El formato de hoja de vida se puede ver en los anexos de este documento.

Después de la organización de los grupos se entregó la instrucción de configuración del ambiente de trabajo en cada una de las máquinas de los estudiantes. Dicha configuración constaba de la instalación de la versión 7.0_51 de la máquina virtual de Java y la versión 4.3 del IDE Eclipse (Kepler).

La condición más importante para poder participar en el experimento era cumplir con el horario establecido con anterioridad y participar en la mayoría de sesiones, para ello se tomó la asistencia en cada una de las sesiones programadas.

Los estudiantes se reunieron los días jueves de 8:20 p.m. a 9:50 p.m. y los días sábados de 1:40 p.m. a 4:50 p.m. para resolver los siguientes retos propuestos:

- **Reto1:** indicar la cantidad total y de cada una de las palabras reservadas definidas en un archivo de texto dentro de un párrafo o código fuente que se encuentra en otro archivo de texto, teniendo en cuenta se debe tener al menos una palabra reservada y que estas no pueden contener números ni espacios en blanco, y que se ingresa un párrafo o código fuente.

- **Reto2:** teniendo un archivo de texto con n líneas, en el cual cada línea se compone de las longitudes de los tres lados de un triángulo separados por coma, se debe indicar cuál es el triángulo más grande y el más pequeño según su área y perímetro. Se debe tener en cuenta que los valores de las longitudes de los triángulos son números positivos no nulos.

- **Reto 3:** teniendo un archivo de texto en donde cada línea está separada por comas y corresponde al nombre de un producto, su cantidad en inventario, la cantidad máxima a almacenar en el inventario y el precio unitario de venta; el sistema debe indicar que producto debo pedir en caso que su existencia en el inventario sea menor o igual al 20% de la capacidad máxima del inventario y que producto debo vender a mitad de precio si su existencia en el inventario es mayor o igual al 80% de su capacidad máxima de inventario.

- **Reto 4:** teniendo dos archivos de texto, donde en uno hay placas correctas de vehículos colombianos separadas por coma y que corresponden a los vehículos

que tienen convenio con un parqueadero y en el segundo archivo se encuentran las placas de vehículos, la fecha y hora de entrada y salida del vehículo del parqueadero. El sistema debe indicar para cada uno de los vehículos el monto a pagar por concepto de parqueo teniendo en cuenta que el valor del minuto es de 60 pesos, la tarifa plena es de 10.000 pesos si se aplica por más de cuatro horas de parqueo y que los vehículos que tienen convenio son acreedores a un descuento del 40% sobre el valor total.

- **Reto 5:** teniendo un archivo de texto donde se encuentran, separados por coma, el nombre de un producto, la cantidad de pedidos y el valor unitario del producto. El sistema debe generar la factura por concepto de venta de los productos en un archivo de texto, incluyendo el subtotal, el valor de IVA y la propina sobre el subtotal y el total de la venta.

- **Reto 6:** teniendo un archivo de texto correspondiente a una cuenta bancaria, con la fecha y hora de un movimiento, el tipo de movimiento (1 es retiro, 2 es consignación y 3 es pago en línea) y el valor de la transacción, el sistema debe generar el consolidado de movimientos por mes indicando la cantidad de movimientos y el valor total de los movimientos para cada uno de los tipos.

- **Reto 7:** teniendo dos archivos de texto en donde uno contiene los nombres, separados por coma, de los equipos participantes en un torneo de futbol y en otro los resultados de los partidos del torneo; el sistema debe generar en un archivo de texto la tabla de posiciones ordenándola por puntos, goles de diferencia, goles a favor de visitante y orden alfabético.

A cada uno de estos retos se les dio una calificación a la complejidad del problema, que tiene en cuenta los siguientes criterios: la complejidad de abstracción de la solución al problema que tiene un peso del 50%, la complejidad en las validaciones con un peso de 20%, el tipo y cantidad de entradas con peso de 10% y el tipo de salida con un 20%.

Teniendo en cuenta los pesos dados a cada criterio, se asigna a cada uno un valor entre uno y cinco y con esto se calcula el ponderado (la suma del producto de la nota dada a daca criterio por su peso) para cada uno de los retos, dando como resultado la complejidad del problema, estos valores son los siguientes:

	Complejidad Abstracción (50%)	Complejidad Validaciones (20%)	Complejidad de Entradas (10%)	Complejidad de Salida (20%)	Complejidad del Problema
Reto 1	1	2	4	2	1,7
Reto 2	2	2	2	2	2
Reto 3	3	4	2	3	3,1
Reto 4	3	4	4	3	3,3
Reto 5	3	4	2	4	3,3
Reto 6	4	5	4	3	4
Reto 7	5	4	5	5	4,8

Tabla 2: ponderación de la complejidad de los retos propuestos en el experimento.

Los estudiantes contaron con alrededor de 3 o 4 sesiones de trabajo, tiempo en el cual debían presentar la solución al reto dado en forma de archivo binario ejecutable, con el fin de someterla a diversas pruebas de aceptación. En caso de encontrar errores en la versión entregada, los estudiantes deben realizar las correcciones correspondientes hasta completar el reto acorde a la especificación dada y que satisfaga todas las pruebas de aceptación. La metodología que los estudiantes siguieron para hacer el desarrollo de la solución es el desarrollo en cascada.

Para el desarrollo de los dos primeros retos no se les impone a los equipos la forma de trabajar, de manera que cada uno de ellos afrontaba el problema de diferentes maneras y comenzaba a trabajar en el problema.

Después de la terminación del primer reto, uno de los estudiantes debe salir del experimento debido a otros compromisos con la universidad, por lo tanto se realiza una reorganización de los equipos de trabajo y se entregan las especificaciones del segundo reto, el cual los estudiantes no terminaron en el tiempo dado.

Dado el bajo rendimiento de los equipos en los dos primeros retos y teniendo en cuenta que para el segundo reto no se entregó ninguna versión, se solicita al profesor de la práctica experimental hablar con los estudiantes para descubrir las falencias y proceder a corregirlas. El profesor toma la decisión de motivar a los estudiantes asignándole a cada uno la nota de 3 sobre 5 en el corte después de hablar con cada uno de ellos.

A partir del tercer reto se les solicita a los equipos que apliquen 2 técnicas, uno de los equipos aplicaría las inspecciones de código y el otro equipo aplicaría el desarrollo dirigido por pruebas. Adicionalmente, ambos equipos realizarían el diseño de la solución de manera obligatoria y como prerrequisito para continuar con el desarrollo

de la solución, con diferencia que el equipo que aplica el desarrollo dirigido por pruebas debería adicionar los pseudocódigos de la solución.

Los estudiantes que aplicaron la técnica de inspecciones de código, utilizaron una lista de chequeo en los últimos dos retos. Esta lista de chequeo contiene defectos básicos y tiene como objetivo generar un consenso, sobre defectos comunes, entre los inspectores para tratar de descartar la inexperiencia en inspecciones. Esta lista de chequeo se puede ver en los anexos de este documento.

Para que los estudiantes aplicaran dichas técnicas, se realizaron clínicas por grupo en donde se mostraron los procedimientos a seguir para cada una de las técnicas en cuestión.

En el momento que la versión entregada es aprobada, cada equipo debe entregar los artefactos correspondientes a su técnica aplicada (resultado de inspecciones de código, diseño de alto nivel y detallado, pseudocódigo y código fuente) con el fin de ser analizado en el experimento.

Adicionalmente, en cada reto se tomaron datos como la cantidad de pruebas fallidas por versión, tiempos usados en cada una de las fases del ciclo de desarrollo, y el porcentaje de tiempo usado en cada fase.

Las motivaciones aplicadas a los estudiantes fueron las siguientes:

- No realizar trabajo fuera del horario de clases.
- Si el reto se termina antes de completarse el tiempo asignado, los estudiantes quedan en descanso hasta el inicio del siguiente reto.
- Se prestó asesoría técnica en determinados problemas de programación.
- Se concedieron pequeñas extensiones al tiempo de finalización de los retos para que los estudiantes pudiesen terminar la solución.

8. Resultados.

Los resultados arrojados por el experimento se resumen en los siguientes gráficos:

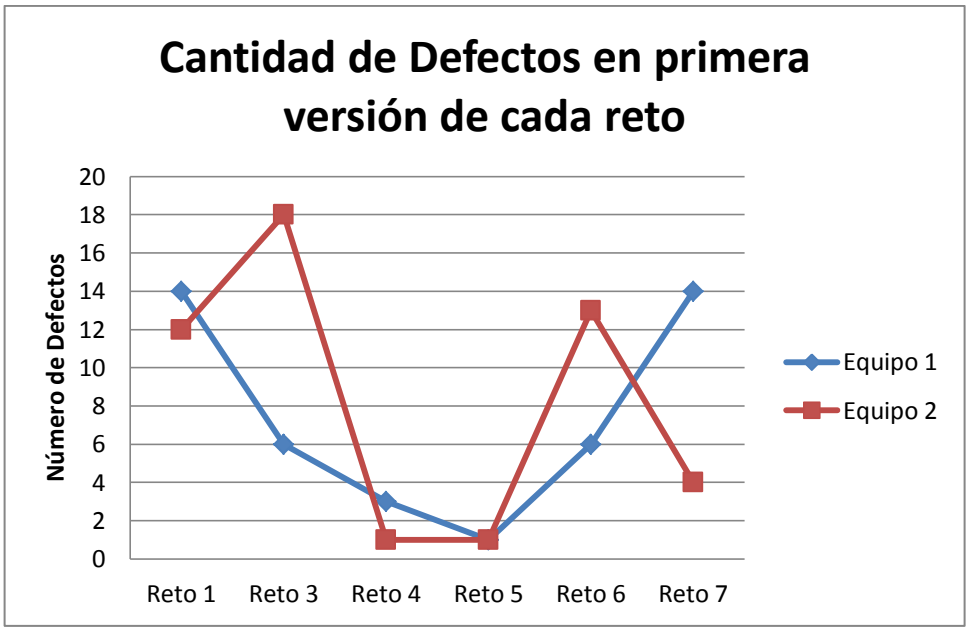

Gráfico 1: Cantidad de defectos en primera versión de cada reto por equipo.

La cantidad de defectos en la primera versión en los retos con complejidad entre 3 y 4, se redujo sustancialmente debido a la aplicación de las técnicas de desarrollo dirigido por pruebas e inspecciones de código y a la experiencia sumada en dichas técnicas con el transcurrir de los retos, sin embargo, en los retos con complejidad mayor a 4, la cantidad de defectos aumento debido a la poca experiencia demostrada por los integrantes de los equipos.

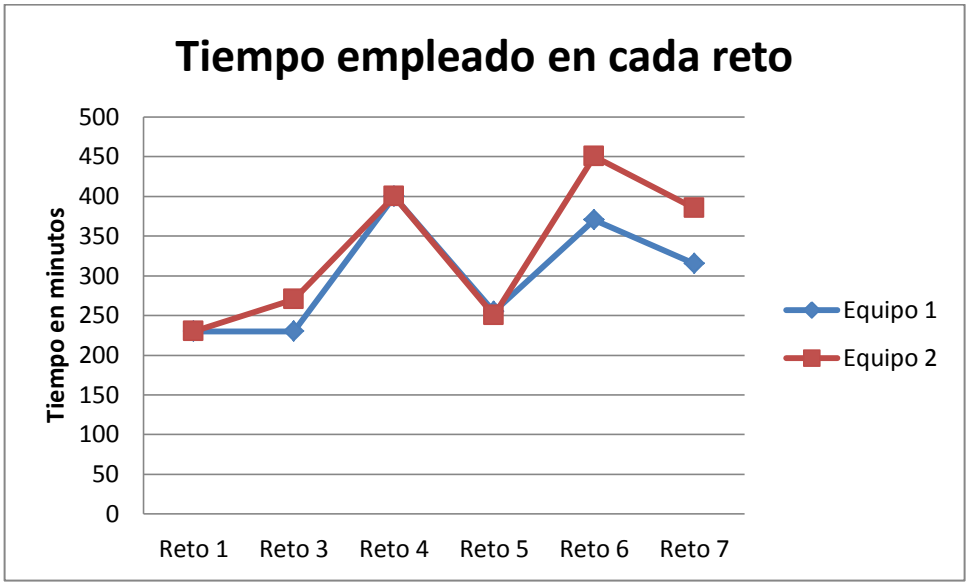

Gráfico 2: Tiempo empleado en cada reto por equipo.

Los tiempos empleados por reto son muy similares entre los dos equipos, también es necesario destacar que a varios grupos se les dio una extensión en el tiempo para la terminación en el reto, sobre todo para el equipo 2 en el último reto que es el que mayor complejidad tiene.

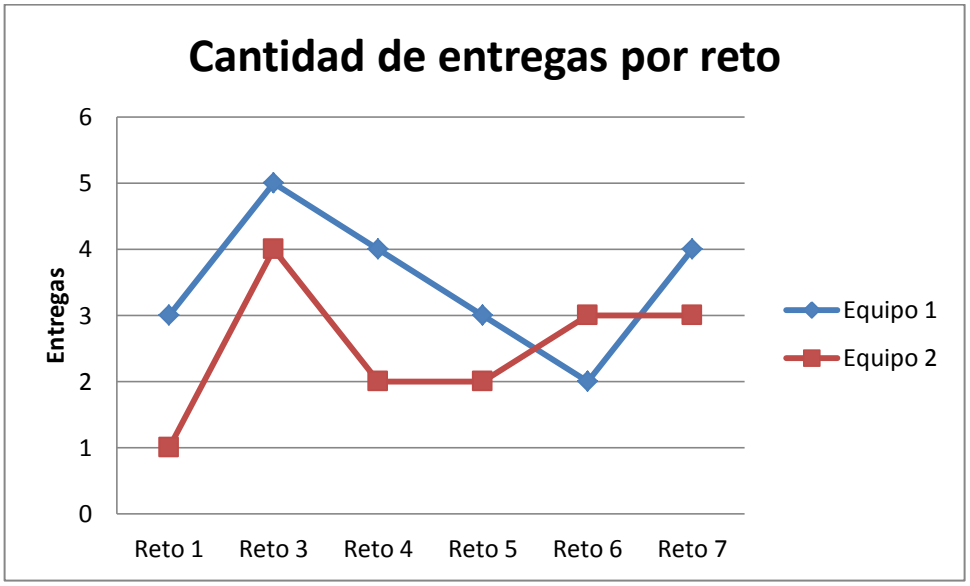

Gráfico 3: Cantidad de entregas por reto por equipo. Se debe tener en cuenta que el equipo 2 no finalizó los retos 1 y 3.

La cantidad de entregas por retos para el equipo 1, en el reto 3 pudo ser menor dado que se presentó inyecciones de defectos al realizar correcciones de manera apresurada sin realizar las correspondientes validaciones que asegurasen la no afectación de otras funcionalidades validadas correctamente, sin embargo, se evidencia una evolución positiva atribuible a la acción de la técnica de desarrollo dirigido por pruebas.

Con respecto al equipo 2, se evidencia un número de entregas constante, teniendo en cuenta que en los primeros retos no se llegó a una versión exitosa del reto.

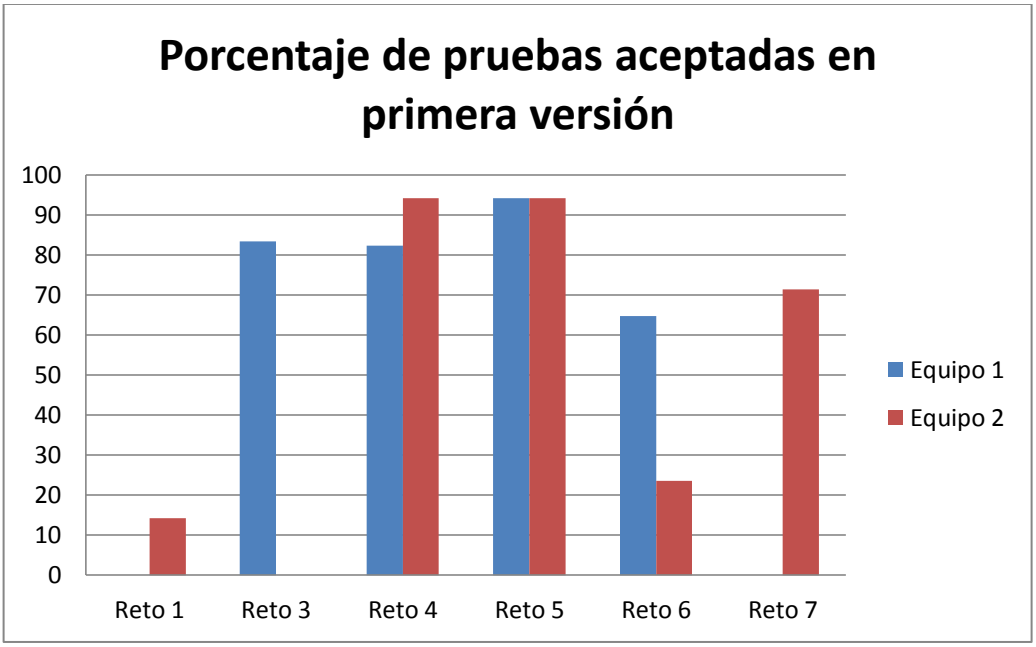

Gráfico 4: porcentaje de pruebas aceptadas en la primera versión para cada uno de los retos por equipo.

Durante la aplicación de las técnicas, se puede evidenciar un alto porcentaje de pruebas aceptadas en los retos de complejidad entre 3 y 4, sin embargo, en los retos de complejidad mayor a 4, el porcentaje disminuyo en un 30% aproximadamente mostrando la dificultad en encontrar la solución en problemas más complejos haciendo que se deba entregar más versiones para obtener una versión final exitosa.

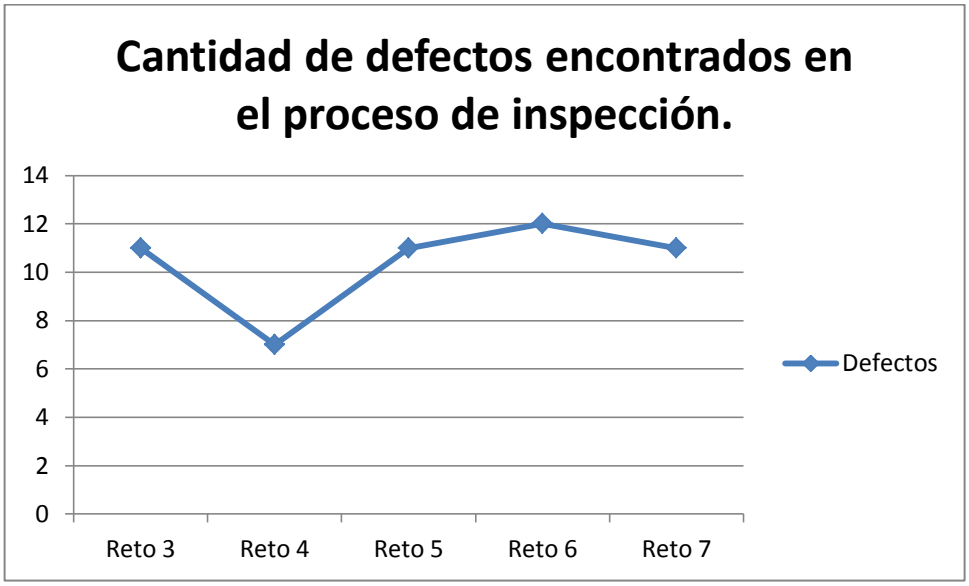

Gráfico 5: cantidad de defectos encontrados en el proceso de inspección de código.

La aplicación de la técnica de inspecciones de código en el reto 3 presento una gran cantidad de defectos encontrados, sin embargo ese reto no tuvo una versión final por parte del equipo debido al vencimiento del tiempo dado para este reto. Sin embargo en los siguientes retos se encontraron defectos que mejoraron la calidad del producto final con menos entregas. En los retos 6 y 7, se aplicó las lista de chequeo, la cual no produjo el resultado esperado posiblemente por el alto nivel de complejidad de los retos y la experiencia adquirida en la técnica con los retos anteriores.

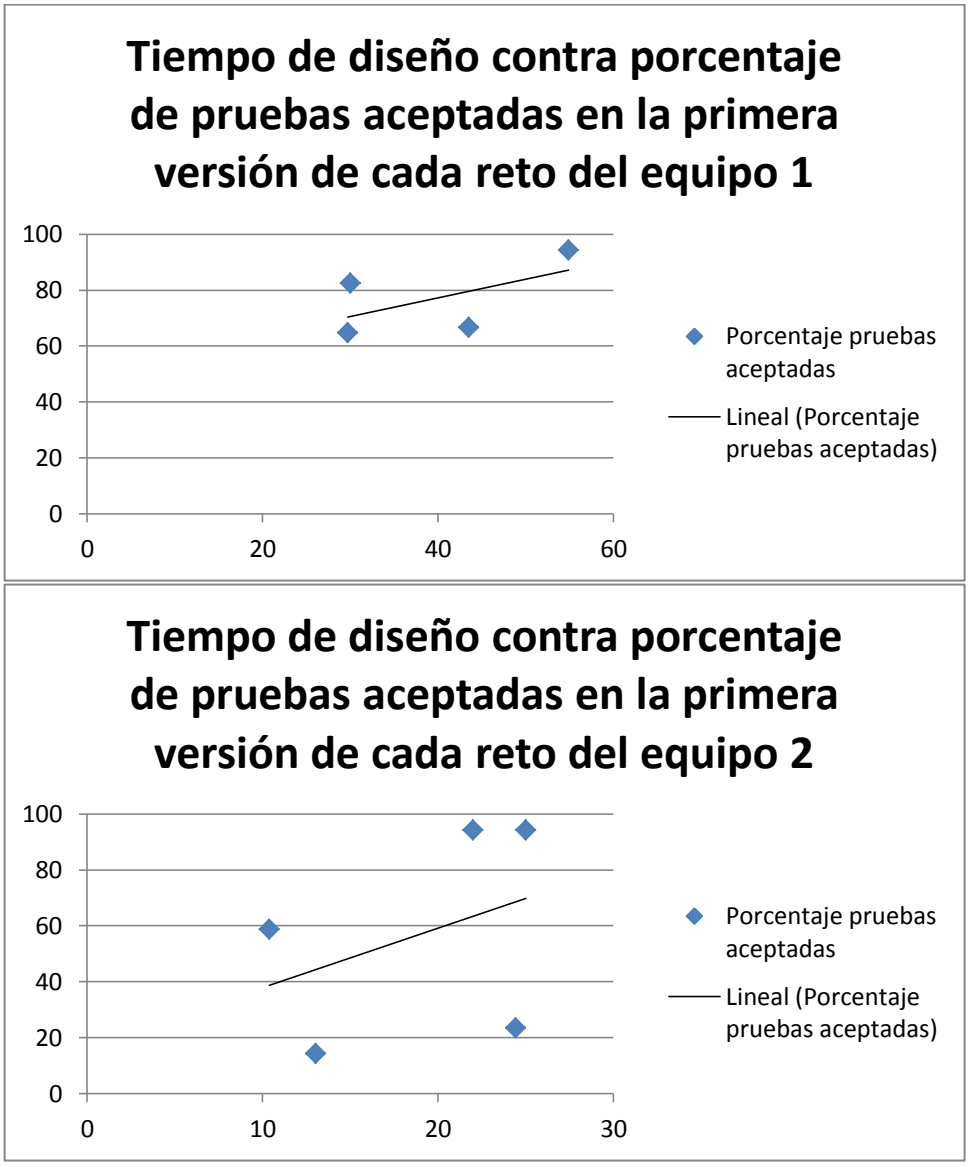

Gráfico 6: Tiempo de diseño normalizado vs porcentaje de pruebas aceptadas en la primera versión de cada reto.

Teniendo en cuenta el número de pruebas tomadas y la dispersión de las muestras, se evidencia una leve dependencia entre el tiempo dedicado al diseño con la calidad del producto, sin embargo este aumento de la calidad de puede atribuir a la aplicación de las técnicas.

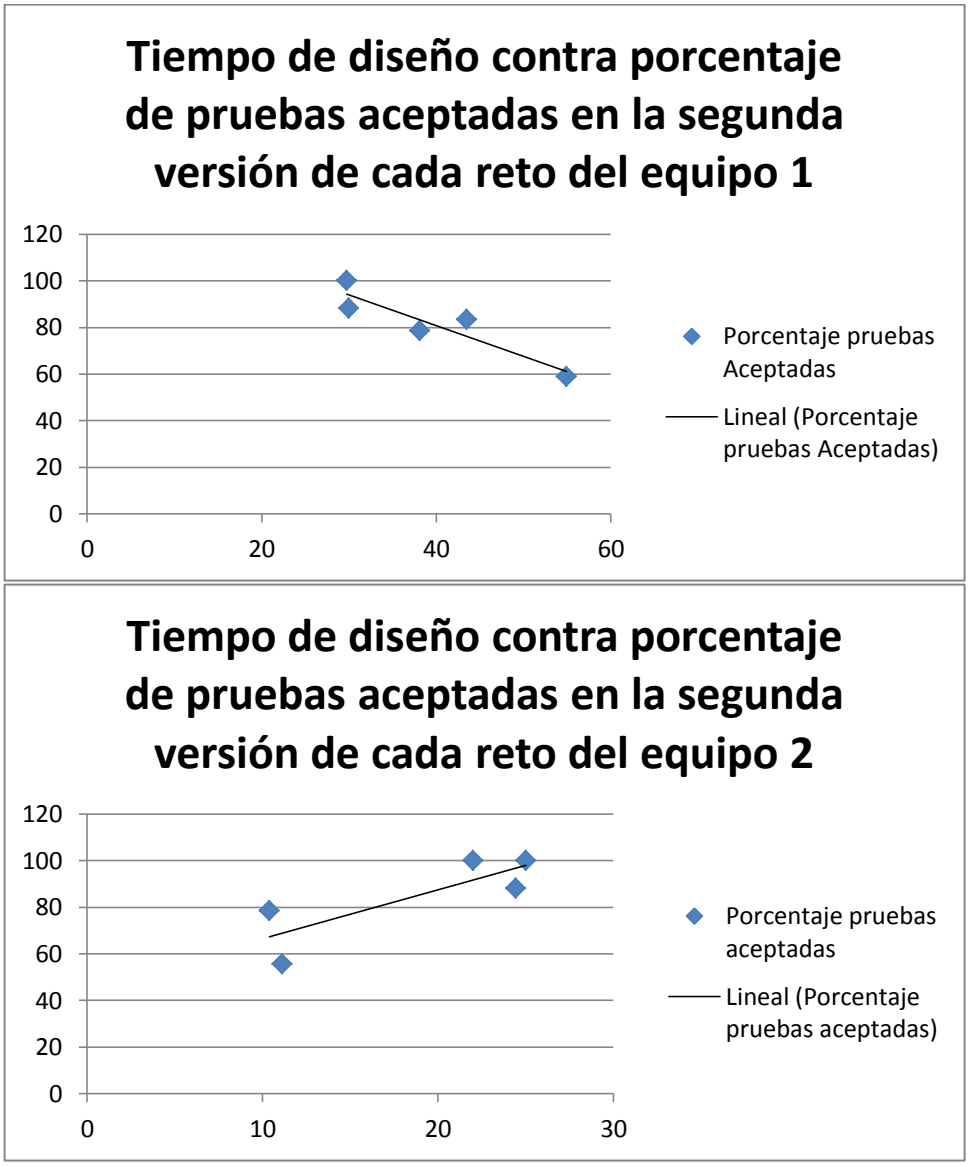

Gráfico 7: Tiempo de diseño normalizado vs porcentaje de pruebas aceptadas en la segunda versión de cada reto.

En la segunda versión del equipo 1, se evidencia que a mayor cantidad de tiempo empleado en el diseño, la calidad disminuye, sin embargo esto es atribuible a la inyección de defectos realizada por el equipo en la corrección de los errores encontrados en la primera versión, mientras que en el equipo 2 se reafirma que a mayor tiempo de diseño mejor es la calidad.

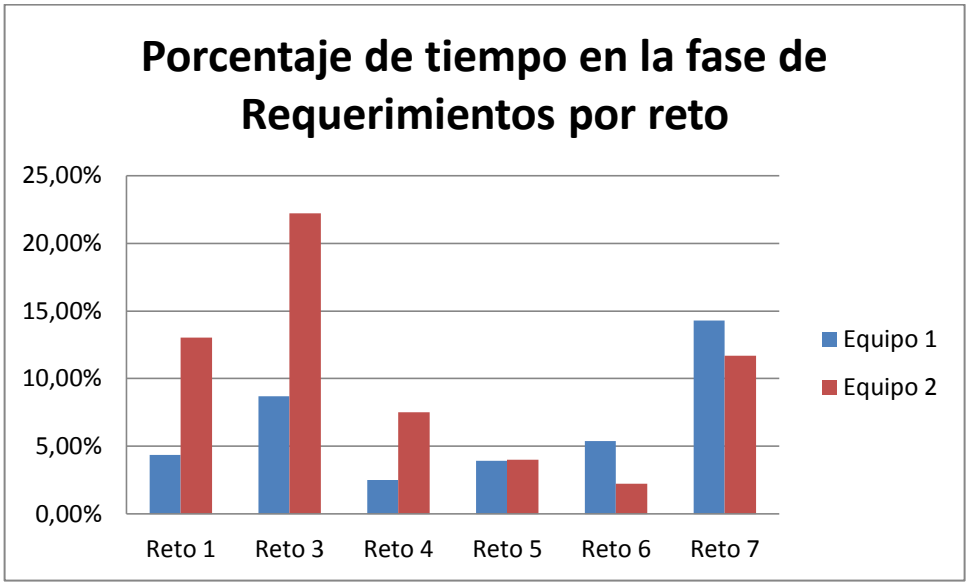

Gráfico 8: Porcentaje de tiempo utilizado en la fase de requerimientos por reto.

El tiempo empleado en la fase de requerimientos es muy bajo y a medida que se van desarrollando los retos, el tiempo es mucho menor, sin embargo a medida que se iban desarrollando los retos, se solucionaban dudas relacionadas los requerimientos del reto.

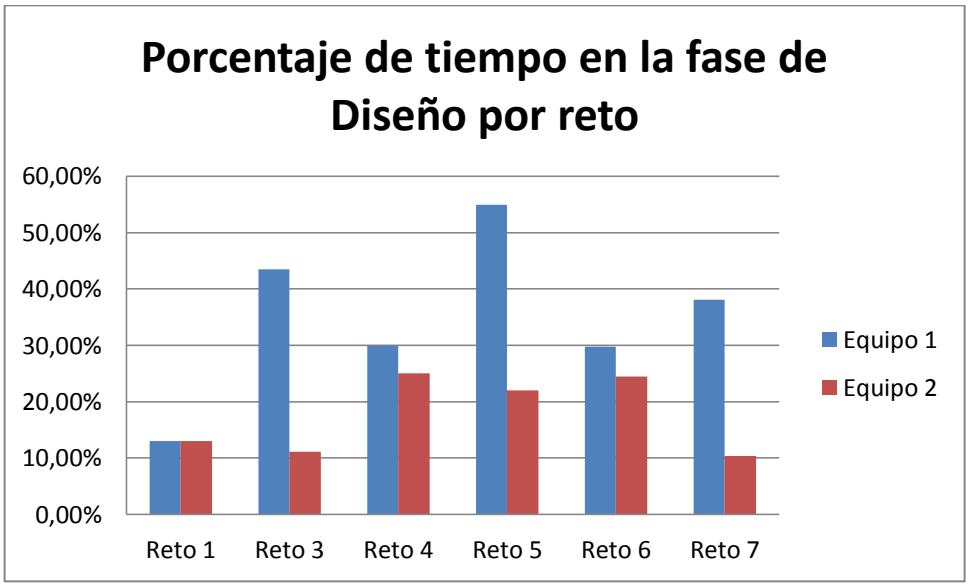

Gráfico 9: Porcentaje de tiempo utilizado en la fase de diseño por reto.

El tiempo utilizado en el diseño era muy bajo en los primeros retos, sin embargo al solicitarles a los equipos que lo aplicaran, este tiempo aumento considerablemente, sobretodo en el equipo que aplicó la técnica de desarrollo dirigido por pruebas junto con el diseño que incluía la realización de pseudocódigo.

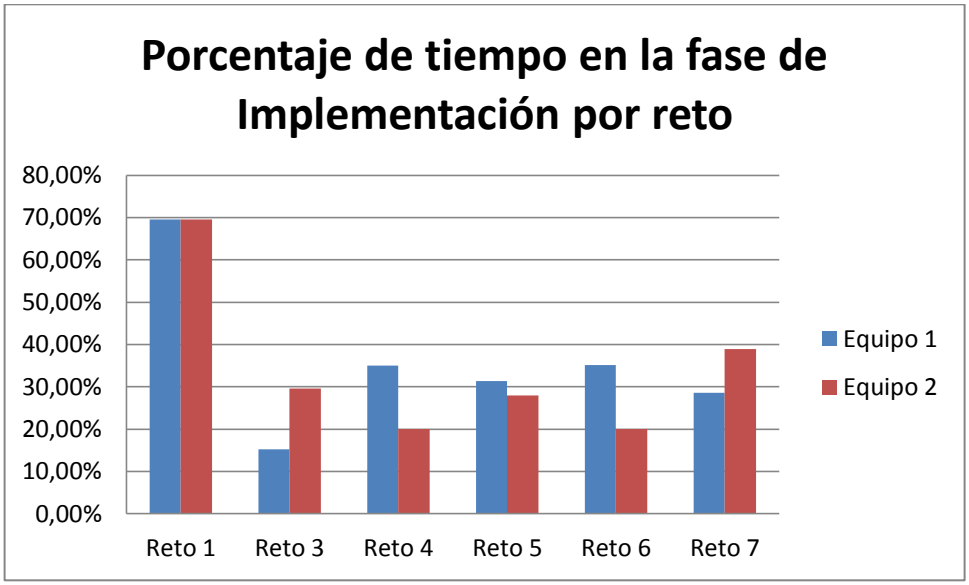

Gráfico 10: Porcentaje de tiempo utilizado en la fase de implementación por reto.

Después de la aplicación del diseño, el tiempo de la fase de implementación disminuyó considerablemente, mostrando que el diseño es el mejor punto de partida para obtener un producto de buena calidad.

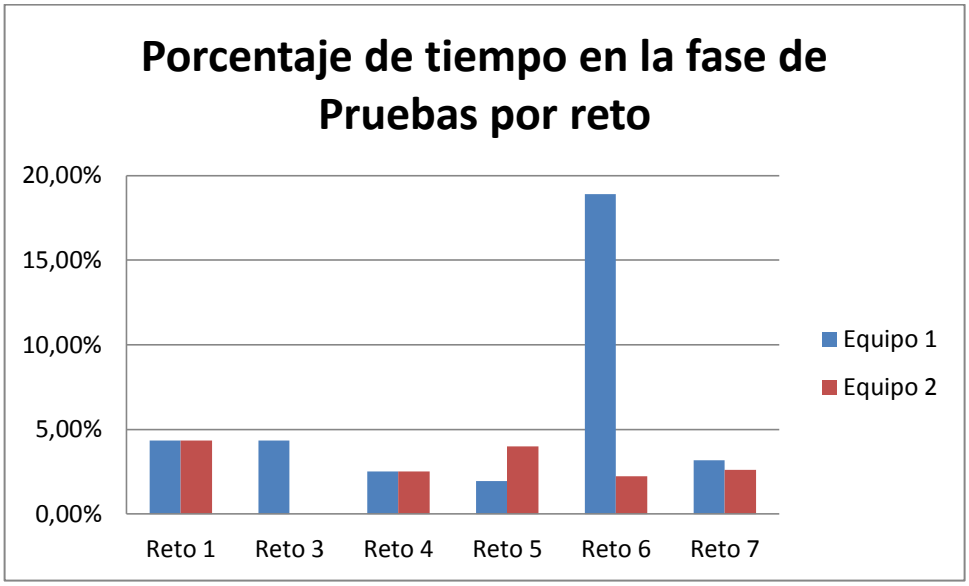

Gráfico 11: Porcentaje de tiempo utilizado en la fase de pruebas por reto.

Desafortunadamente los tiempos en la fase de pruebas son mínimos en comparación con las otras fases, eso nos muestra que los equipos no estaban probando de una manera apropiada. Esto es un factor determinante en la calidad del producto en la primera versión entregada dado que muchos de los defectos que se encontraron en la primera versión, se podían encontrar haciendo un set de pruebas previo a la entrega.

Teniendo que el equipo 1 estaba aplicando la técnica del desarrollo dirigido por pruebas, esto evidencia que la técnica no se estaba aplicando correctamente, aunque se realizaron pruebas unitarias, estas no contemplaban todos los casos, adicionalmente que por hacer las correcciones de manera rápida se obviaban las pruebas y con esto no se evidenciaba la inyección de defectos.

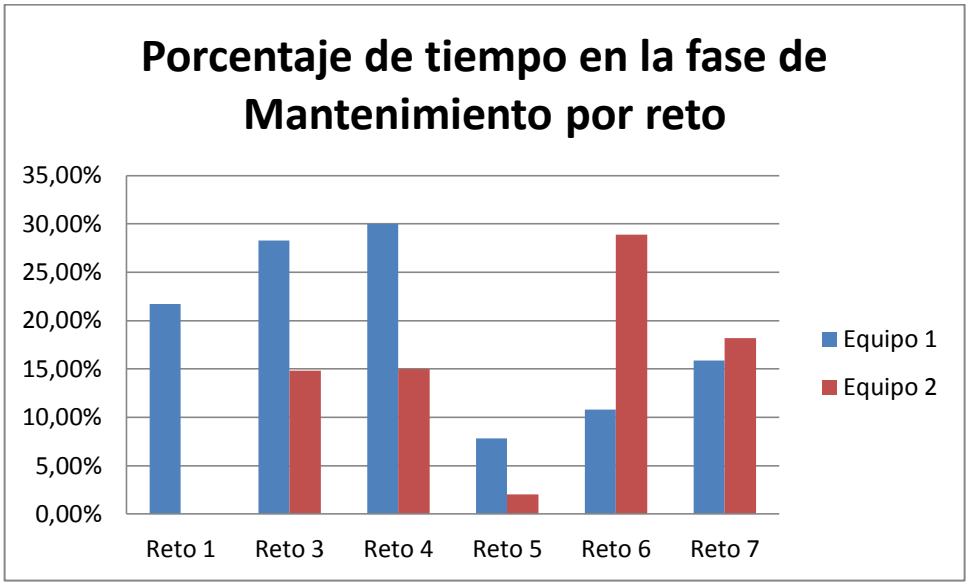

Gráfico 12: Porcentaje de tiempo utilizado en la fase de mantenimiento por reto.

El tiempo de mantenimiento se relaciona con la cantidad de entregas realizadas en los retos y el porcentaje de pruebas aceptadas en la primera versión, aunque la cantidad de entregas no es muy alta y en varios casos el porcentaje de pruebas aceptadas es alto, el tiempo dedicado al mantenimiento es más alto que el tiempo de las pruebas.

Es mejor dedicar el tiempo del mantenimiento a las pruebas para asegurar la calidad del producto, que a corregir los errores que se pudieron encontrar por medio de las pruebas y de esta manera tener un producto de alta calidad en la primera versión.

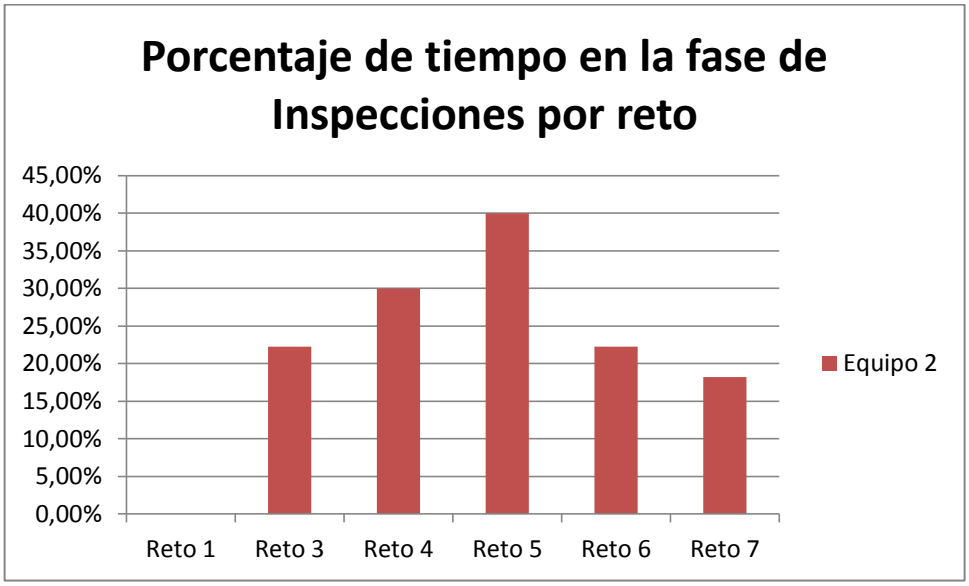

Gráfico 13: Porcentaje de tiempo utilizado en la fase de inspecciones por reto.

El tiempo utilizado en las inspecciones comenzó a disminuir gracias a la experiencia que el equipo había adquirido en esta técnica, sin embargo en los retos de complejidad mayor a 4, el porcentaje de pruebas aceptadas no fue tan alto como en los retos de complejidad entre 3 y 4. Esto se puede atribuir al intento de ahorro en el tiempo de la fase para utilizarlo en las correcciones de los defectos encontrados en las inspecciones.

La disminución del tiempo también se puede atribuir a la aplicación de las listas de chequeo, dado que el esfuerzo de encontrar los defectos en el código se disminuye por la acción de las recomendaciones en la lista de chequeo.

9. Conclusiones y Recomendaciones.

Las listas de chequeo no son efectivas al ser aplicadas después de la adquisición de experiencia en las inspecciones de código. Se recomienda aplicar las listas de chequeo en las primeras inspecciones para demostrar su utilidad y capacidad de unificar los criterios de los inspectores.

La técnica de desarrollo dirigido por pruebas no fue aplicada correctamente, en las revisiones del código entregado por parte de los equipos se evidencia la construcción de una única prueba unitaria la cual no cubre todas las clases implementadas.

Las técnicas de motivación de equipos de trabajo dan un resultado positivo con relación a la productividad de los desarrolladores, en el momento de aplicar una motivación (en este caso la baja calificación), el equipo aumento su productividad, mostrando un mayor interés por resolver los retos propuestos.

La experiencia de los integrantes del equipo de trabajo es importante en el resultado de la calidad de los desarrollos. La cantidad de defectos fue mínima en los retos de baja complejidad programática aplicando las técnicas ágiles de desarrollo dirigido por pruebas es inspecciones de código. Sin embargo, en los retos de alta complejidad se obtuvieron más defectos en la primera versión aunque la experiencia en la aplicación de las técnicas era mayor. Su experiencia en programación no permitió que se encontraran más defectos tanto en inspecciones como en la no identificación de casos de prueba. Se recomienda mejorar el formato de hoja de vida y complementarlo con una prueba sencilla con el fin de medir la experiencia de los integrantes del equipo.

Para obtener mejores resultados se recomienda ejecutar el experimento con al menos 4 grupos de 5 personas cada uno (seleccionadas mediante una mejora en el formato de hoja de vida en conjunto con una prueba sencilla de programación), aplicar al menos 10 retos, con tiempo limitado y de diversos niveles de complejidad programática, aplicando las técnicas de desarrollo dirigido por pruebas e inspecciones de código de manera conjunta en al menos 2 grupos.

10. Bibliografía.

Aurum, A., Petersson, H., & Wohlina, C. (2002). State-of-the-Art: Software Inspections after 25 Years. *Software Testing, Verification and Reliability, 12*(3), 133-154.

Beck, K. (1999). *Extreme Programming Explained.* Boston: Addison-Wesley Professional.

Bird, C., Nagappan, N., Devanbu, P., Gall, H., & Murphy, B. (2009). Does Distributed Development Affect Software Quality? An Empirical Case Study of Windows Vista. *Communications of the ACM*, 85-93.

Bisant, D., & Lyle, J. (1989). A Two Person Inspection Method to Improve Programming Productivity. *IEEE Transactions on Software Engineering.*

Blé, C., Beas, J. M., Gutiérrez, J., Reyes, F., & Mena, G. (2010). *Diseño Ágil con TDD.* Creative Commons.

Boegh, J., Depanfilis, S., Kitchenham, B., & Pasquini, A. (1999). A Method for Software Quality: Planning, Control, and Evaluation. *IEEE Software*, 69-77.

Boehm, B. (1981). *Software Engineering Economics.* Nueva Jersey: Englewood Clifs.

Briand, L. C., Wüst, J., Daly, J. W., & Porte, V. (2000). Exploring the Relationships between Desing Measures and Software Quality in Object – Oriented Systems. *Journal of Systems and Software*, 245–273.

Brito e Abreu, F., & Melo, W. (1996). Evaluating the Impact of Object – Oriented Desing Software Quality. *Proceedings of the 3rd International Software Metrics Symposium*.

Brooks, F. P. (1995). *The Mythical Man-Month.* Boston: Addison-Wesley.

Bryan, S., du Boulay, B., & Romero, P. (2006). XP and Pair Programming practices. *PPIG Newsletter*.

Burnstein, I., Suwannasart, T., & Carlson, C. R. (1996). Developing a Testing Maturity Model: Part I. *Crosstalk*.

Casallas, R. (2009). *Inspecciones de Software.* Bogotá: Universidad de los Andes.

Ciolkowski, M., Laitenberger, O., Rombach, D., Shull, F., & Perry, D. (2002). Software Inspections, Reviews & Walkthroughs. *Communications of ACM*, 641-642.

Cockburn, A. (2000). *Agile Software Development.* The Agile Software Development Series Editors.

Fagan, M. (1976). Design and code inspections to reduce errors in program. *IBM SYST J*, 183-211.

Fagan, M. (1986). Advances in Software Inspections. *IEEE Transactions on Software Engineering*, 744-751.

Gately, A. (1999). Design and Code Inspection Metrics. *ASM*.

Geoff Dromey, R. (1996). Cornering the Chimera. *IEEE Software*, 33-43.

Hilburn, T. B., & Towhidnejad, M. (2000). Software Quality: A Curriculum Postscript? *SIGCSE technical symposium on Computer science education*, 167-171.

Hulkko, H., & Abrahamsson, P. (2005). A Multiple Case Study on the Impact of Pair Programming on Product Quality. *Communications of ACM*.

Humphrey, W. (2000). *Introduction to the Team Software Process.* Boston: Addison-Wesley Professional.

Huo, M., Verner, J., Zhu, L., & Babar, M. A. (2004). Software Quality and Agile Methods. *28th Annual International Computer Software and Applications Conference*.

Informática, L. O. (03 de 12 de 2012). *PMOinformática.com.* Recuperado el 23 de 09 de 2013, de Test Driven Development (TDD): Ventajas y desventajas: http://www.pmoinformatica.com/2012/12/test-driven-development-ventajas-y.html

Janzen, D., & Saiedian, H. (2008). Does Test-Driven Development Really Improve Software Design Quality? *IEEE Software*.

Jonnes, C. (2010). *Software Engineering Best Practices: Lessons from Successful Projects in Top Companies.* New York: McGraw-Hill.

Jonnes, C. (2012). *Software Quality in 2012: A Survey of the State of the Art.*

Kantorowitz, E., Guttman, A., & Arzi, L. (1997). The Performance of the N – Fold Requirements Inspection Method. *Requirements Engineering Journal*.

Khoshgoftaar, T., & Seliya, N. (2004). Comparative Assessment of Software Quality Classification Techniques: An Empirical Case Study. *Empirical Software Engineering*, 229–257.

Kitchenham, B., & Pfleeger, S. L. (1996). Software Quality: The Elusive Target. *IEEE Software*, 12-21.

Knight, J., & Myers, E. A. (1993). An Improved Inspection Technique. *Communications of ACM*.

López, M. (2013). *Hacia la Identificación de un Método Universal de Calidad de Software.* Bogotá.

Martin, J., & Tsai, W. (1990). N – Fold Inspection: A Requirements Analysis Technique. *Communications of ACM*, 225-232.

Mathur, S., & Malik, S. (2010). Advancements in the V-Model. *International Journal of Computer Applications*, 29-34.

McConnell, S. (1996). *Rapid Development. Taming Wild Software Schedules.* Microsoft Press.

McConnell, S. (2004). *Code Complete. A Practical Handbook of Software Construction.* Microsoft Press.

Nagappan, N., Murphy, B., & Basili, V. R. (2008). The Influence of Organizational Strcuture on Software Quality: An Empirical Case Study. *Communications of ACM*.

Nguyen, C. D., Perini, A., & Tonella, P. (2007). A Goal-Oriented Software Testing Methodology. *Agent – Oriented Software Engineering VIII*.

Notenboom, E. (n.d.). Multiple V – model in relation to testing.

Ortega, M., Pérez, M. A., & Rojas, T. (2003). Contruction of a Systemic Quality Model for Evaluating a Softwate Product. *Software Quality Journal*, 219-242.

Parnas, D. L., & Weiss, D. M. (1985). Active Desing Reviews: Principles and Practices'. *8th international conference on Software engineering*, 132-136.

Porter, A. A., Siy, H. P., & Votta, L. G. (1995). A Review of Software Inspections. *Advances in Computers*.

Porter, A. A., Siy, H. P., Toman, C. A., & Votta, L. G. (1997). An Experiment to Assess the Cost – Benefits of Code Inspections in Large Scale Software Development. *IEEE Transactions on Software Engineering*, 329-346.

Raymond, E. (2013). Pair Programming. *Integrated Introduction to CS*.

Suma, V., & Gopalakrishnan, N. (2009). Defect Management Strategies in Software Development. *Recent Advances in Technologies*, 379-404.

Tang, A., Tran, M. H., Han, J., & van Vliet, H. (2008). Design Reasoning Improves Software Design Quality. *Springer – Verlag Berlin Heidelberg*, 28-42.

Travassos, G. H., Shull, F., Fredericks, M., & Basili, V. R. (1999). Detecting Defects in Object Oriented Desings: Using Reading Techniques to Increase Software Quality. *Conference on Object – Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

Wiegers, K. E. (1995). Improving Quality Through Software Inspections. *Process Impact*.

Zamiriano, R. F. (2007). *Las Inspecciones de Software y las Listas de Comprobación.* Universidad del Valle.

11. Anexos.

11.1. Formato de hoja de vida.

POLITÉCNICO GRANCOLOMBIANO

Formato de Hoja de vida

Nombres: _____

Celular: _____ email: _____

El objetivo de esta encuesta es recopilar información acerca de su experiencia en determinados aspectos vitales para el proceso a desarrollar, le solicitamos responder las siguientes preguntas con total honestidad.

1. **¿Cuánto tiempo tiene de experiencia en el lenguaje JAVA?**
   a) **Menos de un año.**
   b) **Entre uno y tres años.**
   c) **Entre tres y cinco años.**
   d) **Más de cinco años.**

2. **¿La experiencia adquirida en el lenguaje JAVA es?**
   a) **Solo académica.**
   b) **Solo en la industria de software.**
   c) **En ambas.**

3. **Si usted tiene experiencia en la industria de software, indique cuantos años tiene de experiencia:**
   a) **No tiene experiencia en la industria de software.**
   b) **Menos de un año.**
   c) **Entre uno y tres años.**

d) Entre tres y cinco años.
e) Más de cinco años.

4. ¿Cuánto tiempo tiene de experiencia trabajando en equipos de desarrollo?
   a) Menos de un año.
   b) Entre uno y tres años.
   c) Entre tres y cinco años.
   d) Más de cinco años.

5. Califique de 1 a 5 (siendo 1 el más bajo y 5 el más alto) sus conocimientos en las siguientes herramientas:

	1	2	3	4	5
JAVA					
Manejo de archivos					
Bases de datos (JDBC)					
Interfaces gráficas (Swing)					

6. Indique la cantidad de proyectos que ha realizado con las siguientes herramientas:

	Menos de 1 año	Entre 1 y 3 años	Entre 3 y 5 años	Más de 5 años
JAVA				
Manejo de archivos				
Bases de datos (JDBC)				
Interfaces gráficas (Swing)				

11.2. Formato de lista de chequeo.

POLITECNICO GRANCOLOMBIANO

Inspecciones de Código

Fecha:
Autor:
Inspector:
Clase:
Método:

Ítem	SI	NO	N/A	Observaciones
¿La clase se encuentra en el paquete correcto?				
¿Todas las variables están inicializadas?				
¿Se implementa más cosas que las especificadas en el diseño?				
¿Se implementa menos cosas que las especificadas en el diseño?				
¿Los nombres de las clases corresponden a los nombres especificados en el diseño?				
¿Los métodos corresponden con la signatura especificada en el diseño?				
¿Todos los métodos tienen el número correcto de parámetros?				
¿Todos los parámetros de los métodos tienen los tipos correctos?				
¿Todos los métodos retornan el tipo de dato correcto?				
¿Las Excepciones se están capturando correctamente?				

¿Los ciclos implementados terminan?				
¿Los archivos abiertos se cierran correctamente después de ser utilizados?				
¿Los arreglos utilizados son del tamaño suficiente?				
¿Todas las variables declaradas son utilizadas?				
¿Todos los métodos declarados son utilizados?				
¿La variable que recibe el cálculo de tipos numéricos es del tipo adecuado?				
¿Existen muchos ciclos anidados? Máximo 2 ciclos anidados				
¿Hay líneas de código innecesarias?				