UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

"MEJORA Y EVALUACIÓN DEL PAQUETE DE PUESTA EN OPERACIÓN DE CONSTRUCCIÓN Y PRUEBAS UNITARIAS DEL ESTÁNDAR ISO/IEC 29110 BASIC VSE PROFILE"

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN INGENIERÍA

(COMPUTACIÓN)

P R E S E N T A:

JAVIER FLORES FLORES

DIRECTORA DE TESIS:

DRA. HANNA J. OKTABA

México, D.F. **2011.**

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Agradecimientos

A mi directora de tesis, la Dra. Hanna J. Oktaba, por su motivación y apoyo incondicional en todo momento.

A todas las personas que participaron e hicieron posible este proyecto, muchas gracias por su apoyo y enseñanza:

- Dra. Hanna J. Oktaba
- Mtra. Ana Vázquez
- Dr. Claude Laporte
- Dr. Roger Champagne
- M. En C. María Guadalupe Elena Ibargüengoitia González
- Mtro. Gustavo Arturo Márquez Flores
- Mtia. Elsa Ramírez Hernández
- Dra. María del Pilar Ángeles

Y en especial a mi padre, por ser el mejor ejemplo a seguir de la perseverancia y dedicación al trabajo, a mi madre, por el optimismo y apoyo que me ha dado toda la vida, y a mi pareja, por su paciencia y por estar siempre conmigo en los momentos más difíciles.

Este trabajo es todos ustedes. Muchas gracias

Javier Flores Flores.

ÍNDICE

1. Introducción 5
1.1 Problemática 6
1.2 Objetivos 7
1.2.1 Objetivo general 7
1.2.2 Objetivos específicos 7
2. Marco teórico y estado del arte 8
2.1 Normas y estándares de evaluación y mejora de procesos de desarrollo de software (Iniciativas para grandes organizaciones) 8
2.1.1 CMMI (Integración de Modelos de Madurez de Capacidades) 8
2.1.2 ISO/IEC 12207 9
2.2 Normas y estándares de mejora de procesos de desarrollo de software (Iniciativas para pequeñas organizaciones) 11
2.2.1 VSE (Very Small Entities – Pequeñas organizaciones) 11
2.2.2 Porque las VSE no usan estándares 12
2.2.3 Beneficios de certificarse en algún estándar o norma 13
2.2.4 Surgimiento y objetivo del grupo de trabajo WG24 del ISO/IEC JTC1/SC7 13
2.2.5 MoProSoft 14
2.2.6 ISO/IEC 29110 15
2.2.7 Paquetes de Puesta en Operación (PPO) 18
2.2.8 Justificación y necesidad de un nuevo PPO 19
3. PPO de Construcción y Pruebas unitarias (Versión 0.4) 20
3.1.1 Perspectiva general del paquete 20
3.1.2 Fortalezas 20
3.1.3 Debilidades 21
3.1.4 Análisis de la estructura del paquete y cambios propuestos para la nueva versión 21
4. PPO de Construcción y pruebas unitarias (Versión 0.5) 29
4.1.1 Perspectiva general 29
4.1.2 Cambios realizados a la estructura de la versión 0.4 29
4.1.3 Aportaciones más importantes de la versión 0.5 y algunas comparativas entre ambas versiones. 31
4.1.4 Fortalezas 44

4.1.5 Debilidades 44
5. Propuesta para evaluar la comprensión de la nueva versión del Paquete 45
5.1 Objetivos de la evaluación 45
5.1.1 Objetivos a largo plazo 45
5.1.2 Objetivo a corto plazo 45
5.2 Estrategia de investigación seleccionada 46
5.3 El proceso de la encuesta 47
5.3.1 Identificar los objetivos de investigación 47
5.3.2 Identificar y caracterizar la audiencia objetivo 49
5.3.3 Diseñar el plan de muestreo 50
5.3.4 Diseñar y escribir el cuestionario 51
5.3.5 Realizar una prueba piloto del cuestionario 63
5.3.6 Distribuir el cuestionario 63
5.3.7 Analizar los resultados y reportarlos 66
6. Prácticas y actividades adicionales de apoyo al Paquete 76
6.1 Técnica de Prevención de Defectos: FMEA – *Failure Modes and Effects Analysis* 76
6.2 Base de conocimientos del Desarrollador 79
6.3 Reglas de oro para Pruebas unitarias 80
7. Conclusiones 81

1. Introducción

De acuerdo a la Organización para la Cooperación Económica y el Desarrollo (OECD, por sus siglas en inglés) las pequeñas y medianas empresas (PYMES) representan más del 95% de las empresas en todo el mundo (OECD, 2002). Dentro de éste grupo se encuentran también las pequeñas organizaciones de desarrollo de software (VSE de aquí en adelante, Very Small Entities, por sus siglas en inglés (Laporte C. Y., A Software Engineering Lifecycle Standard for Very Small Enterprises, 2008)) las cuales de acuerdo a estudios realizados (INTECO, 2008) representan más del 90% de las empresas europeas de desarrollo de software.

Sin embargo existe la problemática de que la mayoría de los estándares y normas actuales no satisfacen las necesidades de éste grupo mayoritario, ya que fueron diseñados específicamente para grandes empresas de desarrollo, las cuales al contar con mayores recursos tanto económicos como humanos, pueden invertir grandes cantidades de dinero y esfuerzo en certificaciones y evaluaciones como el *Standar CMMI Appraisal Method for Process Improvement* (SCAMPI) o el *Software Process Improvement Capability Determination* (SPICE) para evaluar los modelos CMMI® o ISO/IEC 12207 respectivamente.

Por esto surgió la necesidad de crear iniciativas que delimitan el alcance y se adaptan a las necesidades de las VSEs, surgiendo así modelos y normas como MoProSoft (Modelo de Procesos para la Industria de Software), I.T.Mark™ (del European Software Institute) , y una de las más recientes la norma ISO/IEC 29110, la cual se compone de un conjunto de perfiles basados en procesos y prácticas de otros estándares ISO/IEC y de guías que han sido desarrolladas de acuerdo a un conjunto de características de las VSEs, de tal manera que sean adaptables al contexto correspondiente.

Sin embargo la adopción de un nuevo estándar no siempre resulta sencilla, por ello para el estándar ISO/IEC 29110 se han creado una serie de herramientas que sirven de apoyo para ésta tarea. Dichas herramientas se llaman Paquetes de Puesta en Operación (PPO) (*Deployment packages* en inglés (Laporte C. Y., The Application of International Software Engineering Standards in Very Small Enterprises, 2008 - I, pág. 7)) y el objetivo de esta investigación es la creación y evaluación de la nueva versión del paquete correspondiente a las actividades de Construcción y pruebas unitarias.

Para ello el presente documento está estructurado de la siguiente forma:

En el capítulo 1 – ***Introducción***, se presenta un panorama del la problemática que se pretende atacar, así como los objetivos generales y específicos que la atienden.

En el capítulo 2 – ***Marco teórico***, se muestra algunas de las iniciativas existentes para la mejora de procesos de desarrollo de software para grandes y pequeñas organizaciones, sus características más importantes así como sus fortalezas y debilidades.

Capítulo 3 – ***PPO de Construcción y pruebas unitarias (Versión 0.4),*** en éste capítulo se resume brevemente la perspectiva general del paquete, se muestran sus fortalezas y debilidades. Se hace también un análisis de la estructura del paquete actual y se proponen y justifican algunos cambios para la nueva versión del paquete.

Capítulo 4 ***– PPO de Construcción y pruebas unitarias (Versión 0.5)***, se describe la perspectiva general del nuevo paquete, una comparativa con los cambios realizados al paquete anterior, las aportaciones más importantes, las fortalezas que se conservaron y las nuevas que surgieron, las debilidades que se resolvieron y las nuevas detectadas.

Capítulo 5 ***– Propuesta para evaluar la comprensión de la nueva versión del paquete***, aquí se define el alcance de los objetivos de la evaluación, se proponen objetivos a corto y a largo plazo, con la finalidad de delimitar el tiempo que se tiene para realizar el experimento. También se describe cada uno de los pasos del proceso de la encuesta, desde su diseño y realización hasta su distribución y análisis de los resultados.

Capítulo 6 – ***Prácticas y actividades adicionales de apoyo al paquete***. Debido a que en el proceso de investigación surgieron otros temas que son de utilidad para las actividades de construcción y pruebas unitarias, pero que por cuestiones de restricción en la cantidad de contenido que es recomendable que tenga un PPO, ya no se añadieron en la nueva versión del paquete. Se decidió mencionar dichos temas en esta sección como una extensión de las actividades del paquete y dejar a libertad del lector considerar su importancia a la hora de realizar las tareas pertinentes.

Capítulo 7 – ***Conclusiones y trabajo futuro***. Finalmente con base en los resultados obtenidos y a las críticas recibidas se concluye sobre el cumplimiento del objetivo planteado para ésta tesis y sobre las posibles líneas de trabajo futuro que queden abiertas para dar seguimiento al proyecto o a trabajos afines.

Referencias bibliográficas. Por último se enlistan las referencias consultadas para la realización del trabajo.

1.1 Problemática

En la actualidad debido a la gran competencia que existe, la creación de productos de software de calidad es algo que les concierne a las compañías de desarrollo de software sin importar el tamaño, tipo de productos o servicios que éstas ofrecen. Especialmente las VSE al estar la mayoría inmersas en ambientes de *outsourcing* resulta crítico entregar a los clientes los productos esperados en el tiempo y con la calidad adecuada.

Para solucionar éste problema han surgido una serie de estándares enfocados a la calidad de procesos de software, los cuales con el tiempo han madurado y han obtenido aceptación por la mayoría de las empresas.

Sin embargo debido a los recursos limitados con los que cuentan las VSE, estudios y encuestas han demostrado (Laporte C. Y., The Application of International Software Engineering Standards in

Very Small Enterprises, 2008 - I) que los estándares actuales de ingeniería de software, como los desarrollados por ISO/IEC y la IEEE, no se adaptan a las necesidades de éstas organizaciones y en la mayoría de los casos resulta muy difícil o en ocasiones imposible su adopción.

Entre las razones principales del porqué las VSE presentan resistencia en el uso de éstos estándares se encuentra: falta de recursos, demasiada burocracia, falta de guías, plantillas o ejemplos. La gran mayoría de las VSE encuestadas sugieren la creación de estándares más ligeros, fáciles de entender y con ejemplos o plantillas.

Afortunadamente el grupo WG24 propuso la creación de un estándar que atendiera éstas necesidades, el estándar ISO/IEC 29110 que cuenta con una serie de perfiles que atienden las características y requerimientos de las VSE (Ribaud, Saliou, & O´Connor, Software Engineering Support Activities for Very Small Entities, 2010).

Sin embargo, a pesar del hecho de que éste nuevo estándar es más ligero y comprende las partes esenciales en los procesos de desarrollo de software, su adopción aún puede presentar dificultades con el hecho de que como cualquier otro estándar se enfoca en el "Que hacer" más que en el "Cómo hacerlo". Por ello sigue siendo necesario desarrollar alguna herramienta o guía que facilite la adopción y entendimiento de los procesos de cada perfil.

Algunas de estas guías ya han sido desarrolladas y se conocen como Paquetes de Puesta en Operación, los cuales atienden áreas de procesos del estándar y del ciclo de vida del software. Sin embargo la problemática con estas guías es que algunas de ellas se encuentran desactualizadas, incompletas o carecen de suficientes ejemplos o plantillas. Además de que no han sido evaluadas por personas ajenas al grupo WG24, por lo que su esparcimiento ha sido muy limitado y con ello el punto de vista de VSE reales.

1.2 Objetivos

1.2.1 Objetivo general

Elaborar una nueva versión y evaluar la comprensión lectora del Paquete de Puesta en Operación de Construcción y pruebas unitarias, como herramienta de apoyo para la adopción de los procesos de software en las VSE de acuerdo al estándar ISO/IEC 29110 Perfil Básico.

1.2.2 Objetivos específicos

1. Elaborar y someter a aprobación la nueva versión del paquete como reporte técnico del estándar internacional ISO/IEC 29110 Perfil Básico.
2. Traducir el paquete al idioma español
3. Crear un sitio web para montar las evaluaciones
4. Aplicar las evaluaciones a por lo menos dos VSE
5. Obtener los resultados de las evaluaciones y definir conclusiones

2. Marco teórico y estado del arte

2.1 Normas y estándares de evaluación y mejora de procesos de desarrollo de software (Iniciativas para grandes organizaciones)

Actualmente los modelos de evaluación y mejora de procesos han tomado un papel determinante en la identificación, integración, medición y optimización de las buenas prácticas existentes sobre desarrollo software. Entre los más conocidos se encuentran:

2.1.1 CMMI (Integración de Modelos de Madurez de Capacidades)

Éste modelo constituye un marco de referencia de la capacidad de las empresas de desarrollo de software en el desempeño de sus diferentes procesos, proporcionando una base para la evaluación de la madurez de las mismas y una guía para implementar una estrategia para la mejora continua de los mismos. (De la villa, 2004)

Estudia los procesos de desarrollo y produce una evaluación de la madurez (indicador para medir la capacidad para construir software de calidad) de la empresa según una escala de cinco niveles (inicial, repetible, definido, dirigido y optimizado). Los modelos describen el camino para evolucionar y mejorar desde procesos inmaduros a procesos disciplinados, maduros con calidad y eficiencia mejorada.

CMMI presenta dos representaciones del modelo: continua (capacidad de cada área de proceso) y/o por etapas (madurez organizacional)

En la representación por etapas cada nivel de madurez tiene un conjunto de áreas de proceso que indican donde una organización debería enfocar la mejora de su proceso. Cada área de proceso se describe en términos de prácticas que contribuyen a satisfacer sus objetivos. Las prácticas describen las actividades que más contribuyen a la implementación eficiente de un área de proceso; se aumenta el 'nivel de madurez' cuando se satisfacen los objetivos de todas las áreas de proceso de un determinado nivel de madurez (Ver tabla 1).

Nivel de madurez de la organización	Centrado en
1. Inicial	Procesos impredecibles, control reactivo
2. Gestionado	Gestión básica del proyecto
3. Definido	Proceso caracterizado por la organización y proactivo
4. Gestionado cuantitativamente	Control cuantitativo del proceso
5. Optimizado	Mejora continua del proceso

Tabla 1 Áreas en representación por etapas

En la representación continua, la capacidad de cada área de proceso se enfoca en una línea a partir de la que medir la mejora individual, en cada área. Al igual que el modelo por etapas, el modelo continuo tiene áreas de proceso que contienen prácticas, pero éstas se organizan de manera que soportan el crecimiento y la mejora de un área de proceso individual.

2.1.1.1 Fortalezas

- ✓ Inclusión de prácticas que permiten asegurar que los procesos asociados con cada área de proceso sean efectivos, repetibles y duraderos.
- ✓ Guía paso a paso para la mejora, a través de niveles de madurez y capacidad.
- ✓ Transición del "aprendizaje individual" al "aprendizaje de la organización" por mejora continua, lecciones aprendidas y uso de bibliotecas y bases de datos de proyectos mejorados.

2.1.1.2 Debilidades

- ✗ El CMMI puede llegar a ser excesivamente detallado para algunas organizaciones.
- ✗ Puede ser considerado prescriptivo.
- ✗ Requiere mayor inversión para ser completamente implementado.
- ✗ Puede ser difícil de entender.
- ✗ No existencia de una guía a la medida de pequeñas organizaciones
- ✗ Demasiado grande para pequeñas empresas (gran cantidad de áreas, prácticas y recursos.
- ✗ Demasiado normativo, en especial para pequeñas empresas que, además, funcionan y evolucionan de distinta manera que las grandes.

2.1.2 ISO/IEC 12207

El estándar internacional ISO/IEC 12207 en su versión más reciente (IEEE, 2008) establece un marco de trabajo común para los procesos del ciclo de vida del software, con terminología bien definida que puede ser referenciada por la industria de software. Este estándar contiene procesos, actividades y tareas que pretenden ser aplicadas durante la adquisición de un producto o servicio de software o durante la entrega, desarrollo, operación, mantenimiento y disposición de productos de software que se realicen de manera interna o externa en una organización.

El propósito de este estándar es proveer un conjunto definido de procesos para facilitar la comunicación entre los compradores, proveedores, desarrolladores, personal de mantenimiento, operadores, gestores y técnicos involucrados en el desarrollo de software para que usen un lenguaje común. Este lenguaje común se establece en forma de procesos bien definidos.

Los procesos se clasifican en tres tipos: Principales, de soporte y de la organización. Los procesos de soporte y de organización deben existir independientemente de la organización y del proyecto ejecutado. Los procesos principales se instancian de acuerdo a un proyecto o situación en particular.

- Procesos principales.
  - Adquisición.
  - Suministro.
  - Desarrollo.
  - Operación.
  - Mantenimiento.

- Procesos de soporte.
    - Documentación
    - Gestión de la configuración.
    - Aseguramiento de calidad.
    - Verificación.
    - Validación.
    - Revisión conjunta.
    - Auditoría.
    - Resolución de problemas.

- Procesos de la organización.
    - Gestión.
    - Infraestructura.
    - Mejora.
    - Recursos Humanos.

Cada proceso del estándar es descrito en términos de los siguientes atributos:

- Nombre del proceso
- Propósito del proceso
- Salidas (*outcomes*) esperadas de la realización exitosa del proceso
- Actividades necesarias para obtener la salidas esperadas
- Tareas que sirvan de apoyo en las actividades

2.1.2.1 Fortalezas

- ✓ Estándar internacional dedicado exclusivamente al ciclo de vida del software
- ✓ La estructura estratificada de los procesos facilita la implementación individual de los mismos.
- ✓ Provee dos tipos de implementación: cumplimiento total y cumplimiento a la medida, permitiendo así mayor flexibilidad para aquellos proyectos que no requieran el cumplimiento de todos los procesos.
- ✓ Ampliamente aceptado por grandes organizaciones de desarrollo software

2.1.2.2 Debilidades

- ✗ La implementación total de sus procesos requiere demasiado esfuerzo y tiempo
- ✗ Las evaluaciones pueden llegar a ser muy costosas
- ✗ A pesar de que existe flexibilidad para elegir los procesos necesarios de acuerdo al proyecto, no provee una guía que delimite que procesos son los esenciales para el desarrollo de software.
- ✗ Algunos procesos pueden resultar difíciles de entender
- ✗ Demasiado grande para pequeñas empresas (gran cantidad de áreas, prácticas y recursos).

Debe quedar claro que para proyectos específicos o ciertas organizaciones puede no ser necesario el uso de todos los procesos del estándar. En vez de ello la implementación de solo algunos procesos del estándar puede ser lo más conveniente.

Gracias a esta flexibilidad es que han surgido otros modelos y normas que se enfocan en el cumplimiento de procesos específicos pero al mismo tiempo básicos en el ciclo de vida del software y que tienen como objetivo la mejora de procesos en grupos pequeños de desarrollo, los cuales tienen otras necesidades y en la mayoría de los casos muy poca experiencia en el resto de los procesos.

De estos estándares y normas se hablará más adelante, ya que son parte del objeto de estudio de éste documento.

2.2 Normas y estándares de mejora de procesos de desarrollo de software (Iniciativas para pequeñas organizaciones)

El software forma una parte clave para el crecimiento y supervivencia de casi cualquier empresa. Por ello hoy en día existen cada vez más organizaciones que se dedican a su desarrollo. No obstante hay toda una cadena detrás de su fabricación. La mayor parte de las empresas que se encuentran a la cabeza y que generalmente son grandes organizaciones requieren de uno o más proveedores que fabriquen partes específicas del software. Dichos proveedores en la mayoría de los casos son organismos pequeños y juntos forman el último eslabón en la cadena de producción del software. Pero no por eso son menos importantes ya que la falla de uno de estos eslabones puede hacer que se desplome toda la cadena e incurrir en cuantiosos costos en el desarrollo del producto.

2.2.1 VSE (Very Small Entities – Pequeñas organizaciones)

Estudios y encuestas demuestran que estas pequeñas organizaciones forman la gran mayoría de las empresas de TI. En Europa por ejemplo 85% de las compañías del sector de TI tienen de 1 a 10 empleados. En Canadá, una encuesta en el área de Montreal revela que 78% de las empresas de desarrollo de software tienen menos de 25 empleados y 50% tienen menos de 10 empleados (Laporte C. Y., The Application of International Software Engineering Standards in Very Small Enterprises, 2008 - I). En Brasil pequeñas empresas de IT representan cerca del 70% del número total de compañías (Anacleto, 2004, págs. 33-37).

No obstante existe cierta ambigüedad en la definición de que tan pequeña es una pequeña empresa con respecto a su número de sus empleados, por lo que de ahora en adelante se tomará la definición dada por (Laporte, April, & Renault, 2006): Una VSE (Very Small Entity) es “Cualquier servicio de IT, organización o proyecto de entre 1 a 25 empleados”.

2.2.1.1 Características

La mayoría de las VSE comparten ciertas características que se pueden clasificar en cuatro principales categorías:

Finanzas

Las VSE son económicamente vulnerables ya que generalmente dependen del flujo de efectivo que originan los proyectos. Por lo que resulta de gran importancia acatarse al presupuesto estimado. Sin embargo en la mayoría de los casos eso no es posible y no quedan fondos en el presupuesto para destinarlo a otras actividades importantes como: mantenimiento post-entrega, entrenamiento, actividades de aseguramiento de la calidad, manejo de riesgos, mejora de procesos, certificaciones o evaluaciones etc.

Tipo y relación con los clientes

En las VSE generalmente ocurre que cada producto tiene un único cliente, el cual es el encargado de administrar, instalar y operar el sistema. Además resulta normal que la satisfacción de éste se base en el cumplimiento de requerimientos específicos que pueden cambiar a lo largo del proyecto. Sin considerar ninguna estrategia cuantitativa para medir la calidad de lo que se pide. El contacto y la comunicación constante con el cliente resultan ser un factor de éxito en los proyectos.

Procesos de negocio internos

Los productos de software elaborados generalmente no tienen relación con otros. La administración de proyectos es típicamente realizada por medio de mecanismos informales, con la mayoría de la toma de decisiones, problemas de comunicación y resolución de problemas realizados de manera interpersonal.

Aprendizaje y crecimiento

Estas características se encuentran limitadas debido a la falta de conocimiento en evaluación de procesos y a la falta de recursos humanos que participen en la estandarización.

2.2.2 Porque las VSE no usan estándares

Estudios realizados (Laporte C. Y., The Application of International Software Engineering Standards in Very Small Enterprises, 2008 - I, pág. 5) indican que de un total de VSE encuestadas menos del 18% se encuentran certificadas, y dentro de ese 18% un 75% no usan estándares. Los motivos o razones por las que dichas organizaciones no siguen o adoptan estándares son los que se muestran en la Figura 1.

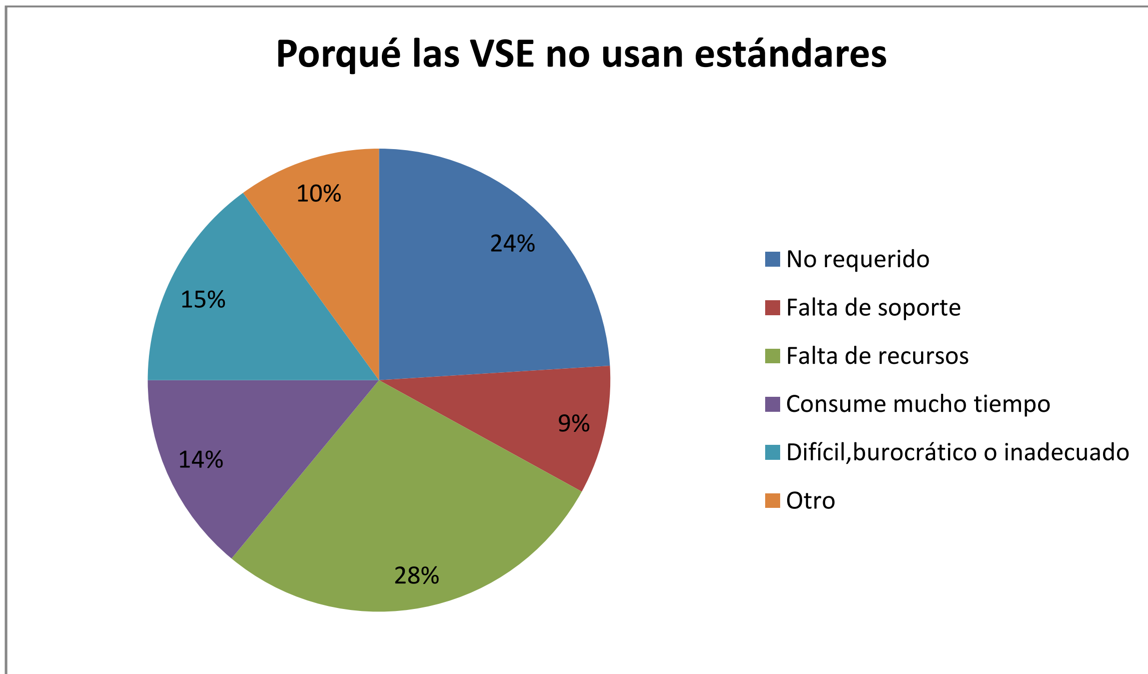


Figura 1 Resultados de encuesta realizada por el grupo WG24. (Laporte C. Y., The Application of International Software Engineering Standards in Very Small Enterprises, 2008 - I)

A pesar de éstas estadísticas la mayoría de las VSE reconocen la importancia de ser evaluadas o certificadas en algún estándar.

2.2.3 Beneficios de certificarse en algún estándar o norma

Desde el punto de vista de las VSE, algunos de los beneficios que provee una certificación se encuentran:

- ✓ Incremento de la competitividad
- ✓ Mayor confianza y satisfacción de los clientes
- ✓ Mayor calidad en los productos de software
- ✓ Disminución del riesgo
- ✓ Reconocimiento y mayor prestigio

2.2.4 Surgimiento y objetivo del grupo de trabajo WG24 del ISO/IEC JTC1/SC7

La reunión llevada a cabo en Australia en el 2004 por miembros del ISO/IEC JTC1/SC7[1] llegó al consenso de que las VSE necesitan estándares adaptados a un tamaño y contexto correspondiente, incluyendo una serie de perfiles y guías. Algunos de los objetivos generales fueron los siguientes:

- ✓ Hacer los estándares de ingeniería de software más accesibles para las VSE

---

[1] El objetivo de ISO/IEC JTC1/SC7 es la estandarización de procesos, herramientas y tecnologías de soporte para la ingeniería de productos y sistemas de software.

- ✓ Proveer documentación que requiera el mínimo ajuste y esfuerzo de adaptación.
- ✓ Proveer documentación que integre los estándares disponibles, tales como estándares de procesos, productos de trabajo y entregables, evaluación y calidad.
- ✓ Alinear los perfiles a las nociones de niveles de madurez de acuerdo al estándar ISO/IEC 15504.

En el 2005 en una reunión del SC7 en Tailandia se propuso la creación de un nuevo grupo de trabajo que atendiera dichos objetivos, surgiendo de ésta manera el grupo WG24 el cual fue apoyado por doce países: Bélgica, Canadá, República checa, Irlanda, Italia, Japón, Corea, Luxemburgo, Sudáfrica, Tailandia, el Reino Unido y los Estados Unidos.

El enfoque considerado por el WG24 para desarrollar un nuevo estándar que cumpliera con los objetivos consiste en los siguientes pasos:

- ✓ Seleccionar un subconjunto de procesos de ISO/IEC 12207 aplicables a las VSE de menos de 10 empleados.
- ✓ Ajustar el subconjunto para que satisfaga las necesidades de las VSE
- ✓ Desarrollar directrices.

A partir de aquí el grupo WG24 comenzó a buscar referencias a estándares internacionales y modelos que estuvieran basados en algún subconjunto del ISO/IEC 12207para VSE con niveles bajos de madurez. La norma mexicana MoProSoft, fue el modelo que mejor cumplía con dicho objetivo y fue seleccionada para ser ajustada y producir así un nuevo estándar internacional.

2.2.5 MoProSoft

En el año 2002, la Secretaría de Economía de México inició el Programa de Desarrollo de Software (PROSOFT) cuyo objetivo es el fortalecimiento de la industria del software en México, definió una serie de estrategias para lograr dicho objetivo. Entre una de ellas se encuentra el alcanzar niveles internacionales en capacidad de procesos, la cual fue delegada a la Asociación Mexicana para la Calidad en la Ingeniería de Software (AMCIS) y haciendo una selección y ajuste de las prácticas de modelos y estándares como ISO 9001:2000, ISO/IEC 12207 y CMMI surgió el Modelo de Procesos para la Industria de Software (MoProSoft) en México en su primera versión en ese mismo año.

MoProSoft está enfocado a procesos y cuenta con un modelo de capacidades basado en ISO/IEC 15504-2. Además considera los tres niveles básicos de la estructura de una organización que son: la Alta Dirección, Gestión y Operación. Cada uno de estos niveles puede ser visto como una categoría de procesos que integran a su vez un conjunto de procesos que abordan la misma área general de actividad dentro de una organización (Ver tabla 2).

Categoría de procesos	Propósito
Alta dirección	Aborda las prácticas de Alta Dirección relacionadas con la gestión del negocio. Proporciona los lineamientos a los procesos de la Categoría de Gerencia y se retroalimenta con la información generada por ellos.
Gerencia	Aborda las prácticas de gestión de procesos, proyectos y recursos en función de los lineamientos establecidos en la Categoría de Alta

	Dirección. Proporciona los elementos para el funcionamiento de los procesos de la Categoría de Operación, recibe y evalúa la información generada por éstos y comunica los resultados a la Categoría de Alta Dirección
Operación	Aborda las prácticas de los proyectos de desarrollo y mantenimiento de software. Esta categoría realiza las actividades de acuerdo a los elementos proporcionados por la Categoría de Gerencia y entrega a ésta la información y productos generados.

Tabla 2 categorías de procesos de MoProSoft (Oktaba, 2005, pág. 10).

MoProSoft ha demostrado ser un modelo de procesos de software a seguir en otros países, pues después de diversas presentaciones a nivel internacional ha sido seleccionado como base para el Programa Iberoamericano de Ciencia y Tecnología de Desarrollo (CYTED), en el proyecto de Mejora de procesos para fomentar la competitividad de las PyMES[2] en Iberoamérica (COMPETISOFT)

Además también fue aceptado como base para una nueva norma, la ISO/IEC 29110: "Software Engineering – Lifecycle Profiles for Very Small Enterprises (VSE)", ubicándolo a la cabeza de los estándares internacionales para PyMES.

2.2.6 ISO/IEC 29110

Una vez seleccionado a MoProSoft como el modelo a seguir, lo siguiente fue definir grupos de perfiles[3] que fueran aplicables a más de una categoría de VSE. De aquí surgió el perfil genérico, el cual está enfocado a aquellas VSE que no generan software crítico. El grupo se compone de cuatro perfiles (De entrada, Básico, Intermedio y Avanzado) de manera que pueda proveer un enfoque progresivo para satisfacer las necesidades de una gran mayoría de las VSE.

El estándar ISO/IEC 29110 se compone de 5 partes, bajo el título general: *Ingeniería de Software - Perfiles de ciclo de vida para Microempresas (VSE)*, Ver tabla 3.

ISO/IEC 29110	Título	Audiencia
Parte 1	Visión general	VSEs.
Parte 2	Marco de trabajo y Taxonomía	Productores de estándares, vendedores de herramientas y metodologías. No enfocado a las VSEs
Parte 3	Guía de evaluación	Asesores y VSEs
Parte 4	Especificaciones de perfil	Productores de estándares, vendedores de herramientas y metodologías. No enfocado a las VSEs
Parte 5	Guías de administración e ingeniería	VSEs.

Tabla 3 Audiencia objetivo de ISO/IEC 29110

[2] PYMES: Pequeñas y medianas empresas. Este grupo incluye a las ya mencionadas VSEs (Very Small Entities - pequeños grupos de desarrollo de software).
[3] Un perfil promueve la integración de estándares base definiendo el cómo usar una combinación de dichos estándares para una función o ambiente determinado en una VSE.

En relación a éstas partes se han generado los siguientes documentos:

ISO/IEC 29110-1: Introduce los conceptos principales para entender y usar el conjunto de documentos de la familia 29110. Introduce los aspectos del negocio, características y requerimientos de las VSEs, y clarifica la justificación de los perfiles específicos para las VSE, los documentos, estándares y guías.

ISO/IEC 29110-2: Establece la lógica detrás de la definición y aplicación de los perfiles, los elementos comunes entre ellos, así como una taxonomía (catálogo) de los mismos.

ISO/IEC 29110-3: Define las directrices de evaluación de procesos y el cumplimiento de los requerimientos necesarios para lograr los objetivos definidos por los perfiles de las VSEs.

ISO/IEC 29110-4-1: Provee la especificación de todos los perfiles del *Grupo de Perfiles Genéricos.* Éste grupo es aplicable a VSEs que no generar productos de software crítico. Los perfiles están basados en subconjuntos de elementos de estándares apropiados.

ISO/IEC 29110-5-1-2: Provee una guía de implementación de administración e ingeniería para el *Perfil Básico* del *Grupo de Perfiles Genéricos* descritos en ISO/IEC 29110 Parte 4-1. El *Perfil Básico* describe el desarrollo de software de una sola aplicación por un único equipo de trabajo sin riesgos especiales o factores situacionales.

La Figura 2 describe el conjunto de documentos y su posición dentro del marco de trabajo. La visión general y las guías son publicadas como Reportes Técnicos (TR – Technical Report) y los perfiles son publicados como estándares internacionales (IS – International Standard).

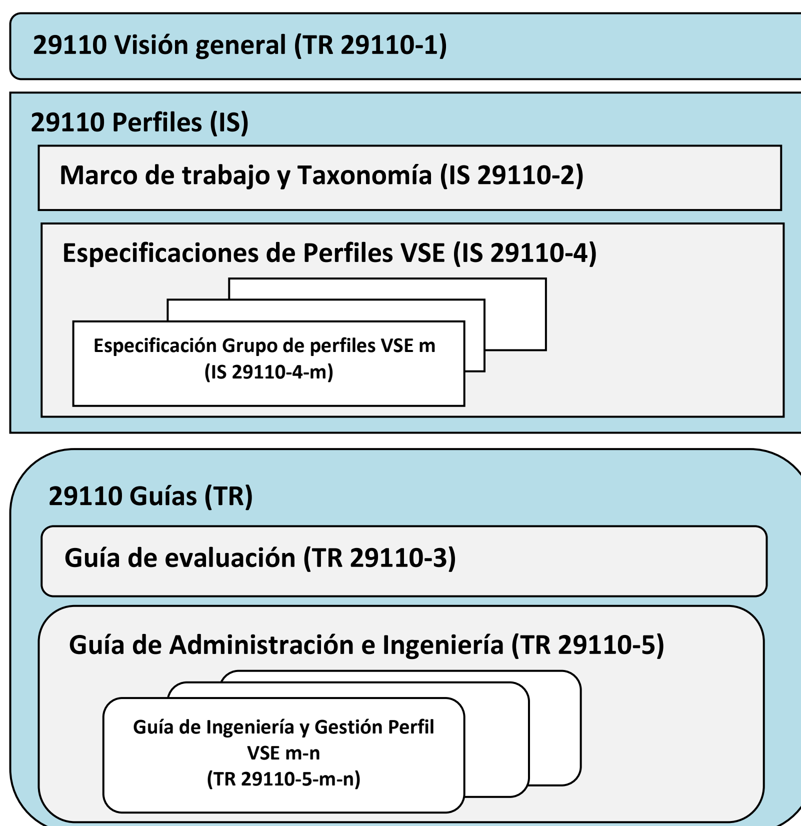


Figura 2 Conjunto de documentos ISO/IEC 29110

Alcance de ISO/IEC 29110:

- Proporcionar a las VSE un reconocimiento como productoras de software de calidad a costes reducidos
- Elaborar guías fáciles de entender y utilizables por VSE
- Producir un conjunto de perfiles que construyan o mejoren procesos
- Dar ejemplos para fomentar a las VSE la adopción y seguimiento de los procesos
- Ayudar a que las VSE puedan trabajar y puedan ser evaluadas de forma conjunta
- Elaborar perfiles y guías conformes con ISO/IEC 12207, 15504 y/o ISO 9001:2000

Hasta la fecha el perfil que ha quedado publicado es el *perfil básico*, cuyo propósito es definir una guía para el *desarrollo de software y la gestión de proyectos* para un conjunto de procesos y salidas apropiadas para las características y necesidades de las VSEs. El perfil básico describe el desarrollo de software de una única aplicación por un único equipo de trabajo. La razón principal para incluir la administración de proyectos es que la principal actividad de negocio en las VSE es el desarrollo del software y su éxito financiero depende de los beneficios del proyecto.

2.2.6.1 Parte 5-1-2 Guía de Ingeniería y Gestión – Perfil Básico VSE

La parte 5-1-2 es un Reporte Técnico (TR) que es aplicable a proyectos específicos de desarrollo de software en las VSEs (empresas de 1 a 25 personas, aunque su uso podría ser de utilidad para empresas más grandes). Provee una guía de Ingeniería y Gestión al Perfil Básico descrito en ISO/IEC 29110 Parte 4-1 a través de los procesos de *Implementación de software y Administración de proyectos* (Considerados por algunos los procesos básicos dentro de una VSE (Ribaud, Saliou, & O´Connor, Software Engineering Support Activities for Very Small Entities, 2010)).

Usando ésta guía las VSE pueden obtener beneficios en algunos de los siguientes aspectos:

- ✓ Un conjunto de requerimientos de proyecto aprobados y productos esperados es entregado al cliente.
- ✓ Se lleva a cabo la realización de un proceso disciplinado de administración de procesos que provee visibilidad del proyecto y acciones correctivas de problemas en el proyecto y desviaciones.
- ✓ Se coordina un proceso sistematizado de implementación de software que satisface las necesidades del cliente y asegura la calidad en los productos.

Sin embargo algunas partes de los procesos que abarca la guía aún pueden considerarse relativamente complejas o no muy claras para algunas VSE, por lo que son requeridos técnicas o métodos que ayuden en el entendimiento de los procesos y en la adopción del estándar. Algunas de estas herramientas de apoyo ya se han creado y se conocen con el nombre de Paquetes de Puesta en Operación*.*

2.2.7 Paquetes de Puesta en Operación (PPO)

Con el objetivo de que las VSE entiendan y adopten los estándares, algunos miembros del grupo WG24 y otros investigadores han desarrollado un conjunto herramientas llamadas Paquetes de Puesta en Operación (en inglés conocido como *Deployment Packages).*

Un PPO es un conjunto de artefactos desarrollados para facilitar la aplicación de un conjunto de prácticas de un marco seleccionado en una VSE. Los paquetes de puesta en operación que se han desarrollado hasta el momento son para ayudar a implementar los procesos del Perfil Básico.

Un PPO trata de no ser prescriptivo, es decir los pasos a seguir para cada una de sus actividades no son obligatorios, pero si recomendados para tener una mejor calidad en las tareas efectuadas.

Estructura:
El contenido de un paquete de puesta en operación típico se muestra en la Tabla 4 (Laporte C. Y., Deployment Packages and Eclipse Process Framework Project). El mapeo a los estándares y modelos es dado como información para mostrar que un PPO posee una conexión explicita con el perfil básico y con otros estándares ISO, como ISO/IEC 12207 o modelos como CMMI.

1. Descripción técnica a. Propósito de este documento b. Porqué es importante este tópico
2. Definiciones
3. Relaciones con ISO/IEC 29110
4. Descripción de procesos, actividades, tareas, pasos, roles y productos a. Descripción del rol b. Descripción del producto c. Descripción del artefacto
5. Plantilla
6. Ejemplo
7. Lista de verificación
8. Herramienta
9. Referencias a otros estándares y modelos (e.g. ISO 9001, ISO/IEC 12207, CMMI)
10. Referencias
11. Formas de evaluación

Tabla 4 Estructura de un Paquete de Puesta en Operación. (Laporte C. Y., Deployment Packages and Eclipse Process Framework Project)

Paquetes de puesta en operación existentes para el Perfil Básico VSE:

- Análisis de requerimientos
- Construcción y pruebas unitarias
- Integración y pruebas de software

- Entrega de productos
- Control de versiones
- Administración de proyecto
- Verificación y validación
- Auto-evaluación
- Arquitectura y diseño detallado

2.2.8 Justificación y necesidad de un nuevo PPO

Uno de los objetivos principales de los PPOs es que de manera autodidacta, las VSE puedan implementar las prácticas de los procesos que consideran pertinentes, sin tener para esto que alcanzar un nivel de madurez de manera horizontal en todos los procesos.

Sin embargo en la actualidad no existe evidencia sólida de la eficiencia de estos PPO, al no ser puestos en práctica por las VSE. Sin embargo se han desarrollado proyectos piloto (Ribaud, Saliou, & O´Connor, Software Engineering Support Activities for Very Small Entities, 2010) en donde empleados de las VSE han tenido cursos de entrenamiento basados en las actividades que contienen algunos PPOs.

Una de las razones del porqué estos PPOs no han sido puestos en práctica es debido a que algunos de ellos aún necesitan mayor detalle, mejores ejemplos y más plantillas para llevar a cabo las actividades que plantean. Es importante recordar que desde que la audiencia de estos paquetes es el personal de las VSE es necesario llamar su atención y darles a conocer un material lo más práctico posible pero manteniendo siempre la integridad e importancia de las tareas más importantes de cada proceso.

Cabe mencionar que aunque todos los PPOs son importantes dentro del ciclo de vida del software, uno de los más imprescindibles es el de *Construcción y pruebas unitarias,* en el cual se describen las tareas necesarias para construir lo que en algunas ocasiones es la única evidencia del trabajo de las VSE, "El Software" y en donde también se describen las pruebas que arrojan los defectos menos costosos en todo el ciclo de desarrollo, las *"Pruebas unitarias".*

Debido a esto, el presente trabajo es el resultado de la revisión crítica del PPO actual de *Construcción y pruebas unitarias*, en la cual se identificaron sus fortalezas y debilidades, y la propuesta de una nueva versión del PPO que fortalezca dichas debilidades con el objetivo final de elaborar una versión mejorada del paquete.

Además, como objetivo secundario, se pretende evaluar la nueva versión del PPO para verificar si cumple o no con su objetivo principal, que es el de transmitir conocimientos (de manera autodidacta) a trabajadores de VSE reales sobre el área de *Construcción y pruebas unitarias*.

3. PPO de Construcción y Pruebas unitarias (Versión 0.4)

Este PPO fue elaborado en el año 2009 por Ana Vázquez, miembro del grupo WG24 del ISO/IEC JTC1/SC7 y revisado y editado por Claude Laporte (École de technologie supérieure, Canada), editor en jefe del grupo WG24.

El idioma en el que fue redactado es Inglés y su nombre original es: *"Deployment Package Construction and Unit Testing Basic Profile" (versión 0.4).*

El documento (Vazquez) se encuentra disponible en el sitio público del ISO/IEC JTC1/SC7 Grupo de trabajo WG24[4], junto con el resto de los paquetes que han sido elaborados hasta el momento.

3.1.1 Perspectiva general del paquete

El documento sigue la estructura definida para los PPOs especificada por el grupo WG24 (ver punto 4.2.7 de este documento). Las tareas definidas en el paquete se apegan a las tareas de la *actividad* 7.1.5: Construcción de Software, del *proceso* 7: Implementación de Software, del Perfil Básico.

El paquete cuenta también con dos sub-tareas relacionadas con la actividad de *Análisis de Requerimientos de Software,* que aunque no se encuentran especificadas de manera explícita en la norma, se considera importante que se realicen antes de comenzar la Construcción de software. El objetivo de las tareas será descrito más adelante.

En cuanto al material adicional de soporte que consiste en plantillas, ejemplos, listas de verificación y herramientas, el paquete actual cuenta con muy poca información al respecto. Su aportación se concentra en la sección 4 correspondiente a la *Descripción de Procesos, Actividades, Tareas, Pasos, Roles y Productos.*

El resto de la estructura del paquete aunque igual de importante sigue las convenciones tradicionales de los paquetes; descripción técnica, definiciones, referencias a otros estándares y modelos, referencias y un formulario de evaluación.

Para definir que modificaciones fueron necesarias, cada uno de los elementos de la estructura del paquete actual serán analizados y se hará un comentario al respecto, el cual ayudará a decidir si es necesario realizar modificaciones, agregaciones o eliminaciones a la parte correspondiente.

3.1.2 Fortalezas

El paquete actual cuenta con las siguientes fortalezas:

- ✓ Las tareas del paquete están acorde a lo establecido en la norma ISO/IEC 29110 parte 5
- ✓ Los pasos de las tareas son sencillos y cortos

[4] Sitio Público del ISO/IEC JTC1/SC7 Grupo de trabajo WG24: http://profs.etsmtl.ca/claporte/English/VSE/index.html

- ✓ El paquete es muy ligero
- ✓ Las referencias del paquete con otros estándares y modelos se encuentran bien definidas

3.1.3 Debilidades

- ✗ Los pasos de la tarea de Diseño de casos de pruebas unitarias son muy generales y carecen de detalle suficiente para una mejor comprensión y correcta aplicación.
- ✗ Algunas actividades omiten u obvian ciertos pasos que pueden variar el resultado esperado si no se describen oportunamente.
- ✗ Carece de plantillas específicas
- ✗ Carece de ejemplos que sirvan de soporte para la comprensión de las tareas y actividades tratadas en el paquete.
- ✗ El único ejemplo que incluye es de la generación de casos de prueba a partir de un cierto código. Sin embargo los casos obtenidos son un poco ambiguos y no se cita ningún criterio de salida para dejar de producirlos. Además el ejemplo se encuentra en la sección de herramientas en vez de la sección de ejemplos.
- ✗ Faltan listas de verificación que permitan mapear la realización de cada una de las tareas mencionadas en el paquete.
- ✗ La sección de herramientas cuenta con sólo una (Matriz de trazabilidad). Debido al número y al tipo tareas se esperaba que se mencionara al menos un par de herramientas más.

3.1.4 Análisis de la estructura del paquete y cambios propuestos para la nueva versión

A continuación se mencionan cada una de las partes del paquete, una descripción breve acerca de su objetivo o propósito y una tabla con los detalles del cambio propuesto para la nueva versión, en caso de que hubiera alguno.

El significado de los tipos de cambios propuestos es el que se muestra en la Tabla 5.

Tipo de cambio	***Significado***
Ninguno	No se realiza ningún cambio a la sección correspondiente
Modificación Parcial	Sólo una parte de la sección es modificada dejando el resto como se encuentra actualmente en la versión 0.4
Modificación Total	El título de la sección o tarea se deja como en la versión 0.4 pero el contenido, cuerpo y estructura es modificada
Agregar información	Se añade contenido a la sección correspondiente, dejando el resto como viene actualmente en la versión 0.4
Eliminación Parcial	Una parte de la sección o tarea es eliminada o bien es eliminada pero se hace referencia a ella en alguna otra parte.
Eliminación Total	Una sección o tarea es eliminada definitivamente

Tabla 5 Tipos de cambios propuestos al paquete y su significado

1. Descripción Técnica

Propósito del documento

Se define formalmente qué es un Paquete de Puesta en Operación, su origen, objetivo y a quién va dirigido.

Cambio propuesto	**Ninguno**
Justificación	Cumple con su objetivo
Propuesta de cambio	---

¿Por qué es importante la construcción de software y pruebas unitarias?

En esta parte se explica la importancia del tema tratado a lo largo del paquete, mostrando referencias y estadísticas para ejercer un mayor poder de convencimiento sobre el lector.

Cambio propuesto	**Modificación Parcial**
Justificación	Falta de datos estadísticos para hacer notar la importancia de la construcción de software y las pruebas unitarias.
Propuesta de cambio	Listar algunas otras razones para hacer notar la importancia de la etapa de Construcción y mostrar algún dato estadístico que resalte la importancia de las pruebas unitarias.

2. Definiciones

Esta sección representa un pequeño glosario con las definiciones de términos genéricos (aplicados a todos los paquetes) y específicos (propios del contenido del paquete en cuestión)

Cambio propuesto	**Agregar información**
Justificación	Falta definir algunos términos específicos
Propuesta de cambio	Añadir algunas definiciones a los términos específicos, definiciones como: *Cobertura de código* podrían ser un buen ejemplo.

3. Su relación con ISO/IEC 29110

En este punto, se muestran una lista de procesos de Implementación de Software, actividades, tareas y roles de la parte 5 que están directamente relacionados con el paquete en cuestión.

Cambio propuesto	**Ninguno**
Justificación	Cumple con su objetivo
Propuesta de cambio	---

4. Descripción de Procesos, Actividades, Tareas, Pasos, Roles y Productos

Aquí se describen de manera estructurada cada una de las tareas correspondientes a la actividad de Construcción y pruebas unitarias. Se define el objetivo de cada tarea, su justificación, los roles involucrados, el estado de los productos, artefactos al final de la tarea y la descripción de los pasos para llevarla a cabo.

Sub-tarea: Seleccionar el estándar de interfaz de usuario

El objetivo de esta sub-tarea es seleccionar un estándar de interfaz de usuario, ya sea del repositorio de proyectos anteriores o bien de alguno propuesto por el mismo cliente.

Cambio propuesto	**Eliminación Parcial** (Cambio de lugar dentro del paquete)
Justificación	Esta sub-tarea debe de ser realizada antes de la fase de construcción.
Propuesta de cambio	Sólo para asegurarse de que se cuenta con un estándar de interfaz de usuario antes de entrar a la etapa de Construcción. Se propone agregar una lista de verificación para revisar la realización de ésta sub-tarea.

Sub-tarea: Definir estándares de construcción

El objetivo de esta tarea es servir de guía en la codificación de software para producir código fácil de mantener dentro o fuera del proyecto.

Cambio propuesto	**Ninguno**
Justificación	Cumple con su objetivo
Propuesta de cambio	---

Tarea: Asignar tareas a los miembros del equipo de trabajo

El objetivo de esta tarea es definir la secuencia de construcción y asignar tareas (de acuerdo a un rol) a los miembros del equipo de trabajo

Cambio propuesto	**Modificación Parcial**
Justificación	• Falta mayor detalle en la explicación de las estrategias de integración mencionadas. • No se menciona ningún criterio a considerar para la asignación eficiente de tareas a los miembros del equipo de trabajo
Propuesta de cambio	• Explicar y detallar otras estrategias de integración no tan radicales • Definir un criterio de salida para las pruebas unitarias • Mencionar criterios importantes que se deben tomar en cuenta para la asignación de tareas a los programadores

Tarea: Construir o actualizar componentes de software

El objetivo de esta tarea es producir componentes de software como fueron diseñados

Cambio propuesto	**Modificación Parcial**
Justificación	• Los pasos propuestos para cumplir la tarea son demasiado generales. • Sólo menciona un enfoque para la construcción de componentes.
Propuesta de cambio	• Dividir los pasos actuales en sub-tareas que atiendan a necesidades más específicas. • Explicar algún otro enfoque para la construcción sistemática de componentes

Tarea: Diseñar o actualizar casos de pruebas unitarias y aplicarlos

El objetivo de esta tarea es encontrar y corregir defectos introducidos durante la construcción a través del diseño y aplicación de casos de pruebas unitarias.

Cambio propuesto	**Modificación Total**
Justificación	• Sólo se menciona una estrategia para la generación de casos de prueba y su explicación es demasiado escueta. • La descripción de los pasos para la creación de los casos de prueba tiene muy poco detalle y hace referencia a un ejemplo un tanto ambiguo. • No se menciona en ninguna parte los pasos para crear y ejecutar las pruebas unitarias en base a los casos de prueba generados • La definición de un criterio de salida para las pruebas unitarias no se menciona de manera explícita en ninguno de los pasos.
Propuesta de cambio	• Explicar algunos criterios de salida para la creación de casos de pruebas unitarias de caja blanca, especialmente los correspondientes a *cobertura de código* • Describir la estructura que deben de tener los casos de prueba • Mencionar algunas estrategias para la codificación de las pruebas unitarias basadas en los casos de prueba, usando un marco de trabajo interno y externo. • Mencionar algunas estrategias para la ejecución de las pruebas unitarias ya sea usando herramientas especializadas o bien de manera manual. • Recomendar el uso de herramientas de cobertura de código para verificar que se cumpla el criterio de salida seleccionado.

Tarea: Corregir los defectos

El objetivo de esta tarea es corregir los defectos encontrados por las pruebas unitarias.

Cambio propuesto	**Agregar información**
Justificación	• Una vez detectado el defecto es necesario que se mencionen algunas

	ideas o prácticas para localizarlo dentro del código de la implementación.
Propuesta de cambio	• Mencionar algunas prácticas para localizar los defectos, como el uso de listas de revisión de código, uso del depurador y reproducción de errores.

Tarea: Actualizar el registro de trazabilidad

El objetivo de esta tarea es asegurarse que todos los componentes de software puedan ser trazados a un elemento de diseño una vez que sean construidos.

Cambio propuesto	**Ninguno**
Justificación	Cumple con su objetivo
Propuesta de cambio	---

Descripción de roles

En esta sección se presenta una lista alfabética de los roles involucrados en el paquete. Se muestra el acrónimo correspondiente y la lista de competencias como se encuentran definidas en el ISO 29110 Parte 5-1-2.

Cambio propuesto	**Agregar información**
Justificación	• Se descubrió que este rol es necesario para la realización de las tareas: o Asignar tareas a los miembros del equipo de trabajo o Diseñar o actualizar casos de pruebas unitarias y aplicarlos • Y de las sub-tarea: o Definir estándares de construcción
Propuesta de cambio	Añadir el rol a la lista de roles, junto con su acrónimo y sus competencias tal y como son definidas en el ISO 29110 Parte 5-1-2.

Descripción de productos

En esta parte se describe cada uno de los productos relacionados con la actividad de Construcción y pruebas unitarias, así como la fuente o el proceso en donde se generan.

Cambio propuesto	**Ninguno**
Justificación	Cumple con su objetivo
Propuesta de cambio	---

Descripción de artefactos

En esta sección se describen cada uno de los artefactos que pueden ser producidos para facilitar algunas tareas de la actividad de Construcción y pruebas unitarias.

Cambio propuesto	**Modificación Parcial**
Justificación	Con la propuesta de modificación de algunas tareas y sub-tareas surgen algunos nuevos artefactos que facilitan su realización y con la eliminación de algunas sub-tareas también se sugiere la eliminación de los artefactos correspondientes.
Propuesta de cambio	Añadir a la lista los siguientes artefactos: • Conjunto de pruebas unitarias • Taxonomía de defectos Eliminar de la lista los siguientes artefactos: • Especificación de interfaz de usuario

5. **Plantillas**

En esta parte se añaden aquellas plantillas que puedan servir de guía o apoyo para la realización de algunas tareas del paquete.

Cambio propuesto	**Modificación Total**
Justificación	En el paquete actual se muestran enlaces a diferentes sitios web que contienen plantillas de codificación para diferentes lenguajes. Sin embargo existe el riesgo de que dichos enlaces se rompan o que el servidor en el que se encuentran sea dado de bajo o cambie de dominio.
Propuesta de cambio	Añadir una plantilla de codificación para un lenguaje en específico *(lenguaje tentativo: Java)*

6. **Ejemplos**

En esta parte se agregan ejemplos de las tareas que requieran una mayor explicación o aclaración de ciertos pasos. Los ejemplos tratan de ser lo más sencillos y cortos posibles para facilitar su comprensión.

Cambio propuesto	**Agregar información**
Justificación	No existe ningún ejemplo en el paquete actual
Propuesta de cambio	Se pretende agregar ejemplos sencillos y cortos para facilitar su comprensión. Seguramente existen ejemplos más detallados y extensos en la literatura o en otras fuentes, pero quedarán fuera del alance y de objetivo de este paquete. Los ejemplos que se sugieren son los siguientes: • Ejemplo de un estándar genérico de construcción: • Ejemplo de Pseudocódigo

	• Ejemplo de Diagrama de Flujo • Jerarquía de criterios de cobertura, del más débil al más fuerte • Diseño de casos de prueba, escritura de pruebas unitarias y uso de una herramienta de cobertura de código • Pruebas unitarias siguiendo la técnica de Pruebas estructuradas • Corregir los defectos • Ejemplos de Test runners • Ejemplo de una macro para aserciones • Ciclo de vida de las pruebas unitarias

7. Listas de verificación

En esta sección se añaden listas de verificación que pudieran ser de utilidad o apoyo para la completitud de ciertas tareas. Cada lista pretende a través de una serie de pasos asegurar el cumplimiento de los objetivos relacionados con alguna tarea del paquete.

Cambio propuesto	**Agregar información**
Justificación	No existen listas de verificación que permitan el cumplimiento de cada uno de los pasos descritos en las tareas del paquete.
Propuesta de cambio	• Agregar una lista de verificación para cada una de las tareas y actividades del paquete. • Agregar listas de verificación con las especificaciones que el arquitecto y diseñador deben proveer antes de empezar la fase de construcción.

8. Herramientas

En este apartado se mencionan las herramientas que pueden ser de utilidad para la realización de ciertas tareas y la fuente de donde se pueden conseguir y obtener más información al respecto.

Cambio propuesto	**Agregar información**
Justificación	Falta de herramientas de soporte
Propuesta de cambio	• Mencionar la fuente y las herramientas más populares de cobertura de código para ciertos lenguajes. • Mencionar la fuente y los frameworks más populares para pruebas unitarias para ciertos lenguajes.

9. Referencias a otros estándares y modelos

Esta sección proporciona referencias de éste Paquete de Puesta en Operación a una selección de estándares ISO e ISO/IEC y a la Integración de Modelos de Madurez de Capacidades.

Cambio propuesto	**Ninguno**
Justificación	Cumple con su objetivo

Propuesta de cambio	---

10. Referencias

Proporciona referencias a la bibliografía y otras fuentes usadas como medio de investigación para la planeación y realización del paquete.

Cambio propuesto	**Agregar información**
Justificación	Se añadirá nuevo material al paquete que proviene de fuentes no mencionadas en el paquete actual.
Propuesta de cambio	Añadir las nuevas fuentes a la sección de Referencias.

11. Formulario de evaluación

Representa una forma de evaluación del paquete. Las respuestas obtenidas de los lectores sirven como retroalimentación para mejorar las partes débiles del paquete.

Cambio propuesto	**Ninguno**
Justificación	Cumple con su objetivo
Propuesta de cambio	---

4. PPO de Construcción y pruebas unitarias (Versión 0.5)

4.1.1 Perspectiva general

La propuesta para la nueva versión del Paquete de Puesta en Operación de Construcción y pruebas unitarias fue redactada en inglés con el nombre original de *"Deployment Package, Construction and Unit Testing, Basic Profile" (versión 0.5).*

Esta nueva versión fue elaborada tomando como base la versión anterior (Versión 0.4) y realizando las modificaciones pertinentes bajo el asesoramiento de su autora, Ana Vázquez, miembro del grupo WG24.

El paquete fue escrito en el laboratorio A-3440 Logiciel TI de la École de Technologie Supérieure, Montreal, Quebec, Canadá. Con el apoyo del Dr. Roger Champagne, Profesor asociado del departamento de Ingeniería de Software y profesor de la materia de Pruebas unitarias de la misma universidad.

El documento fue aprobado por el editor en jefe del grupo WG24 del ISO/IEC JTC1/SC7 el Dr. Claude Y. Laporte. Adhiriéndose así como un paquete más al conjunto de paquetes que sirven de apoyo para al cumplimiento de la norma ISO/IEC 29110 Perfil Básico.

El documento fue traducido al idioma Español en la Universidad Nacional Autónoma de México por el mismo autor de la versión en inglés (Autor de la presente tesis) bajo el nombre de: "Paquete de Puesta en Operación, Construcción y Pruebas Unitarias, Perfil Básico". A partir de ahora y en el resto del documento hará referencia a la versión en español de dicho documento (Ver anexo 1).

El objetivo principal del nuevo paquete es trabajar en las debilidades del paquete anterior y proponer una solución para que la nueva versión sea más robusta. Los detalles de los cambios más importantes serán mencionados más adelante.

4.1.2 Cambios realizados a la estructura de la versión 0.4

Antes de comenzar a explicar en detalle los cambios realizados a la versión 0.4, en la Tabla 6 se muestra un resumen de los cambios realizados a la estructura del paquete.

El texto sin colorear (en color negro por default) indica las partes que no sufrieron ninguna modificación o alteración en su estructura, contenido o título. El texto coloreado en ***rojo*** indica que dicha parte o sección fue eliminada, el texto en ***verde*** indica que fue modificada y el texto en ***azul*** indica que es una parte nueva agregada al paquete.

VERSIÓN 0.4	VERSIÓN 0.5
Descripción Técnica • Propósito de éste documento • ¿Por qué es importante la construcción de software y pruebas unitarias?	**Descripción Técnica** • Propósito de éste documento • ¿Por qué es importante la construcción de software y pruebas unitarias?

Definiciones • Términos genéricos • Términos específicos	**Definiciones** • Términos genéricos • Términos específicos
Su relación con ISO/IEC 29110	**Su relación con ISO/IEC 29110**
Descripción de Procesos, Actividades, Tareas, Pasos, Roles y Productos ○ Sub-tarea: Seleccionar el estándar de interfaz de usuario. ○ Sub-tarea: Definir el estándar de construcción. • Asignar tareas a los miembros del equipo de trabajo • Construir o actualizar componentes de software • Diseñar o actualizar casos de pruebas unitarias y aplicarlos • Corregir los defectos • Actualizar el registro de trazabilidad • Descripción de los roles • Descripción de los productos • Descripción de los artefactos	**Descripción de Procesos, Actividades, Tareas, Pasos, Roles y Productos** ○ ~~Sub-tarea: Seleccionar el estándar de interfaz de usuario.~~ ○ Sub-tarea: Definir el estándar de construcción ○ Sub-tarea: Reportar taxonomía de defectos de etapas previas • Asignar tareas a los miembros del equipo de trabajo • Construir o actualizar componentes de software • Diseñar o actualizar casos de pruebas unitarias y aplicarlos • Corregir los defectos • Actualizar el registro de trazabilidad • Descripción de los roles • Descripción de los productos • Descripción de los artefactos
Plantillas • Vínculos a plantillas de estándares de codificación • Vínculos a plantillas de especificación de interfaz de usuario	**Plantillas** • ~~Vínculos a plantillas de estándares de codificación~~ • ~~Vínculos a plantillas de especificación de interfaz de usuario~~ • Plantilla de construcción para Java
Ejemplos • Ejemplo de desarrollo de casos de pruebas usando Pruebas Estructuradas para un programa en Java. (Este ejemplo viene en la sección de herramientas, aunque si lugar corresponde a ésta sección).	**Ejemplos** • ~~Ejemplo de desarrollo de casos de pruebas usando Pruebas Estructuradas para un programa en Java~~ • Ejemplo de un estándar genérico de construcción • Ejemplo de Pseudocódigo • Ejemplo de Diagrama de Flujo • Jerarquía de criterios de cobertura, del más débil al más fuerte • Diseño de casos de prueba, escritura de pruebas unitarias y uso de una herramienta de cobertura de código • Pruebas unitarias siguiendo la técnica de Pruebas Estructuradas y mapeándola a los pasos de la tarea “Diseñar o actualizar casos de pruebas

<table>
<tr><td></td><td>unitarias y aplicarlos”
Corregir los defectos
Ejemplos de Test runners
Ejemplo de una macro para aserciones en C++
Ciclo de vida de las pruebas unitarias</td></tr>
<tr><td>Listas de verificación
Revisión de código</td><td>Listas de verificación:

Listas de verificación de tareas
Asignar tareas a los miembros del equipo de trabajo
Construir o actualizar componentes de software
Diseñar o actualizar casos de pruebas unitarias y aplicarlos
Corregir los defectos
Listas de verificación de soporte
Revisión de código
Lo que el arquitecto y el diseñador deben proveer</td></tr>
<tr><td>Herramientas
Matriz de trazabilidad</td><td>Herramientas
Matriz de trazabilidad
Herramientas de cobertura de código
Marcos de trabajo para pruebas unitarias (Frameworks)</td></tr>
<tr><td>Referencias a otros Estándares y Modelos
ISO 9001 Matriz de referencia
ISO/IEC 12207 Matriz de referencia
CMMI Matriz de referencia</td><td>Referencias a otros Estándares y Modelos
ISO 9001 Matriz de referencia
ISO/IEC 12207 Matriz de referencia
CMMI Matriz de referencia</td></tr>
<tr><td>Referencias</td><td>Referencias</td></tr>
<tr><td>Formulario de evaluación</td><td>Formulario de evaluación</td></tr>
</table>

Tabla 6 Estructura de la versión 0.4 y 0.5 del Paquete de Puesta en Operación de Construcción y Pruebas unitarias

4.1.3 Aportaciones más importantes de la versión 0.5 y algunas comparativas entre ambas versiones.

Enseguida se muestra el detalle de los cambios más relevantes realizados al paquete, su aportación, y en donde se requiera una comparativa entre la estructura definida en la versión anterior y en la nueva.

Descripción Técnica

- **¿Por qué es importante la construcción de software y pruebas unitarias?**

Aportaciones más relevantes:

Para hacer un mayor hincapié en la importancia de la Construcción de software y pruebas unitarias se añadieron algunas razones adicionales y algunas estadísticas que revelan el costo de la corrección de defectos a través de las diferentes etapas en el ciclo de desarrollo de software.

Definiciones

- **Términos específicos**

Aportaciones más relevantes:

Se añade el término ***Cobertura de código*** al glosario de términos específicos. Esto con la finalidad de que los conceptos explicados en la sección de *Diseño de casos de pruebas unitarias* queden bien comprendidos y dentro del domino del problema.

Descripción de Procesos, Actividades, Tareas, Pasos, Roles y Productos

- ~~Sub-tarea: Seleccionar el estándar de interfaz de usuario~~

Aportaciones más relevantes:

Debido a que para la etapa de Construcción esta tarea ya debería estar realizada, se decidió eliminarla de ésta sección. Sin embargo como medida preventiva, se decidió añadir una lista de verificación que permita revisar si la tarea fue realizada de manera previa antes de empezar la Construcción.

La lista de verificación fue colocada en la sección 7 correspondiente a *Listas de verificación* como una sub-lista de la lista 7.6 *Lo que el arquitecto y el diseñador deben proveer.*

Estándar de interfaz de usuario

☐	¿Cuentas ya con algún estándar de interfaz?
☐	¿Has seleccionado ya un estándar de interfaz?
☐	¿Cuentas ya con la aprobación del cliente?
☐	¿Tus programadores ya han adoptado el estándar?
☐	¿Has verificado la adopción del estándar?

- **Sub-tarea: Reportar taxonomía de defectos de etapas previas**

Aportaciones más relevantes:

Esta sub-tarea no se encuentra en la versión 0.4 y su objetivo es reportar defectos de etapas previas en el ciclo de desarrollo al área que se encuentre a cargo. Esto con la finalidad de evitar la propagación de defectos que con el tiempo se pueden volver costosos. La sub-tarea no sólo puede

aplicarse a las actividades de construcción y pruebas unitarias sino que puede usarse en otros paquetes con el mismo objetivo.

Además la taxonomía puede servir en un futuro como un registro de los defectos que se han suscitado en los proyectos y proponer técnicas de prevención para evitar caer en los mismos errores.

La sub-tarea está dirigida a dos roles, el Programador y el Líder Técnico. Cada uno tiene que realizar una serie de pasos antes de que la tarea se realice completamente.

Programador:

1. Confirmar el defecto
2. Escribir una descripción breve del defecto
3. Escribir dónde fue encontrado el defecto
4. Escribir las posibles causas
5. Escribir la extensión del daño

Líder técnico:

1. Verificar el reporte del defecto
2. Escribir una estrategia de mitigación
3. Reportar el defecto al área encargada

- **Tarea: Asignar tareas a los miembros del equipo de trabajo**

Aportaciones más relevantes:

Los cambios más significativos a esta tarea fueron realizados en los pasos 2 y 4 respectivamente. Los pasos 1 y 3 quedaron sin cambio alguno y se añadió un quinto paso a la tarea.

Modificaciones al paso 2: Seleccionar la estrategia de integración, se propusieron otras dos estrategias de integración, las cuales son estrategias híbridas que pretenden no ser tan radicales a la hora de integrar componentes:

- *La estrategia de Integración orientada a riesgos*
- *La estrategia de Integración orientada a características*.

También se eliminó la estrategia *Class Test Order* debido a que es un poco más compleja de llevar a cabo y su explicación requiere más detalle.

Modificaciones al paso 4: Asignar tareas a los miembros del equipo de trabajo, se mencionaron algunos aspectos adicionales sobre los programadores que se deben tomar en cuenta para tener una asignación de tareas más efectiva, tales como:

- Experiencia

- Habilidades
- Conocimiento

Se añade el paso 5: Definir un criterio de salida para las pruebas unitarias, en este paso se describe la importancia de seleccionar un criterio de salida para que los programadores sepan cuando dejar de crear casos de prueba. Se propone una clasificación de componentes de acuerdo al riesgo que involucran y un criterio de cobertura recomendado dependiendo del riesgo asociado. Esta comparativa se aprecia mejor en la Tabla 7.

Versión 0.4	*Versión 0.5*
1. Obtener la documentación del diseño de software 2. Seleccionar la estrategia de integración 3. Detallar el calendario del proyecto 4. Asignar tareas a los miembros del equipo de trabajo	1. Obtener la documentación del diseño de software 2. Seleccionar la estrategia de integración 3. Detallar el calendario del proyecto 4. Asignar tareas a los miembros del equipo de trabajo 5. Definir un criterio de salida para las pruebas unitarias

Tabla 7. Comparativa de la tarea "Asignar tareas a los miembros del equipo de trabajo" entre las versiones 0.4 y 0.5

- **Tarea: Construir o actualizar componentes de software**

Aportaciones más relevantes:

Los cambios que se realizaron a esta tarea fueron básicamente estructurales. Inicialmente el paquete contenía 3 pasos que incluían una gran cantidad de sub-tareas heterogéneas que no necesariamente pertenecían a alguno de los tres pasos. Por ello se decidió dividir dichas sub-tareas en 5 pasos que las involucren de manera más apropiada. Además en los nuevos pasos se hace referencia al uso de estándares de construcción y se propone otro enfoque además del pseudocódigo para definir la lógica del componente, el Diagrama de flujo. Esta comparativa se aprecia mejor en la Tabla 8.

Versión 0.4	*Versión 0.5*
1. Diseñar el componente a. Verificar la contribución del componente b. Definir el problema que el componente va a resolver c. Buscar funcionalidades disponibles en las librerías estándar. d. Escribir el pseudocódigo e. Diseñar los datos del componente f. Revisar el pseudocódigo	1. Entender el diseño detallado del componente y su contribución. a. Verificar la contribución b. Detalle suficiente 2. Buscar funcionalidades disponibles en las librerías estándar. 3. Definir la lógica del componente a. Proceso de programación por Pseudocódigo b. Diagrama de flujo 4. Codificar el componente de acuerdo al

2. Codificar el componente a. Escribir la declaración del componente b. Convertir el pseudocódigo en comentarios de alto nivel c. Escribir comentarios en el código d. Revisar si el código necesita ser dividido 3. Verificar el componente a. Revisión de código b. Compilar el código c. Pasar el código en el depurador	estándar de construcción a. Verificar si el código necesita ser dividido 5. Verificar el componente a. Compilar el código

Tabla 8 Comparativa de la tarea "Construir o actualizar componentes de software" en las versiones 0.4 y 0.5

- **Tarea: Diseñar o actualizar casos de pruebas unitarias y aplicarlos**

Aportaciones más relevantes:

Esta tarea fue modificada completamente, dejando únicamente el título de la tarea de la versión anterior.

El alcance de la tarea en la versión 0.4 era llegar hasta el diseño de los casos de prueba, sin llegar a la creación de las pruebas unitarias y mucho menos a su aplicación. Debido al alcance tan limitado que éste presentaba se decidió ampliar el panorama para cubrir con mayor detalle estas actividades que representan una parte importante del paquete.

La nueva estructura de la tarea en su primer paso **Obtener el criterio de salida** pretende que el programador conozca algunos de los criterios de cobertura de código más conocidos para que sirvan como condiciones de paro a la hora de crear los casos de prueba. Cabe mencionar que aunado a esto el Administrador de proyecto escogerá un porcentaje para el criterio seleccionado que delimitará aun más el número de casos requeridos para cada componente.

Cada criterio tiene cierto nivel de dificultad y permite probar ciertas partes del código, mientras más esfuerzo requiera el criterio mejor probados resultan los componentes.

El siguiente paso consiste en **Diseñar los casos de prueba** de acuerdo a un cierto formato (Myers, 2004), explicando brevemente en qué consiste cada campo y qué tipo de información le corresponde.

Una vez diseñados los casos de prueba lo siguiente es **Codificar las pruebas unitarias** basadas en dichos casos. Para ello en este paso se explican algunos marcos de trabajo de pruebas unitarias que facilitan este proceso, de igual manera se explica brevemente como crear un marco de trabajo de manera manual para el caso en el que no exista un marco para el lenguaje que se esté usando.

El paso número 4 consiste en explicar las estrategias para **Ejecutar las pruebas unitarias** las cuales pueden ser manuales o bien utilizando alguna de las herramientas que ofrecen los marcos de trabajo.

El último paso se enfoca a **Analizar los resultados**, en caso de haber errores en la implementación hacer las correcciones pertinentes y en caso contrario determinar si se ha cumplido con el criterio establecido por el Administrador del proyecto. Esta comparativa se aprecia mejor en la Tabla 9.

Versión 0.4	***Versión 0.5***
1. Definir el número de casos de prueba necesarios 2. Definir los casos de prueba 3. Aplicar los casos de prueba	1. Obtener el criterio de salida a. Criterios de cobertura i. Cobertura de Sentencia ii. Cobertura de Decisión iii. Cobertura de Decisión-Condición b. Porcentaje para el criterio de cobertura seleccionado 2. Diseñar los casos de prueba 3. Codificar las pruebas unitarias a. Sin un framework de pruebas unitarias externo b. Con un framework de pruebas unitarias externo 4. Ejecutar las pruebas unitarias a. Sin un framework de pruebas unitarias externo b. Con un framework de pruebas unitarias externo i. Test runner de línea de comandos ii. Test runner gráfico 5. Analizar los resultados

Tabla 9 Comparativa de la tarea "Diseñar o actualizar casos de pruebas unitarias y aplicarlos" en las versiones 0.4 y 0.5

- **Tarea: Corregir los defectos**

Aportaciones más relevantes:

El principal cambio realizado a esta tarea es la adición del paso 2 **Determinar la naturaleza y ubicación del defecto** en el cual se proporcionan prácticas para localizar los defectos dentro del código de la implementación. Recordemos que una cosa es detectar un defecto y otra muy diferente es localizar la ubicación exacta y la fuente del problema.

Entre las prácticas de detección de defectos que se mencionan se encuentran:

- Listas de revisión de código
- Uso del depurador
- Reproducción de errores por diferentes medios

Por último se cambio el nombre del último paso al de **Verificar las correcciones** ya que su nombre original de **Aplicar los casos de pruebas (pruebas de regresión)** hace alusión a la ejecución de casos de prueba, cuando en realidad lo que se ejecutan son las pruebas unitarias. Otro posible nombre para este paso podría ser **Aplicar las pruebas unitarias.** Sin embargo se decidió dejar el primer nombre debido a que el objetivo principal de éste paso es precisamente verificar que las correcciones trabajen adecuadamente. Esta comparativa se aprecia mejor en la Tabla 10.

Versión 0.4	***Versión 0.5***
1. Confirmar el defecto 2. Corregir el defecto 3. Aplicar los casos de prueba (pruebas de regresión)	1. Confirmar el defecto 2. Determinar la naturaleza y ubicación del defecto 3. Corregir el defecto 4. Verificar las correcciones

Tabla 10 Comparativa de la tarea "Corregir defectos" en las versiones 0.4 y 0.5

- **Descripción de roles**

Aportaciones más relevantes:

Se añadió el rol del administrador del proyecto debido a que se consideró su participación en las tareas de **Asignar tareas a los miembros del equipo de trabajo** y a la tarea de **Diseñar o actualizar casos de pruebas unitarias y aplicarlos.**

Se agregó también su acrónimo y sus competencias:

Administrador del proyecto	AP	Capacidad de liderazgo con experiencia en la toma de decisiones, planeación, administración de personal, delegación y supervisión, finanzas y desarrollo de software.

- **Descripción de artefactos**

Aportaciones más relevantes:

Se hicieron algunas modificaciones a la lista de artefactos:

1. Se eliminó la especificación de interfaz de usuario, ya que desde que la sub-tarea: *seleccionar el estándar de interfaz de usuario* fue eliminada del paquete, dicho artefacto ya no es necesario
2. Se añade el *Conjunto de pruebas unitarias* como un artefacto útil para realizar en futuro pruebas de regresión.

3. Se añade la *Taxonomía de defectos* como un artefacto de la tarea: *Reportar taxonomía de defectos de etapas previas*, y que sirve como registro de los diferentes tipos de defectos encontrados en etapas previas a la fase de construcción.

~~3.~~	~~User Interface Specification~~	~~Provides specification for:~~ ~~- Windows~~ ~~- Menu Bar~~ ~~- Dialogs~~ ~~- Messages~~ ~~- Business Rules~~ ~~among others.~~
3.	Conjunto de pruebas unitarias	Conjunto de rutinas escritas en un lenguaje específico diseñadas para probar los casos de prueba
4.	Taxonomía de defectos	Es un método que reduce el número de defectos en el producto aprendiendo de los tipos de errores que se comenten en el proceso de desarrollo, de tal manera que puedan ser analizados para mejorar el proceso de reducir o eliminar la probabilidad de que se cometa el mismo tipo de error en el futuro.

Plantillas

- ~~Vínculos a plantillas de estándares de codificación~~
- ~~Vínculos a plantillas de especificación de interfaz de usuario~~

Aportaciones más relevantes:

En esta sección se mostraban enlaces a sitios web con plantillas de estándares de codificación para diferentes lenguajes, así como un enlace a una plantilla para la especificación de interfaz de usuario.

El motivo por el cual se eliminaron dichos enlaces, es debido a que los sitios en donde se encuentran los documentos son ajenos al paquete, y existe el riesgo de que dejen de estar disponibles si son eliminados o cambiados de servidor.

- **Plantilla de construcción para Java**

Aportaciones más relevantes:

En vez de mostrar enlaces a plantillas de codificación externas, se decidió agregar una plantilla de construcción para un lenguaje en específico, en este caso se optó por Java al ser uno de los lenguajes más usados al momento.

La plantilla cuenta con las siguientes características:

- Divisiones para separar el componente en agrupaciones lógicas
- Formato para control de versiones
- División para resumen del componente
- Formato para las sentencias de importación
- División para el objetivo de la case
- División para la descripción de métodos, excepciones y valores de retorno
- Formato para constructores
- Formato para métodos públicos, privados y protegidos
- Formato para atributos privados y protegidos.

Entre otras características.

- **Plantilla de construcción para Java**

Aportaciones más relevantes:

En vez de mostrar enlaces a plantillas de codificación externas, se decidió agregar una plantilla de construcción para un lenguaje en específico, en este caso se optó por Java al ser uno de los lenguajes más usados al momento.

Ejemplos

- ~~Ejemplo de desarrollo de casos de pruebas usando Pruebas Estructuradas para un programa en Java~~

Aportaciones más relevantes:

Los motivos de la eliminación de éste ejemplo son los siguientes:

- No se explican los pasos de la técnica de pruebas estructuradas, por lo que no existe ninguna guía a lo largo del ejemplo.
- En algunas partes del código la semántica no es muy clara
- La interpretación de los casos de prueba obtenidos no es muy evidente

Además el ejemplo se encuentra mal ubicado al estar en la sección de herramientas.

- **Ejemplo de un estándar genérico de construcción**

Aportaciones más relevantes:

Sabiendo que no tener un orden en la construcción de software puede causar productos con mala calidad, poco estandarizados y con defectos por cuestiones de formato. Se decidió agregar un estándar de construcción genérico (aplicables a casi cualquier lenguaje de programación) que contiene un conjunto de lineamientos y consejos para estandarizar la construcción de los

componentes de software, con la finalidad de reducir los tiempos de construcción, facilitar el mantenimiento, la búsqueda de defectos y en general producir un componente de mejor calidad.

Pretende estandarizar los siguientes elementos:

- Elementos de formato
- Comentarios
- Módulos y archivos
- Variables
- Constantes
- Expresiones y sentencias
- Estructuras de control
- Funciones

Si se quiere ser más riguroso se puede usar el mismo estándar como una lista de verificación y revisar el cumplimiento de cada lineamiento.

- **Ejemplo de Pseudocódigo**
- **Ejemplo de Diagrama de Flujo**

Aportaciones más relevantes:

Con el objetivo de hacer más ilustrativos los enfoques para la generación de componentes mencionados en las tareas del paquete, se decidió agregar un ejemplo sencillo para cada enfoque respectivamente.

- **Jerarquía de criterios de cobertura, del más débil al más fuerte**

Aportaciones más relevantes:

Con el objetivo de comprender mejor los criterios mencionados en la tarea de *Diseñar o actualizar casos de pruebas unitarias y aplicarlos,* y para que la selección del criterio no se deje al azar, se decidió tomar como base un código sencillo de ejemplo y obtener los casos de prueba para satisfacer cada uno de los siguientes criterios:

- Criterio de cobertura de sentencia
- Criterio de cobertura de Decisión/Brinco
- Criterio de cobertura de Decisión-Condición

Los criterios son listados del más débil al más fuerte, mostrando sus bondades y desventajas en su uso práctico. Además se dan algunos consejos importantes para facilitar la creación de los casos de prueba del criterio de Decisión-Condición.

- **Diseño de casos de prueba, escritura de pruebas unitarias y uso de una herramienta de cobertura de código**

Aportaciones más relevantes:

Para poder ejemplificar la transición de los casos de prueba a las pruebas unitarias, se añadió este ejemplo, el cual no sólo muestra nuevamente la obtención de casos de prueba para satisfacer los criterios mencionados anteriormente a partir del código de una rutina, sino también las pruebas unitarias codificadas en un lenguaje y marco de trabajo específico. Además para hacer más completo el ejemplo también se muestran capturas de pantalla de una herramienta de cobertura de código para verificar que efectivamente se cubra el criterio definido.

El ejemplo tiene las siguientes características:

- Casos de prueba para satisfacer los criterios de Sentencia, Decisión y Decisión-Condición de un código dado.
- Código Java de las pruebas unitarias usando el framework Junit.
- Resultados de la herramienta de cobertura de código Codecover obtenidos a partir de las pruebas unitarias.

- **Pruebas unitarias siguiendo la técnica de Pruebas Estructuradas y mapeándola a los pasos de la tarea "Diseñar o actualizar casos de pruebas unitarias y aplicarlos"**

Aportaciones más relevantes:

La versión 0.4 contenía un ejemplo de esta técnica, sin embargo los pasos a seguir no estaban bien definidos y el código usado era un tanto rebuscado. Sin embargo la importancia de esta técnica radica en que al llevarla a cabo se garantiza automáticamente un 100% en los criterios de cobertura de Sentencia y Decisión. Por ello se decidió mantenerla en el paquete pero con otro ejemplo.

Entre las características más importantes de esta técnica se encuentran:

- Utiliza un gráfico de control de flujo obtenido a partir de un código, y lo usa como base para el análisis (En el ejemplo se explica cómo crear estos gráficos).
- Usa una ecuación para definir el número mínimo de casos de prueba necesarios para satisfacer los criterios mencionados.
- Usa el concepto de "ruta o camino base" dentro de un grafo, para que siguiendo un pequeño algoritmo se obtengan un conjunto de rutas base que servirán para definir los casos de prueba necesarios.

- **Corregir los defectos**

Aportaciones más relevantes:

La aportación de este ejemplo se encuentra en hacer notoria la presencia de un defecto dentro del código de la implementación, para después tomar las medidas necesarias para la corrección del mismo.

En él se ve como al ejecutar el Test runner un indicador (en rojo) aparece, lo cual significa que se ha encontrado un defecto al ejecutar las pruebas unitarias. El paso siguiente es localizar el defecto y corregirlo.

Una vez que los cambios son realizados se vuelve a mostrar el resultado del Test runner el cual muestra ahora un indicador (en verde) diferente, que significa que las pruebas unitarias han pasado satisfactoriamente y no se ha encontrado ningún defecto en ellas.

- **Ejemplos de Test runners**

Aportaciones más relevantes:

En la tarea de *Diseñar o actualizar casos de pruebas unitarias y aplicarlos* en el paso 4. *Ejecutar las pruebas unitarias* se habla acerca de herramientas que facilitan la ejecución de las pruebas al usar frameworks externos.

Con el objetivo de familiarizar al lector con éstas herramientas se dejan algunas capturas de pantalla de Test runners gráficos y un ejemplo de ejecución de un Test runner de línea de comandos.

- **Ejemplo de una macro para aserciones en C++**

Aportaciones más relevantes:

En la tarea de *Diseñar o actualizar casos de pruebas unitarias y aplicarlos* en el paso 3. *Codificar las pruebas unitarias* se habla acerca rutinas que verifican si un componente bajo ciertas circunstancias retorna un valor de verdadero o falso y que resultan de gran utilidad a la hora de codificar las pruebas unitarias. Estas rutinas se llaman *aserciones* y la mayoría de los frameworks externos las soporta, sin embargo si el lenguaje usado no soporta este tipo de frameworks se consideró importante que los programadores puedan crear sus propias aserciones. Debido a ello en este ejemplo se muestra una macro de una aserción en C++.

- **Ciclo de vida de las pruebas unitarias**

Aportaciones más relevantes:

Para evitar que los conceptos relacionados con la creación y ejecución de pruebas unitarias queden dispersos, se añadió a la sección de ejemplos un diagrama con el ciclo de vida de las pruebas unitarias, con el objetivo de integrar y ligar los pasos del proceso.

Listas de verificación

Aportaciones más relevantes:

Esta sección fue dividida en dos partes: Listas de verificación de tareas y listas de verificación de soporte. Las primeras fueron añadidas por recomendación del Dr. Claude Laporte con el objetivo de asegurar los pasos de cada una de las tareas del paquete. Las segundas fueron añadidas como apoyo para algunas tareas o actividades que así lo requerían.

Las listas de verificación de tareas son las siguientes:

- Asignar tareas a los miembros del equipo de trabajo
- Construir o actualizar componentes de software
- Diseñar o actualizar casos de pruebas unitarias y aplicarlos
- Corregir los defectos

Las listas de soporte son las siguientes:

- Revisión de código (Esta lista ya se encontraba en la versión 0.4 del paquete)
- Lo que el arquitecto y el diseñador deben proveer, se divide en la siguientes sub-listas:
    - Requerimientos
    - Arquitectura y diseño
    - Estándar de interfaz de usuario

Herramientas

Aportaciones más relevantes:

La sección de herramientas de la versión 0.4 contaba ya con una matriz de trazabilidad, el único cambio que se le hizo a esta parte fue en la redacción de las instrucciones sobre cómo usarla. En la nueva versión se describe el significado y uso de cada uno de los campos de la matriz.

Además se agregaron otras dos partes a ésta sección:

- Enlaces a las herramientas de cobertura de código más populares para diferentes lenguajes de programación.
- Enlaces a los frameworks de pruebas unitarias más populares para diferentes lenguajes de programación.

La importancia y uso de estas herramientas es mencionado en la tarea: *Diseñar o actualizar casos de pruebas unitarias y aplicarlos*.

4.1.4 Fortalezas

Con el desarrollo de la nueva versión del Paquete de Puesta en Operación de Construcción y pruebas unitarias se pretendió conservar la mayoría de las fortalezas del paquete anterior y convertir sus debilidades en fortalezas. A continuación se presenta el resumen.

Fortalezas que se conservaron del paquete anterior:

- ✓ Las tareas del paquete están acorde a lo establecido en la norma ISO/IEC 29110 parte 5-1-2
- ✓ Los pasos de las tareas son sencillos y cortos
- ✓ Las referencias del paquete con otros estándares y modelos se encuentran bien definidas

Debilidades que se convirtieron en fortalezas:

- ✓ La parte correspondiente a pruebas unitarias se enriqueció dando un mayor detalle en las actividades necesarias para su realización.
- ✓ Cuenta con plantillas y herramientas que pueden ser de gran apoyo para la realización de ciertas tareas.
- ✓ Cuenta con ejemplos ilustrativos que pretenden facilitar la comprensión y adopción de las tareas y prácticas del paquete.
- ✓ Cada elemento del paquete se encuentra en la sección que le corresponde

Nuevas fortalezas detectadas:

- ✓ El nuevo paquete tiene un enfoque más práctico, dejando atrás aquella teoría cuyo fin no sea la aplicación directa del conocimiento en el área laboral.
- ✓ Las VSE que usen el PPO están más cerca de una evaluación exitosa en el nuevo estándar internacional ISO/IEC 29110
- ✓ El PPO representa una estrategia de bajo costo para mejorar las actividades de Construcción y pruebas unitarias de las VSEs.

4.1.5 Debilidades

Las posibles debilidades del paquete sin considerar ninguna evaluación del mismo son las siguientes.

- ✗ Puede considerarse muy extenso en cuanto a contenido, en comparación del paquete anterior y con otros paquetes actuales.
- ✗ Algunas tareas y prácticas pueden considerarse complejas para cierta audiencia.

5. Propuesta para evaluar la comprensión de la nueva versión del Paquete

5.1 Objetivos de la evaluación

5.1.1 Objetivos a largo plazo

5.1.1.1 Objetivo primario

El objetivo a largo plazo es generalizar de manera cuantitativa que tan comprensible es el Paquete de Puesta en Operación de Construcción y Pruebas unitarias en la versión 0.5 por miembros de VSE a través del resultado conjunto de las evaluaciones realizadas. Reduciendo el alcance de la definición de ***Comprensión*** al significado de ***Comprensión lectora*** la cual es el siguiente:

"La comprensión lectora se concibe como el proceso en el que el lector utiliza las claves proporcionadas por el autor en función de su propio conocimiento o experiencia previa para inferir el significado que éste pretende comunicar" (Pérez Zorrilla, 2005).

Esta definición es la más apropiada desde que el objeto a evaluar es un documento escrito.

5.1.1.2 Objetivo secundario

Como objetivo secundario se tiene, que si se realiza la evaluación del PPO en dos fases, una previa a la lectura del PPO y una posterior a la lectura, no solo es posible explorar los conocimientos de los miembros de las VSE en las actividades de Construcción y Pruebas unitarias, sino que se puede hacer una aproximación de qué tanto han aprendido con la ayuda del paquete. Pudiendo medir así la eficiencia del paquete como herramienta ***autodidacta.***

La razón del porqué estos objetivos son a largo plazo es debido a que para poder generalizar el resultado es necesario evaluar a un número considerable de aspirantes. Sin embargo debido al alcance y tiempo estimado para este proyecto no fue posible conseguir dichos objetivos.

5.1.2 Objetivo a corto plazo

5.1.2.1 Objetivo primario

Debido al alcance y tiempo estimado para este proyecto un objetivo ***factible*** es definir las bases de un marco de trabajo para cumplir los objetivos a largo plazo.

Esto es seleccionar una estrategia de investigación y con ello un instrumento que sirva para evaluar al PPO de Construcción y Pruebas unitarias en su versión 0.5 y de cuyo resultado se pueda cuantificar la ***Comprensión lectora*** por parte de los evaluados.

5.1.2.2 Objetivos secundarios

- Proponer una escala que determine el nivel aproximado de eficiencia del paquete como herramienta autodidacta
- Poner a prueba el marco de trabajo y llevar a cabo un cierto número de evaluaciones piloto

5.2 Estrategia de investigación seleccionada

Existen diferentes estrategias de investigación y medios para recolectar información, algunos de ellos son:

- Experimentos de laboratorio
- Casos de estudio
- Experimentos de campo
- Trabajo de campo
- Simulaciones
- Entrevista
- Encuestas

Considerando los aspectos de generalización de la información y descripción del comportamiento del objeto de estudio, la Tabla 11 hace una comparación entre las estrategias mencionadas.

	No generaliza	**Generaliza**
Describe el comportamiento existente	• Entrevistas • Trabajo de campo	• Encuestas
Describe que causa el comportamiento	• Caso de estudio • Experimentos de laboratorio	• Experimentos de campo • Simulación • Experimentos de laboratorio

Tabla 11 Comparativa entre estrategias de investigación. (Kasunic, 2005)

Cada uno de estos enfoques tiene sus ventajas y desventajas. Sin embargo la estrategia que mejor se ajusta con el objetivo planteado es la estrategia de **Encuestas.**

Una encuesta es un enfoque de análisis que usa al cuestionario como instrumento de obtención de datos en el cual los participantes responden una serie de preguntas o sentencias que fueron desarrolladas previamente (Kasunic, 2005).

La Tabla 12 resume las características más importantes de las encuestas.

Característica	Descripción
Sistemática	La encuesta sigue un conjunto de reglas específicas; una lógica de operaciones formal y ordenada.
Imparcial	La encuesta selecciona elementos de la población sin prejuicio o preferencia
Representativa	La encuesta incluye unidades que juntas son representativas del problema de estudio y la población afectada por ella.
Basada en teoría	Las operaciones de la encuesta son guiadas por principios relevantes del comportamiento humano y leyes matemáticas de probabilidad y estadística.
Cuantitativa	La encuesta asigna valores numéricos a características no numéricas del comportamiento humano en formas que permiten la interpretación uniforme de éstas características.
Replicable	Otras personas usando los mismos métodos con los mismos medios obtienen resultados muy similares.

Tabla 12 Características más importantes de las encuestas (Backstrom, 1981)

Su desarrollo involucra un proceso de siete etapas (Kasunic, 2005) que incluye los siguientes pasos:

1. Identificar los objetivos de investigación
2. Identificar y caracterizar la audiencia objetivo
3. Diseñar el plan de muestreo
4. Diseñar y escribir el cuestionario
5. Realizar una prueba piloto del cuestionario
6. Distribuir el cuestionario
7. Analizar los resultados y reportarlos

Existen dos tipos de encuestas:

- Entrevistas
- Cuestionarios auto-administrados

En las **entrevistas** existe un entrevistador que hace las preguntas en base a un cuestionario preparado y registra la información. Las entrevistas pueden ser realizadas preguntándole directamente al entrevistado cara a cara o por teléfono. Una desventaja de éste método es que puede para obtener resultados contundentes puede llegar a ser muy costoso y requerir mucho tiempo del entrevistador.

Los **cuestionarios auto-administrados** por otro lado son aquellos que no necesitan de un entrevistador que lo realice. Este tipo de cuestionarios son auto-explicativos y se dan de manera física o electrónica al entrevistado. Una ventaja de este tipo de encuestas es que pueden hacerse de manera simultánea a muchas personas permitiendo un ahorro considerable en tiempo y costo.

Tomando en cuenta las ventajas de este último tipo de encuestas se optará por ellas para la realización del marco de evaluación. Para facilitar la obtención de resultados se usará un medio electrónico, esto último se discutirá más adelante.

5.3 El proceso de la encuesta

5.3.1 Identificar los objetivos de investigación

Antes de definir el objetivo de la encuesta, es necesario recordar que el propósito del objeto de estudio (que en este caso es el Paquete de Puesta en Operación de Construcción y pruebas unitarias en su versión 0.5) es facilitar la adopción del estándar ISO/IEC 29110 a través de la implementación de prácticas y un conjunto de herramientas afines.

Para lograr esto, el primer paso es verificar que tan comprensible resulta el paquete por la audiencia objetivo. Sin embargo como se mencionó anteriormente el término ***Comprensión*** es

muy amplio por lo que delimitaremos la definición al término de ***Comprensión lectora*** que es el que mejor se ajusta desde que el PPO es un texto del tipo expositivo[5] (Pérez Zorrilla, 2005).

Dado que leer va más allá que descodificar palabras y encadenar sus significados, existe una serie de modelos que explican los procesos implicados en la comprensión lectora, y que coinciden en la consideración de que ésta es un proceso que se desarrolla teniendo en cuenta varios niveles:

- ***Comprensión literal:***
  El lector hace valer sus destrezas de memoria y reconocimiento, al recordar detalles, reconocer ideas principales, relaciones causa-efecto etc.

- ***Reorganización de la información:***
  El lector ordena las ideas mediante procesos de clasificación y síntesis

- ***Comprensión inferencial:***
  El lector liga al texto su experiencia personal para realizar conjeturas e hipótesis, lo que le permite interpretar el texto al hacer deducciones de información e ideas que no aparecen de manera explícita en el texto.

- ***Lectura crítica o juicio valorativo***:
  Permite la reflexión sobre el contenido del texto. Para ello, el lector necesita establecer una relación entre la información del texto y los conocimientos que ha obtenido de otras fuentes, y evaluar las afirmaciones del texto contrastándolas con su propio conocimiento.

- ***Apreciación lectora***:
  Este nivel hace referencia al impacto psicológico y estético del texto en el lector, incluyendo el formato y estilo que el autor le da al texto

Entendiendo una vez los niveles a los que se refiere la comprensión lectora y conociendo el tipo de documento que se va a evaluar, queda definido entonces el objetivo de la encuesta:

Evaluar de manera cuantitativa la comprensión lectora del PPO de Construcción y Pruebas unitarias proporcionado de manera electrónica a trabajadores de VSE usando como herramienta cuestionarios que abarquen los niveles del proceso de comprensión lectora

Cabe resaltar que no existen referencias de evaluaciones de este tipo a los PPO existentes, por lo no hay un punto de partida o de comparación que sirva como retroalimentación para la mejora de las encuestas a realizar.

---

[5] Un texto expositivo se caracteriza por que su finalidad es informar, explicar o persuadir al lector, además éste asume que la información que se transmite es cierta

5.3.2 Identificar y caracterizar la audiencia objetivo

De acuerdo a como lo indica el estándar ISO/IEC 29110 para llevar a cabo todo el proceso de ciclo de vida del software son necesarios los siguientes roles dentro del contexto de una VSE:

- Analista
- Cliente
- Diseñador
- Programador
- Administrador del proyecto
- Líder Técnico

De ellos dado que el PPO estudiado sólo cubre las actividades de Construcción y Pruebas unitarias de éste proceso, el conjunto de roles que participan en éstas tareas y que así lo define el mismo PPO son los siguientes:

- Programadores
- Administrador del proyecto
- Líder Técnico

No obstante dado que la mayor parte del PPO cuenta con actividades para el rol del Programador y sólo una pequeña parte para el líder técnico y para el Administrador del proyecto. La audiencia objetivo para la evaluación la formarán los programadores.

Las características de la audiencia objetivo se muestran en la Tabla 13.

Programadores	
Competencias	Conocimiento y/o experiencia en programación, integración y pruebas unitarias. Experiencia en el desarrollo y mantenimiento de software.
Experiencia	Experiencia mínima de 6 meses desarrollando software

Tabla 13 Caracterización de la audiencia objetivo

Supuestos que se aplican a todos los grupos:

- Tienen estudios relacionados con ciencias de la computación, informática o afines.
- Tienen experiencia en el llenado de formularios impresos y electrónicos (en sitos web)
- Están familiarizados con los conceptos de Construcción de software y Pruebas unitarias y tiene un conocimiento básico-medio en los temas subyacentes.
- Disponen de un tiempo aproximado de 40 minutos para responder una encuesta.

A pesar de la caracterización definida anteriormente aún existen algunos detalles que pueden introducir un sesgo en los resultados obtenidos:

- **Falta de experiencia o conocimientos previos sobre el tema relevante del texto** (Algunas personas pueden afirmar saber sobre cierto tema y en realidad lo desconocen o lo confunden con otro). Dichas diferencias cuantitativas pueden evaluarse mediante pruebas que midan el conocimiento que el lector tiene acerca de los temas que se abordan en ellas antes de la propia prueba de comprensión.
- **Experiencias o conocimientos difieran cualitativamente** de los del autor del PPO. Los desajustes de este tipo pueden, en consecuencia, llevar al lector a elaborar un modelo de significado totalmente inadecuado sin que sea consciente de este problema

Formas de hacer frente al problema

- Elaborar una evaluación previa que permita conocer de antemano el conocimiento de la audiencia sobre los temas tratados. La puntuación obtenida por cada lector en las pruebas de comprensión podría entonces ajustarse en función de la puntuación obtenida en la prueba de conocimiento previo.
- Utilizar un lenguaje que se adecue a la subcultura del lector para realizar la evaluación. Utilizar un lenguaje estándar en las evaluaciones, utilizar distintos contenidos y estructuras, e incluir medidas que permitan discriminar entre aquellos problemas que pueden atribuirse a desajustes en el conocimiento o la experiencia previa y otro tipo de problemas
- Adecuación entre las pruebas de lectura y la instrucción lectora recibida es decir, que aquello que se transmite sea aquello que se evalúe
- A la hora de evaluar la comprensión lectora, es necesario tomar en consideración las exigencias o demandas cognitivas que el procedimiento de evaluación empleado impone al lector.

5.3.3 Diseñar el plan de muestreo

Una vez que se ha identificado y caracterizado la audiencia objetivo, el siguiente paso es determinar el número de individuos a encuestar, para ello existen dos tipos de muestras estadísticas[6]:

Muestra probabilística: Es usada para generalizar los resultados obtenidos hacia toda la población[7], incluso si no todos los individuos de la población forman parte de la encuesta. Generalmente necesita de un número muy grande de participantes y de distribuciones probabilísticas para estimar los resultados.

Muestra no probabilística: Usa el juicio humano para seleccionar a los encuestados. Dicho juicio no posee bases teóricas para estimar características de la población. La selección de individuos puede ser por participación voluntaria o a juicio o conveniencia del investigador. Los resultados

[6] Una muestra estadística es un subconjunto de casos o individuos de una población estadística. (Wilks, 1962)
[7] Es el conjunto de elementos de referencia sobre el que se realizan las observaciones. (Wilks, 1962)

obtenidos sólo se pueden aplicar a aquellos que respondan las encuestas, tratar de generalizar más allá de los encuestados es un error.

Aunque el tipo de muestra probabilística tiene una base teórica más sólida no es factible aplicarla en este proyecto debido a que la población objetivo se encuentra muy limitada debido a ciertas circunstancias:

- Poca disponibilidad de tiempo por parte del los encuestados
- Dificultad para establecer contacto con los encuestados
- En algunos casos poco interés en el tema (Por el poco tiempo de vida de la norma, no a todos les gusta formar parte de experimentos)

Por ello la muestra que mejor se adapta y que fue seleccionada para este experimento es la ***Muestra no probabilística,*** la cual inicialmente será representada por un conjunto pequeño de encuestados. No obstante para mejorar la precisión de los resultados se pretende ir incrementando paulatinamente el tamaño de dicha muestra con el objetivo de que los resultados sean cada vez más contundentes y vengan de un grupo cada vez más amplio.

Tomando en cuenta las consideraciones anteriores el plan de muestreo definido para este proyecto contempla los siguientes puntos.

- Debido a las limitaciones de tiempo con las que se cuenta, la muestra seleccionada pretende contemplar un conjunto mínimo de 4 programadores por VSE. Contemplando como mínimo la participación de 2 VSE.
- La muestra seleccionada no representa al público objetivo pero sí al conjunto que integra la muestra.
- La forma de seleccionar a los individuos de la muestra será a través de invitaciones vía correo electrónico dejando la participación de manera voluntaria.
- Para asegurar que los encuestados sean un grupo representativo del público objetivo se realizará una evaluación de exploración previa.

5.3.4 Diseñar y escribir el cuestionario

Como se mencionó anteriormente, la encuesta consistirá de dos cuestionarios que pretenden evaluar lo siguiente:

- Evaluar de manera exploratoria el conocimiento y experiencia previo de los programadores, antes de leer el PPO de Construcción y pruebas unitarias en su versión 0.5
- Evaluar la comprensión lectora de los programadores que hayan leído el PPO de Construcción y pruebas unitarias en su versión 0.5

Para ello se seguirá un proceso que involucra los siguientes pasos aplicados a la muestra seleccionada (Ver Figura 3).

1. Aplicar un ***cuestionario de exploración*** de experiencia y conocimientos en las tareas relacionadas con el paquete.
2. Entregar la herramienta a evaluar, en este caso el PPO de Construcción y pruebas unitarias.
3. Aplicar un ***cuestionario de evaluación*** de la comprensión lectora.

Figura 3. Fases del proceso de evaluación

Es importante mencionar que uno de los objetivos principales de

Para la elaboración de cada uno de los cuestionarios se seguirá un proceso que involucra de manera iterativa los siguientes pasos:

- Consultar a expertos en el tema para que sugieran algunas ideas para la realización de las preguntas.
- El investigador refina las ideas y escribe las preguntas
- El asesor revisa las preguntas y las aprueba

El Figura 4 ilustra mejor el proceso.

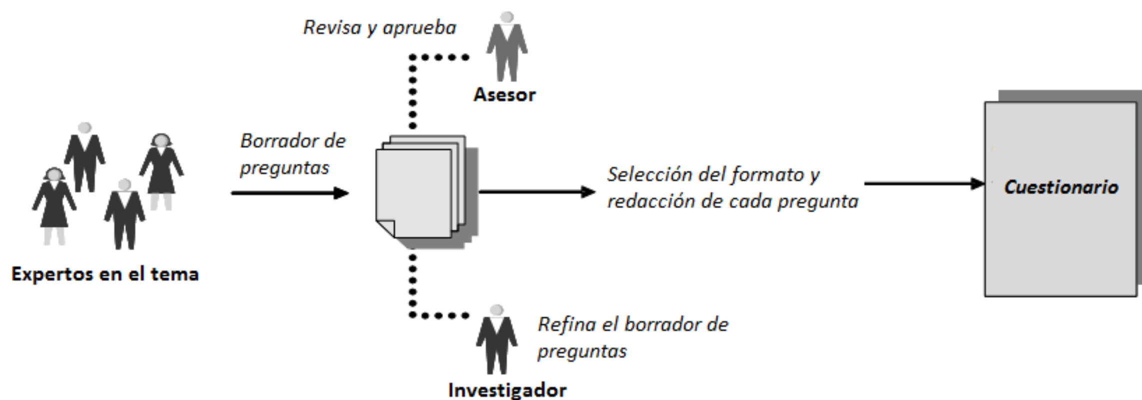


Figura 4 Proceso para la creación de los cuestionarios

Diseño del cuestionario de exploración

Objetivo del cuestionario: Evaluar la experiencia, el conocimientos previo y el interés en las tareas de Construcción y pruebas unitarias, para saber si el encuestado es un elemento representativo de la muestra

Que se mide: La experiencia y el conocimiento previo de los participantes en las tareas del PPO correspondiente.

Preguntas internas:

Las siguientes preguntas son diseñadas por el investigador para el investigador, es decir a través de ellas se pretende descubrir qué se debe preguntar a los encuestados.

1. ¿Qué tanta experiencia tiene el encuestado en el rol de programador?
2. ¿Qué tanto interés tiene el encuestado en las tareas de Construcción y pruebas unitarias?
3. ¿Qué tanto conocimiento o experiencia posee el encuestado en las tares del PPO correspondiente?
    a. Sub-tarea: Definir estándares de construcción
    b. Sub-tarea: Reportar taxonomía de defectos
    c. Tarea: Asignar tareas a los miembros del equipo de trabajo
    d. Tarea: Construir o actualizar componentes de software
    e. Tarea: Diseñar o actualizar casos de pruebas unitarias y aplicarlos
    f. Tarea: Corregir los defectos
    g. Tarea: Actualizar el registro de trazabilidad

Duración aproximada: Se estima que el cuestionario sea respondido por los participantes en un tiempo aproximado de 20 minutos.

Preguntas externas y la forma de medir los resultados:

Las preguntas externas son las preguntas que van dirigidas a la audiencia objetivo, en este caso los programadores y su finalidad es responder una o una parte de las preguntas internas.

Así la pregunta interna:

- ¿Qué tanta experiencia tiene el encuestado en el rol de programador?

Puede ser respondida con el resultado de las preguntas externas 1-3. El resultado de la pregunta externa 1 nos indica directamente la experiencia que tiene el encuestado en el rol de programador en una escala que va desde muy poca experiencia hasta mucha experiencia.

1. ¿Cuánto tiempo llevas como programador de software?	
○ Menos de 1 año	Muy poca experiencia

○ 1 – 3 años	Poca experiencia
○ 4 – 10 años	Experiencia media
○ Más de 10 años	Mucha experiencia

Por otro lado la respuesta a la pregunta 3 nos indica el tiempo promedio invertido en dicho rol y el resultado de la pregunta 2 sólo sirve para justificar dicho tiempo invertido y saber si su experiencia está dividida en diferentes responsabilidades.

2. *¿Qué roles o papeles desempeñas en tu organización?* Marca todos los que desempeñes.
☐ Programador ☐ Analista ☐ Diseñador ☐ Líder Técnico ☐ Administrador de Proyectos (Si es más que sólo programador en teoría el tiempo invertido debe reducirse y verse reflejado en la pregunta 3)

3. ¿En un proyecto común, que porcentaje de tu tiempo le dedicas a la programación de software?	
○ Menos de 30%	Poco tiempo
○ De 30% a 60%	Medio tiempo
○ Más de 60%	Tiempo completo

La pregunta interna:

- *¿Qué tanto interés tiene el encuestado en las tareas de Construcción y pruebas unitarias?*

Es respondida con el resultado de las preguntas externas 4 y 5.

4. ¿Qué opinas acerca de la realización de las pruebas unitarias?	
○ Son indispensables para todo proyecto	Alto interés
○ Pueden ser útiles en algunas ocasiones	Medio interés
○ Representan una pérdida de tiempo	Bajo interés
○ No sé, desconozco el tema	No aplica

5. ¿Consideras que tú como programador necesitas mejorar, aprender o repasar prácticas de construcción de software?	
○ Sí, para desarrollarme profesionalmente	Alto interés (individual)
○ Sí, para ser más productivo en mi organización	Alto interés (cultura organizacional)
○ No, considero que mis conocimientos son suficientes	Ningún interés

Las preguntas externas 6-21 atienden a la siguiente pregunta interna:

- *¿Qué tanto conocimiento o experiencia posee el encuestado en las tares del PPO correspondiente?*

Para ello a cada posible respuesta de cada pregunta se le asigna uno de los siguientes valores:

- Conocido: Significa que el encuestado posee conocimientos básicos sobre la tarea correspondiente
- Poco conocido: Significa que el encuestado sabe de la existencia de la tarea pero no conoce la importancia o los beneficios de ella
- Desconocido: Significa que el encuestado no sabe de la existencia de la tarea

A cada uno de estos valores se le asigna un valor numérico que servirá como un registro cuantitativo del conocimiento de los encuestados en las tareas correspondientes:

- Conocido: 2 puntos
- Poco conocido: 1 punto
- Desconocido: 0 puntos

Sub-tarea: Definir estándares de construcción:

6. ¿Sigues algún estándar de construcción de código (formato de codificación, comentarios, estructuras de control, nombres de variables, constantes, etc.)?	
○ Sí, en la organización contamos con un estándar	Conocido
○ Sí, yo sigo mi propio estándar	Poco conocido
○ No sigo ningún estándar	Poco conocido

Sub-tarea: Reportar taxonomía de defectos:

7. ¿Qué haces cuando encuentras algún defecto en los requerimientos, en el diseño o en la arquitectura?	
○ Lo corrijo yo mismo	Desconocido
○ Lo reporto a mi líder técnico o a mi superior a cargo	Conocido

○ Lo ignoro	Desconocido

Tarea: Asignar tareas a los miembros del equipo de trabajo

8. ¿Qué estrategia de integración de componentes de software es la que sigues generalmente?	
○ Bottom-up (Ascendente)	Conocido
○ Top-down (Descendente)	Conocido
○ Otra	Conocido
○ Desconozco que es una estrategia de integración	Desconocido

9. ¿Te indican de alguna forma la dificultad de los componentes de software que te asignan?	
○ Sí, de manera escrita	Conocido
○ Sí, de manera verbal	Poco conocido
○ No me mencionan nada	Desconocido

Tarea: Construir o actualizar componentes de software

10. ¿Has usado el enfoque de Pseudocódigo para la generación de componentes de software?	
○ Es el enfoque que uso habitualmente	Conocido
○ Lo he usado en ocasiones	Conocido
○ Lo conozco pero no lo uso	Poco conocido
○ Nunca he oído de él	Desconocido

11. ¿Has usado el enfoque de Diagramas de flujo para la generación de componentes de software?	
○ Es el enfoque que uso habitualmente	Conocido
○ Lo he usado en ocasiones	Conocido
○ Lo conozco pero no lo uso	Poco conocido
○ Nunca he oído de él	Desconocido

12. ¿Has usado listas de verificación de código para buscar defectos en tus

componentes?	
○ Las uso habitualmente	Conocido
○ Las he usado en ocasiones	Conocido
○ Las conozco pero no las uso	Poco conocido
○ Nunca he oído de ellas	Desconocido

Tarea: Corregir los defectos

13. Cuando detectas una anomalía o defecto en tu código ¿cómo lo localizas?	
○ Con listas de verificación	Conocido
○ A través del depurador	Conocido
○ A prueba y error	Poco conocido
○ Todas las anteriores	Conocido

Tarea: Actualizar el registro de trazabilidad

14. Después de que creas un componente de software ¿Existe alguna forma de que éste pueda ser trazado a un elemento del diseño?	
○ Sí	Conocido
○ No	Poco conocido
○ Lo desconozco	Desconocido

Tarea: Diseñar o actualizar casos de pruebas unitarias y aplicarlos

15. ¿Sabes en qué consisten las pruebas de caja blanca?	
○ Sí	Conocido
○ Tengo una idea	Poco conocido
○ No	Desconocido

16. ¿Has realizado pruebas unitarias al código que generas?	
○ Sí	Conocido
○ En ocasiones	Conocido

○ Nunca	Poco conocido
○ Desconozco que son las pruebas unitarias	Desconocido

Si la respuesta de la pregunta anterior fue: “*Sí*” o “*En ocasiones*” las siguientes preguntas deben ser contestadas, de lo contrario se pueden dejar en blanco.

17. ¿Cuáles de los siguientes pasos ejecutas cuando realizas las pruebas unitarias?	
○ Diseñar los casos de prueba	Poco conocido
○ Codificar las pruebas unitarias	Poco conocido
○ Ejecutar las pruebas unitarias	Poco conocido
○ Todos los anteriores	Conocido

18. ¿Cómo decides cuando debes dejar de realizar pruebas unitarias a un componente?	
○ Cuando de acuerdo a mi experiencia considero que he realizado suficientes	Poco conocido
○ Cuando he cubierto el criterio que el Administrador de Proyecto me ha indicado	Conocido
○ Cuando todas las pruebas que se me han ocurrido han pasado satisfactoriamente	Poco conocido
○ Cuando he agotado el tiempo que tenía destinado para realizar las pruebas	Poco conocido

19. ¿Sabes en qué consisten los criterios de cobertura de código?	
○ Sí	Conocido
○ Tengo una idea	Poco conocido
○ No	Desconocido

20. ¿Has usado algún framework de pruebas unitarias para codificar las pruebas?	
○ Sí	Conocido
○ No	Poco conocido
○ No sé que es un framework de pruebas unitarias	Desconocido

21. ¿Sabes qué un Test runner?	
○ Sí	Conocido
○ Tengo una idea	Poco conocido
○ No	Desconocido

Por último para evaluar el nivel de conocimiento o experiencia previa de un encuestado en las tareas del PPO correspondiente, es necesario conocer la puntuación que representa el 100% de los puntos para cada tarea (Ver tabla 14).

Tarea o Sub-Tarea	***Total de Puntos***
Definir estándares de construcción	2
Reportar taxonomía de defectos	2
Asignar tareas a los miembros del equipo de trabajo	4
Construir o actualizar componentes de software	6
Diseñar o actualizar casos de pruebas unitarias y aplicarlos	14
Corregir los defectos	2
Actualizar el registro de trazabilidad	2

Tabla 14 Total de puntos por tarea del paquete que representan un 100% en el resultado de la evaluación de exploración

<u>Ejemplo 1</u>

Por ejemplo para un encuestado que obtenga los resultados de la Tabla 15.

Tarea o Sub-Tarea	***Puntos***
Definir estándares de construcción	1
Reportar taxonomía de defectos	0
Asignar tareas a los miembros del equipo de trabajo	3
Construir o actualizar componentes de software	4
Diseñar o actualizar casos de pruebas unitarias y aplicarlos	3
Corregir los defectos	2
Actualizar el registro de trazabilidad	0

Tabla 15. Resultados obtenidos en la evaluación de exploración para el ejemplo 1

Obtendrá la gráfica de la Figura 5.

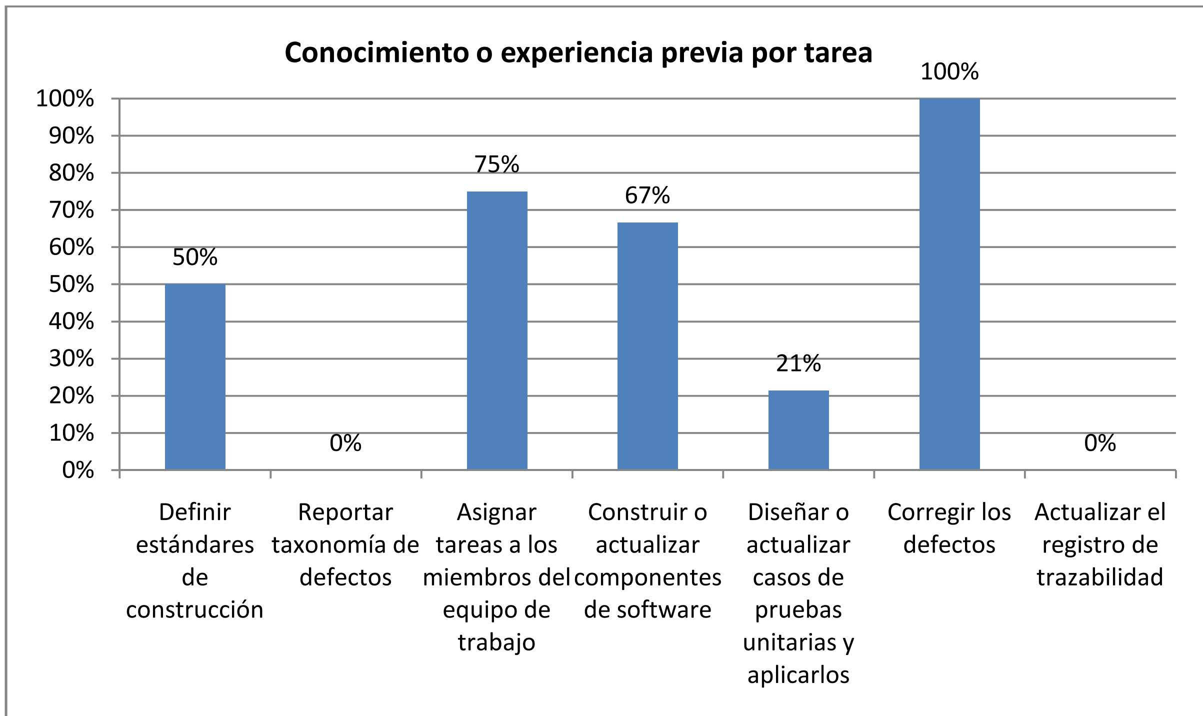


Figura 5. Gráfico obtenido de los valores de la tabla 14

De esta forma se puede obtener una medida cuantitativa del conocimiento y experiencia previa del programador en cada una de las tareas del paquete. Cabe mencionar que una baja puntuación en alguna de las tareas no significa que el programador desconozca completamente el área sino que al menos para la pregunta o las preguntas relacionadas no conoce lo suficiente para responderlas correctamente.

Nota: El cuestionario completo viene en la sección de anexos.

Diseño del cuestionario de evaluación

Objetivo del cuestionario: Evaluar la comprensión lectora del PPO de Construcción y Pruebas unitarias

Que se mide: La comprensión lectora del PPO de Construcción y Pruebas unitarias por los participantes.

Preguntas internas:

Una vez que el encuestado haya leído el paquete es importante conocer lo siguiente:

1. ¿Cuál es el nivel de comprensión literal obtenido por el encuestado?
2. ¿Cuál es el nivel de Reorganización de la información obtenido por el encuestado?
3. ¿Cuál es el nivel de Comprensión inferencial obtenido por el encuestado?
4. ¿Cuál es el juicio valorativo del encuestado?
5. ¿Cuál es la apreciación lectora del encuestado?

Duración aproximada: Se estima que el cuestionario sea respondido por los participantes en un tiempo aproximado de 20 minutos.

Preguntas externas y la forma de medir los resultados:

Se diseñaron una serie de preguntas que atendieran a cada uno de los niveles de comprensión lectora, quedando una distribución como lo muestra la Tabla 16.

Nivel de comprensión	***Número de preguntas que la abordan***
Comprensión literal	15
Reorganización de la información	5
Comprensión inferencial	6
Lectura crítica o juicio valorativo	3
Apreciación lectora	1

Tabla 16. Número de preguntas diseñadas para cada nivel de comprensión lectora.

El número de preguntas mostrado en la tabla de anterior representa una puntuación del 100% para el nivel de comprensión correspondiente. De esta forma en el resultado de las evaluaciones se puede desglosar la puntuación obtenida para cada nivel y calcular el promedio de la comprensión lectora considerando únicamente los niveles de ***Comprensión literal, Reorganización de la información y Comprensión literal***. Esto debido a que sólo estos niveles se pueden medir de manera cuantitativa ya que sus preguntas poseen únicamente una respuesta correcta. Para los niveles de ***Lectura crítica y Apreciación lectora*** donde no existen respuestas correctas o incorrectas, los resultados son cualitativos y su interpretación depende en cierta medida del impacto que causa el paquete en el encuestado considerando aspectos de juicio valorativo y de satisfacción.

Una baja puntuación en alguno de los niveles puede deberse al esfuerzo o capacidades del encuestado o bien a la estructura, contenido u organización del paquete. No obstante lo que determinará esto será el resultado del cuestionario de exploración, el cual evalúa el interés, experiencia y conocimiento del encuestado, mismos que representan un factor importante para poder decidir si el resultado de la puntuación recae totalmente en el paquete o en el desempeño del encuestado.

En caso de que el problema recaiga principalmente en el paquete, los resultados de una baja puntuación (menor o igual a un 50%) en alguno de los niveles de comprensión pueden tener una interpretación como la de la Tabla 17.

Nivel de comprensión	***Una baja puntuación puede representar:***
Comprensión literal	Los temas tratados pueden tener una complejidad elevada o manejar un lenguaje demasiado especializado
Reorganización de la información	La organización de la información en el paquete es confusa
Comprensión inferencial	Las ideas principales no se transmiten

	apropiadamente
Lectura crítica o juicio valorativo	Existen algunos temas que se encuentran demás o bien hace falta que se mencionen algunos otros.
Apreciación lectora	El formato, redacción o presentación del paquete necesita ser mejorado.

Tabla 17 Significado probable de una puntuación menor o igual a 50% en el nivel de comprensión correspondiente

Ejemplo 2

Por ejemplo para un encuestado que obtenga los resultados de la Tabla 18

Nivel de comprensión	***Número de preguntas respondidas correctamente***
Comprensión literal	12
Reorganización de la información	4
Comprensión inferencial	5

Tabla 18 Preguntas respondidas correctamente por nivel de comprensión para el ejemplo 2

Para los niveles de ***Comprensión literal, Reorganización de la información y Comprensión literal*** obtendría una gráfica como la de la Figura 6.

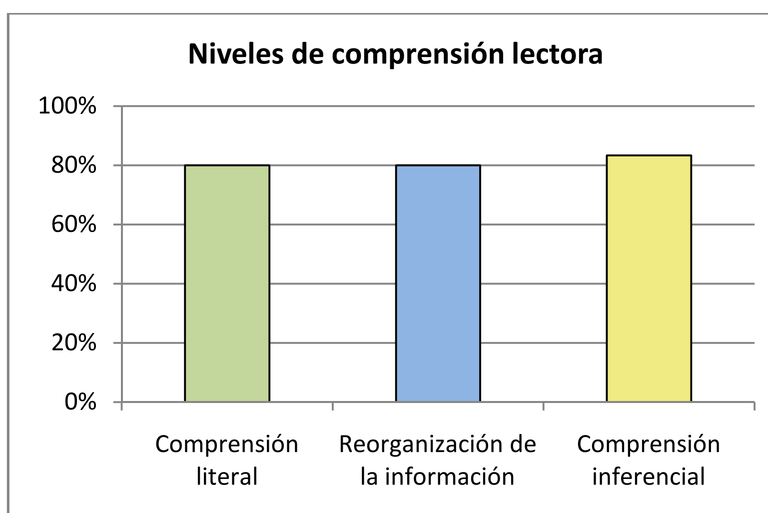


Figura 6 Resultados obtenido de los valores de la tabla 17

De la gráfica anterior se calcula el **promedio** de los tres niveles de comprensión obteniendo un resultado de **81%.** Este valor representa la medida cuantitativa de la comprensión lectora obtenida por el encuestado, donde un 100% representa una excelente comprensión lectora.

Sin embargo este porcentaje de comprensión lectora obtenido representa únicamente un aproximado de la comprensión del paquete como una herramienta autodidáctica de lectura.

Por otro lado las respuestas obtenidas para los niveles de **Lectura crítica y Apreciación lectora** tendrán la función de conocer el juicio y nivel de satisfacción del encuestado, con el objetivo de mejorar aquello que se considere necesario.

Para tener un resultado más objetivo es necesario comparar los resultados obtenidos en el cuestionario de exploración y los obtenidos con esta evaluación, con el objetivo de saber en qué medida el paquete ha logrado transmitir los conocimientos de las áreas correspondientes. Este análisis se discutirá más adelante.

5.3.5 Realizar una prueba piloto del cuestionario

Como se mencionó anteriormente la audiencia objetivo son los programadores de VSE, sin embargo debido al tiempo y a la disponibilidad de la audiencia, para la prueba piloto se optó por invitar a estudiantes de Maestría en Ciencias e Ingeniería de la computación que tuvieran experiencia en la programación de sistemas.

La invitación se realizó y se obtuvieron 8 participantes para la primera etapa (cuestionario de exploración). Sin embargo para la tercera etapa (cuestionario de evaluación) sólo se obtuvo la respuesta de 5 de los 8 participantes.

Cabe mencionar que debido a algunos percances en la configuración del servidor, el sitio diseñado para las evaluaciones no estuvo disponible para la prueba piloto y la aplicación y distribución de los cuestionarios tuvo ciertos cambios que serán mencionadas más adelante.

5.3.6 Distribuir el cuestionario

5.3.6.1 Estrategia de distribución

Para distribuir el cuestionario se construyó y adaptó[8] un sitio web dedicado especialmente para temas relacionados con el estándar ISO/IEC 29110 en América Latina, en el cual también se montó el PPO de Construcción de pruebas unitarias en su versión en español.

El sitio fue desarrollado con el Gestor de contenidos (*Content Management System, CMS de sus siglas en inglés)* Joomla versión 1.5 y las encuestas forman parte de un componente que permite registrar y analizar los resultados.

Para poder visualizar y tener acceso a la evaluación es necesario un nombre de usuario y una contraseña. Estos datos son proporcionados a los participantes que se consideren potenciales para la muestra representativa, previa una invitación formal.

La primera evaluación disponible para los participantes es el "*Cuestionario de Exploración",* el cual corresponde al primer cuestionario del proceso de evaluación.

[8] La primera versión del sitio la realizó la Ing. Ana Luisa Ríos Flores, de la facultad de Ingenier

Por otro lado, debido a ciertas restricciones que presenta el módulo de encuestas, ***el cuestionario de evaluación*** diseñado para medir la comprensión lectora se dividió en dos partes:

- Un cuestionario que mide la comprensión ***literal***, ***inferencial*** y de reorganización ***de la información***, llamado*: "Cuestionario de Evaluación del PPO de Construcción y Pruebas unitarias"*, el cual cuenta con la opción de calificación automática, lo que permite una retroalimentación inmediata al usuario una vez que termina de contestar las preguntas, ya que las preguntas diseñadas para estos niveles tienen una única respuesta correcta, obteniendo así una medida cuantitativa de los resultados.

- Un cuestionario que complementa al cuestionario de evaluación llamado: "Cuestionario de retroalimentación" el cual mide de manera cualitativa los niveles de comprensión lectora restantes: ***Juicio valorativo y apreciación lectora***, a través del análisis de las respuestas diseñadas para dicho cuestionario

Cada cuestionario muestra las instrucciones correspondientes para llenar los formularios (Ver Figura 7).

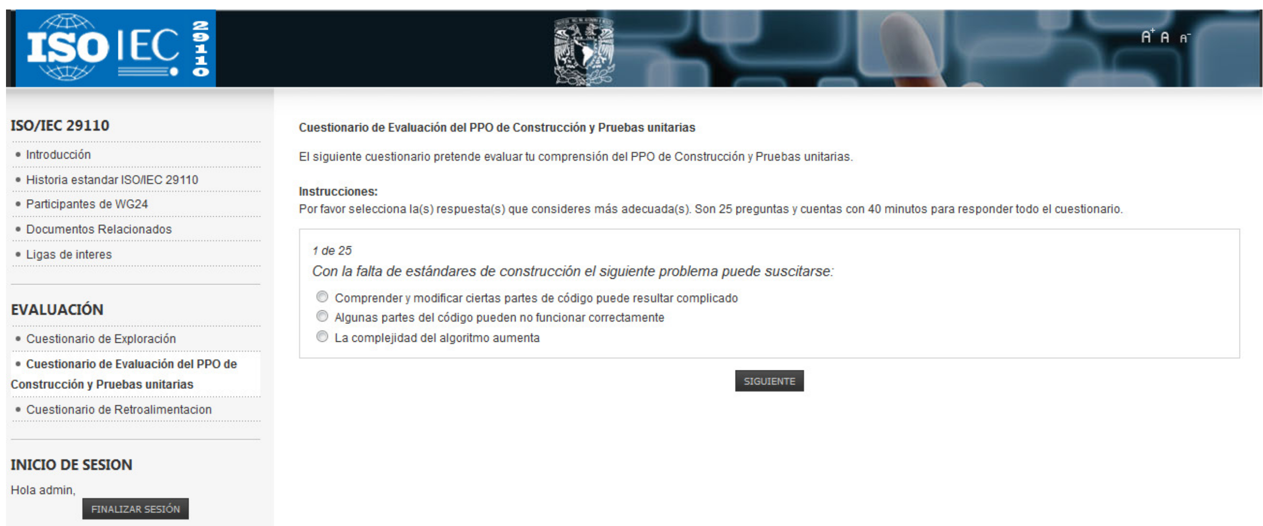


Figura 7 Captura de pantalla del sitio web que contiene el PPO de Construcción y pruebas unitarias y los cuestionarios de la evaluación

El sitio también se encuentra disponible en inglés, así como el paquete, el cual originalmente fue escrito en este idioma (para cambiar el idioma sólo basta con seleccionar el lenguaje deseado en la sección de IDIOMA). Esto fue pensado para que usuarios de países anglosajones puedan conocer los proyectos y trabajos realizados en países de habla hispana (Ver Figura 8).

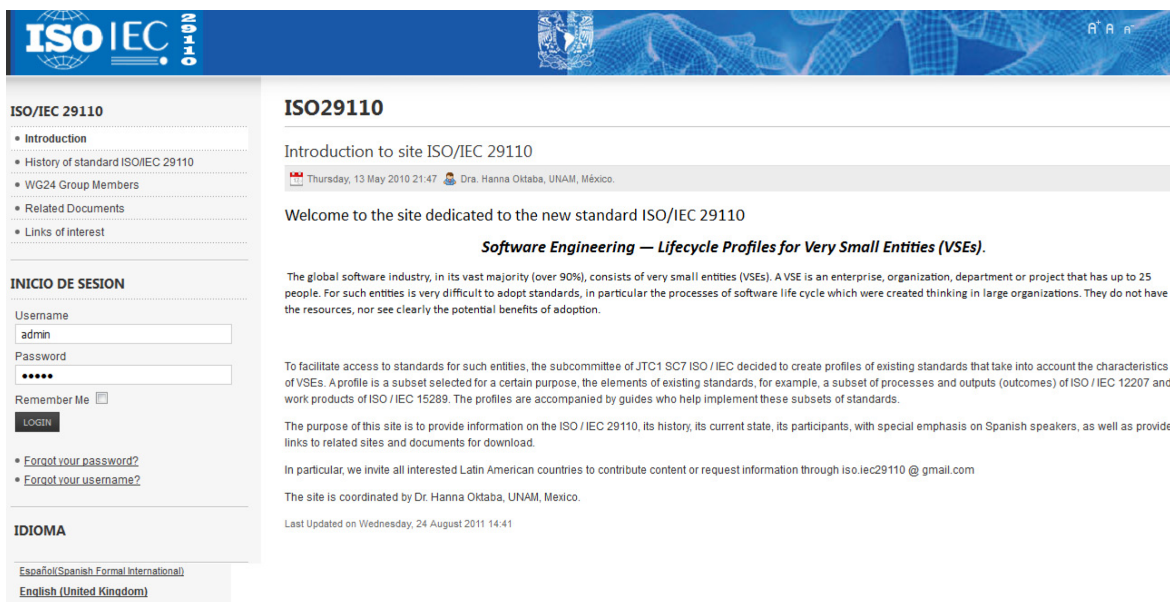


Figura 8 Captura de pantalla del sitio web en su versión en Inglés.

Cabe mencionar que aunque el sitio ya está construido, aún no se ha publicado debido a algunos contratiempos en la instalación del servidor dedicado. Sin embargo este trabajo quedará pendiente como trabajo futuro.

5.3.6.2 Distribución para la prueba piloto

Por las limitaciones mencionadas anteriormente, la distribución de los cuestionarios para la evaluación no fue posible llevarla a cabo a través del sitio web diseñado para éste propósito.

Por lo que la forma de distribuirlos fue la siguiente:

Para el ***cuestionario de exploración*** se entregó el material de manera impresa a cada uno de los 8 participantes, a los cuales se les dio una breve explicación y las instrucciones de manera verbal.

Por otro lado el ***cuestionario de evaluación*** fue distribuido a través de un servicio de encuestas en línea de nombre SurveyMonkey[9]. El cual a su vez por limitaciones en el servicio gratuito de la herramienta se tuvo que dividir en tres partes cuyos enlaces son los siguientes:

http://www.surveymonkey.com/s/YJZ827K

http://www.surveymonkey.com/s/YW8NHCB

http://www.surveymonkey.com/s/YSK9YYK

[9] Servicio gratuito de encuestas en línea: http://es.surveymonkey.com/

El motivo por el cual el primer cuestionario fuera entregado de manera presencial es para tener un panorama general del interés de los participantes en el tema y en la evaluación en general.

5.3.7 Analizar los resultados y reportarlos

Como ya se mencionó anteriormente, los resultados que a continuación se muestran no pretenden generalizar sobre toda la audiencia objetivo, sino sobre la muestra en cuestión. En este caso la muestra corresponde a la selección hecha para la prueba piloto (8 programadores).

Resultados del cuestionario de exploración

El cuestionario fue impreso y aplicado de manera presencial en un salón donde se encontraban los 8 participantes.

Entre algunas de las observaciones que se hicieron notar durante y después de la aplicación del cuestionario se encuentran:

- El tiempo promedio que le tomó responder a cada participante el cuestionario fue de aproximadamente 15 minutos.
- Dos de los participantes comentaron la sección de información adicional que corresponde a problemas, comentarios o sugerencias acerca de las preguntas del cuestionario.

Los resultados obtenidos en el cuestionario de exploración son plasmados en las siguientes gráficas, cuyas escalas corresponden a las definidas en la etapa de construcción del cuestionario.

Experiencia como Programador (en años):

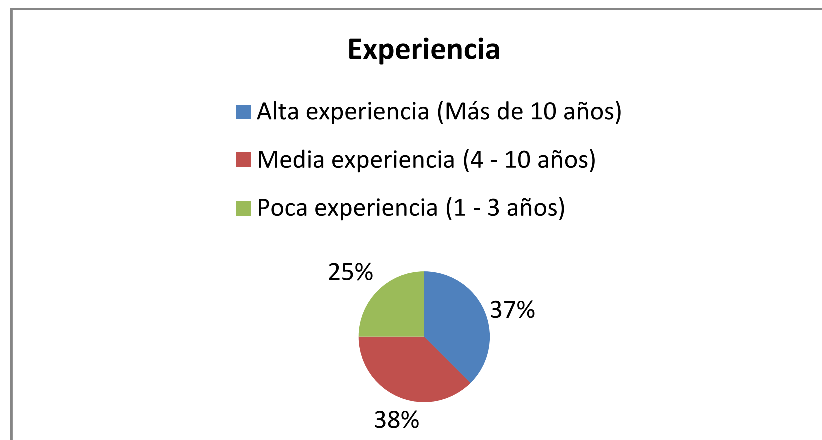


Figura 9 Experiencia de los participantes en el rol de programador

Un 38% de los participantes posee una experiencia media como programadores de software. (Ver Figura 9).

Porcentaje del tiempo invertido en la programación de componentes en un proyecto:

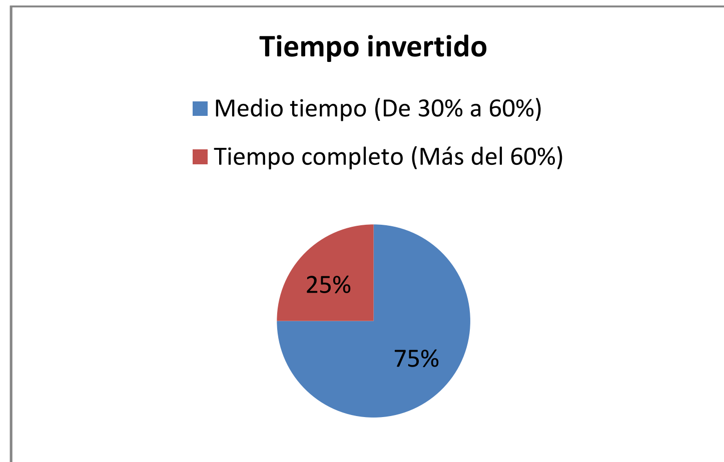


Figura 10 Tiempo invertido por proyecto por los participantes en la programación de componentes.

Un 75% de los participantes invierten alrededor de la mitad de su tiempo en el desarrollo de software en un proyecto, lo que indica que muy probablemente además del rol de programador desempeñan otros roles con distintas responsabilidades. Por lo que su experiencia en años pude verse afectada por el tiempo que le invierten a la construcción de componentes de software en un proyecto (Ver Figura 10).

Interés en la tarea de Pruebas unitarias:

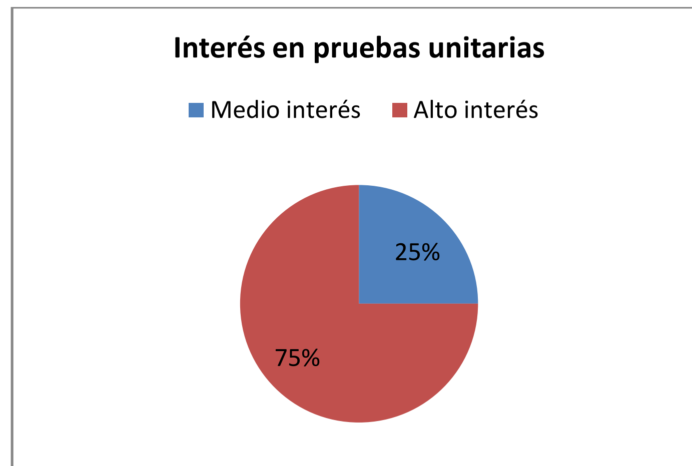


Figura 11 Interés de los participantes en el tema de pruebas unitarias

Un 75% de los encuestados considera que las pruebas unitarias son indispensables para todo proyecto, y un 25% considera que pueden llegar a ser útiles en algunas ocasiones (Ver Figura 11).

Interés en la tarea de Construcción de software:

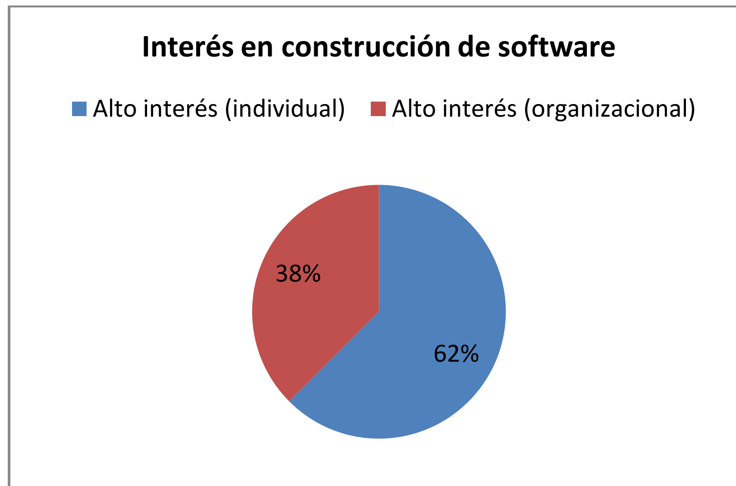


Figura 12 Interés de los participantes en el tema de construcción de software.

Un 62% de los encuestados considera que necesita mejorar, aprender o repasar prácticas de construcción de software para desarrollarse profesionalmente, y un 38% considera lo mismo pero para el desarrollo de su organización más que el individual (Ver Figura 12).

Conocimiento previo en las tareas de Construcción de software y Pruebas unitarias:

A continuación se presentan los porcentajes del conocimiento previo que tienen en promedio los participantes de la encuesta (por tarea correspondiente).

Es importante mencionar que el cálculo para obtener estos resultados fue explicado en el diseño del cuestionario, y que un 100% en alguna de las tareas significa que todos los participantes respondieron con la respuesta que es mapeada al valor de “conocido” lo que indica que tienen conocimientos básicos sobre la tarea correspondiente, considerando únicamente el dominio de las preguntas. Es decir un sujeto puede tener mayores conocimientos en la tarea relacionada, pero como el dominio se restringe únicamente a las preguntas, el conocimiento total no se ve reflejado. Aún así las preguntas fueron diseñadas para ser lo más generales posibles.

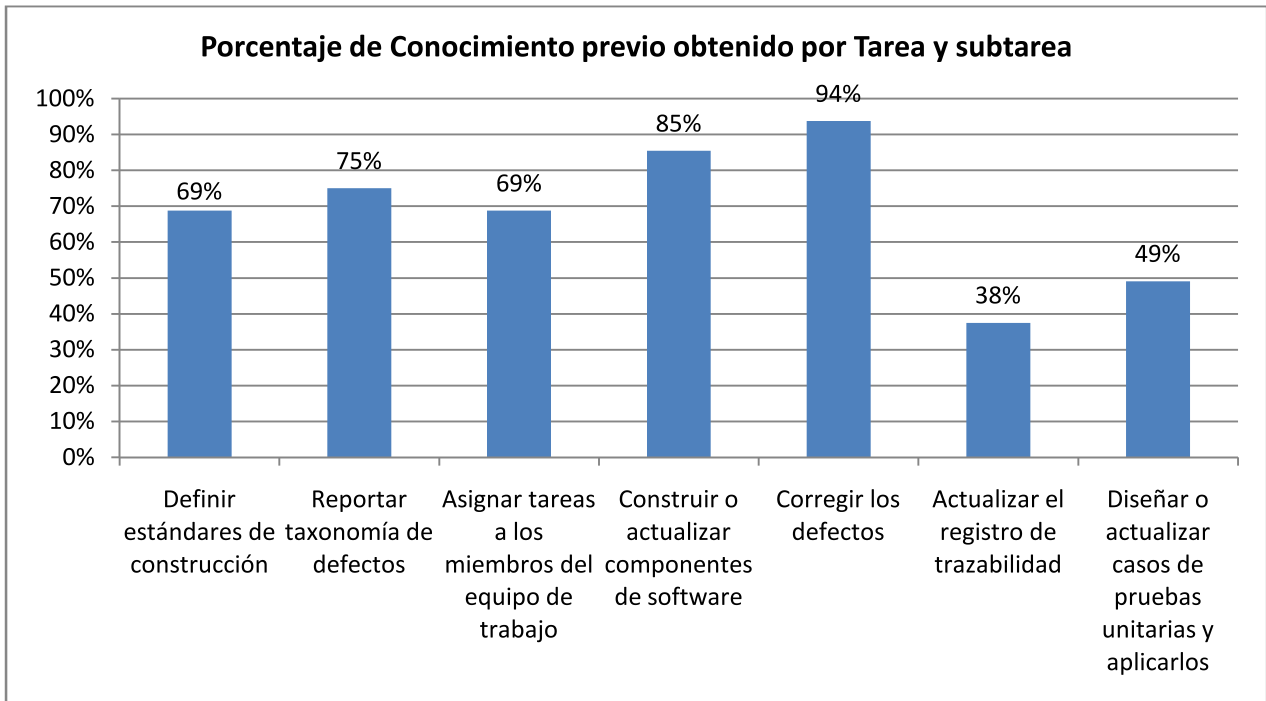


Figura 13 Gráfico generado de los resultados obtenidos en el cuestionario de exploración aplicado a los participantes de la muestra

De los resultados obtenidos (Ver Figura 12), se puede observar que la tarea en la que presentan mayor conocimiento es la de "Corregir defectos" con un 94%, seguida de la tarea de "Construir o actualizar componentes de software" con un conocimiento del 85%. Lo cual indica que tienen conocimientos suficientes en las áreas de construcción de software.

Por otra parte en la tarea de "Diseñar o actualizar casos de pruebas unitarias y aplicarlos" se obtuvo un 49% lo que indica un conocimiento muy pobre en las tareas relacionadas con pruebas unitarias.

Otra tarea en la que se obtuvieron resultados muy bajos fue la de "Actualizar el registro de trazabilidad" con un conocimiento de un 38% lo que indica que la mayoría de los encuestados no cuentan con alguna estrategia sólida para poder mapear los componentes construidos al diseño planteado.

Información adicional de retroalimentación

Como información adicional proporcionada por los encuestados se encuentra las siguientes situaciones de las cuales se plantea una posible situación para cada caso:

Problemática	"En la pregunta 14 no entendí a qué se refiere con ser trazado a un elemento del diseño"
Solución	La frase "Existe alguna forma de que éste pueda ser ***trazado*** a un elemento del

	diseño" fue cambiada por: "Existe alguna forma de que éste pueda ser ***mapeado*** a un elemento del diseño"

Problemática	"En la pregunta 8 no está la opción de: conozco esas estrategias pero no las uso"
Solución	Se añade la opción "No sigo ninguna estrategia de integración" como respuesta para la pregunta 8

Problemática	"En la pregunta 20 no conozco framework para ese fin"
Solución	La opción "No sé que es un framework de pruebas unitarias" se cambia por "No conozco ningún framework de pruebas unitarias"

Además de esta información que fue proporcionada de manera explícita por los encuestados, se cambiaron algunos otros detalles que fueron detectados dentro de las respuestas de los participantes:

Las opciones de la pregunta 17:

¿Cuáles de los siguientes pasos ejecutas cuando realizas las pruebas unitarias?

○ Diseñar los casos de prueba

○ Codificar las pruebas unitarias

○ Ejecutar las pruebas unitarias

○ Todos los anteriores

Se cambiaron por cajas de selección (check-box) ya que la respuesta puede ser más de una.

☐ Diseñar los casos de prueba
☐ Codificar las pruebas unitarias
☐ Ejecutar las pruebas unitarias
☐ Todos los anteriores

Entrega del Paquete de Puesta en Operación de Construcción y Pruebas unitarias

El paquete fue proporcionado de manera electrónica a cada uno de los participantes a través de su correo electrónico. El tiempo que se dispuso para su lectura fue de dos semanas.

Resultados del cuestionario de evaluación

Entre algunas de las observaciones que se hicieron notar durante y después de la aplicación del cuestionario se encuentran:

- El tiempo promedio que le tomó responder a cada participante el cuestionario fue de aproximadamente 34 minutos.
- De los 8 participantes del cuestionario de exploración sólo 5 continuaron con la siguiente etapa del proceso respondiendo el cuestionario de evaluación.

Como se mencionó anteriormente esta evaluación tiene como objetivo medir la comprensión lectora del encuestado en sus diferentes niveles, considerando únicamente el dominio de la muestra de la prueba piloto. Es decir bajo ninguna circunstancia se pretende generalizar para todos los programadores de VSE, sino únicamente para el grupo de encuestados en cuestión, que en este caso se trata de 5 participantes.

Para los niveles de Comprensión literal, Reorganización de la información y Comprensión inferencial se realizaron una serie de preguntas que abordan las áreas de conocimiento que trata el PPO de Construcción y pruebas unitarias.

Los resultados obtenidos fueron los que se muestran en la Figura 14.

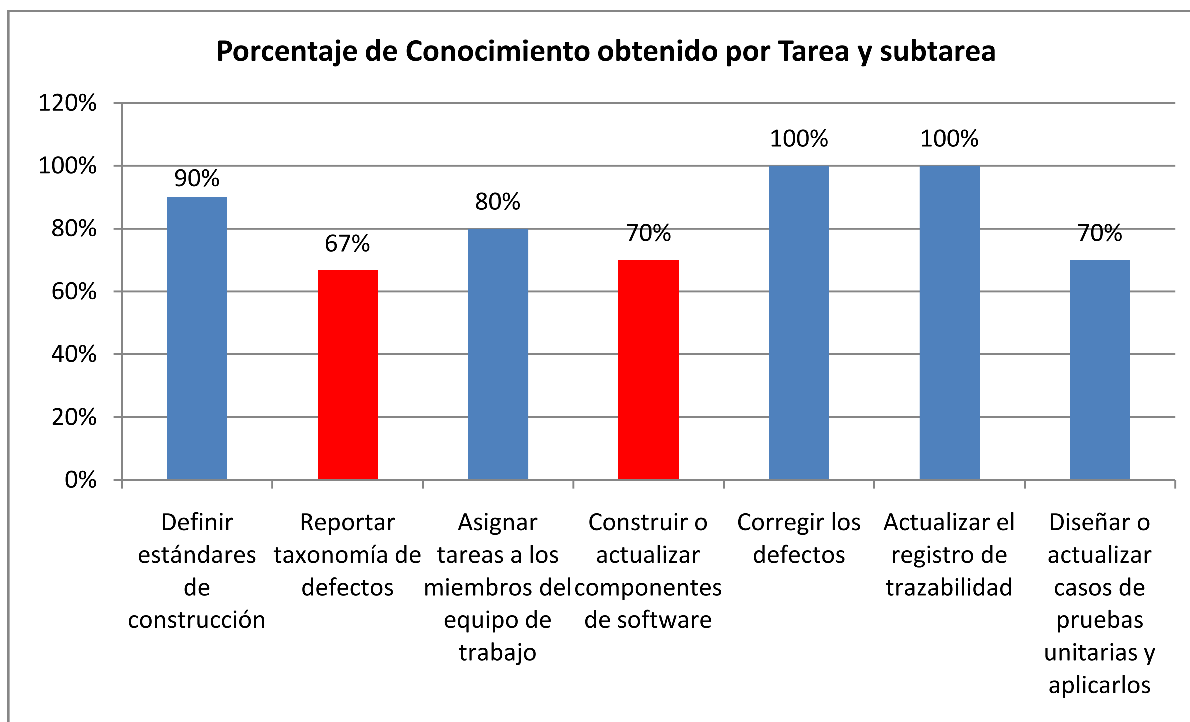


Figura 14 Gráfico generado de los resultados obtenidos en el cuestionario de evaluación aplicado a los participantes de la muestra

De los datos mostrados anteriormente, las barras en azul indican que se obtuvo un incremento en el conocimiento de la tarea o sub-tarea correspondiente con respecto a la obtenida en el cuestionario de exploración. Por otro lado las barras rojas indican un decremento en la comprensión de la tarea relacionada.

Esto se ve más claro en la Tabla 19, en la cual se muestra el porcentaje obtenido en el cuestionario de exploración y en el cuestionario de evaluación. Así como el incremento o decremento asociado

Tarea o sub-tarea	***Cuestionario de exploración***	***Cuestionario de evaluación***	***Incremento/ Decremento***
Definir estándares de construcción	69%	90%	21%
Reportar taxonomía de defectos	75%	67%	-8%
Asignar tareas a los miembros del equipo de trabajo	69%	80%	11%
Construir o actualizar componentes de software	85%	70%	-15%
Corregir los defectos	94%	100%	6%
Actualizar el registro de trazabilidad	38%	100%	62%
Diseñar o actualizar casos de pruebas unitarias y aplicarlos	49%	70%	21%

Tabla 19 Comparativa de los porcentajes obtenidos en los cuestionarios de exploración y evaluación

Como se aprecia en la tabla anterior en la mayoría de las tareas y sub-tareas hubo un incremento en el conocimiento lo cual indica como consecuencia un nivel de comprensión lectora positivo en los casos correspondientes.

No obstante para la sub-tarea de ***Reportar taxonomía de defectos*** y la tarea de ***Construir o actualizar componentes de software*** se obtuvo un resultado negativo. En ambos casos el porcentaje obtenido en el cuestionario de evaluación resultó menor que el obtenido en el cuestionario de exploración, lo cual se puede deber a alguna de las siguientes causas:

- Debido a la dificultad inherente que añade el cuestionario de evaluación con respecto al cuestionario de exploración. (El de exploración sólo verifica que los encuestados conozcan los temas relacionados).
- Debido a algún problema de redacción o de interpretación del PPO de Construcción y pruebas unitarias.
- Debido a algún problema de redacción o de interpretación en alguno de los cuestionarios.
- Debido a la complejidad de la tarea o sub-tarea del PPO de Construcción y pruebas unitarias.
- Debido al sesgo que se inyecta cuando los encuestados poseen experiencias diferentes
- Debido al sesgo que se inyecta cuando el interés de los encuestados varía considerablemente.

Nota: Es importante mencionar que para futuras evaluaciones, los participantes cuyas respuestas representen un sesgo muy representativo en comparación con las respuestas del promedio de los demás participantes deben de ser omitidos de los resultados, ya que se pueden ver como puntos aislados en la gráfica resultante que generan un impacto significativo al resultado final (Claes & Per, 2000)*. Sin embargo en este caso por el tamaño de la muestra de la prueba piloto y para tener una perspectiva más general de la audiencia objetivo se decidió incluir los resultados de todos los participantes de la muestra.*

Para poder medir los niveles de comprensión lectora, las preguntas que abordan las tareas y sub-tareas del paquete fueron diseñadas para cubrir los niveles de Comprensión literal, Reorganización de la información y Comprensión inferencial. La distribución de las preguntas para cada nivel se ve reflejada en la Tabla 20.

Nivel de comprensión	***Preguntas que la abordan***
Comprensión literal	1,2,3,6,8,12,11,15,17,18, 20,22,23,24,26,
Reorganización de la información	4,7,10,14,21
Comprensión inferencial	5,9,13,16,19,25

Tabla 20 Preguntas del cuestionario de evaluación que abordan los niveles de comprensión lectora

Los porcentajes obtenidos en la prueba piloto para cada uno de los niveles de comprensión mencionados anteriormente se muestran en la Figura 15.

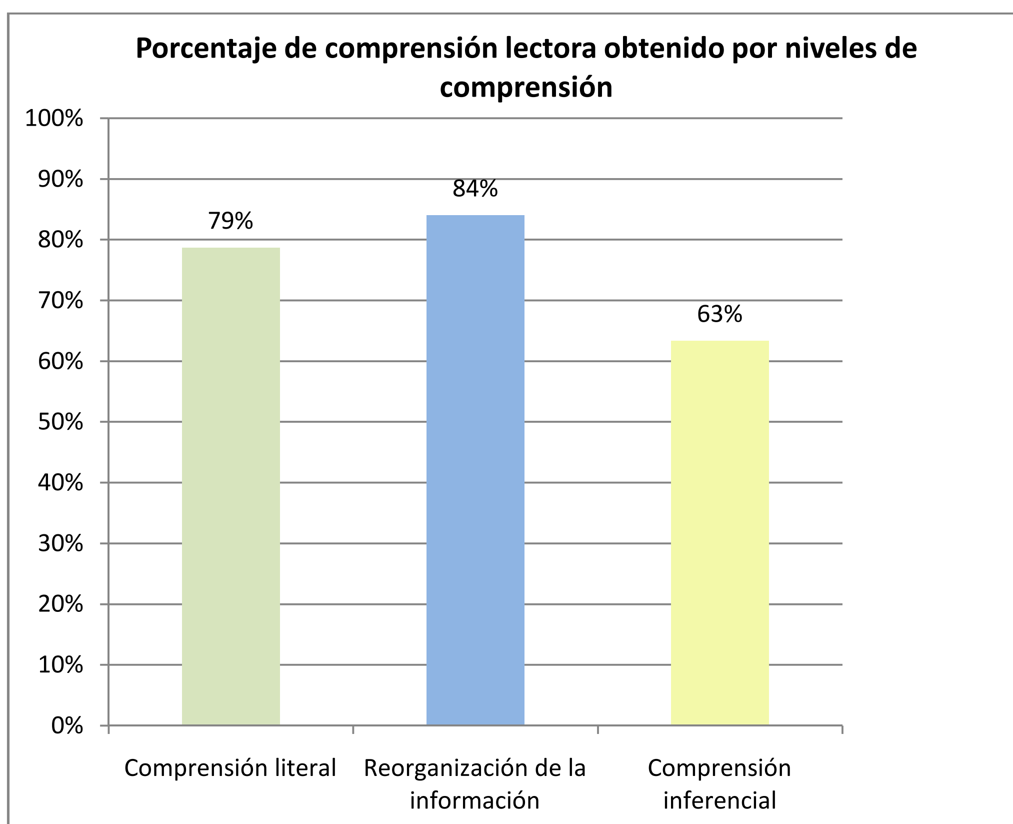


Figura 15 Gráfico generado del porcentaje obtenido por nivel de comprensión en el cuestionario de evaluación.

De la gráfica anterior se aprecia que el porcentaje más alto obtenido fue en el nivel de ***Reorganización de la información*** con un 84%, seguido del nivel de ***Comprensión literal*** con un 79%, seguido de la ***Comprensión inferencial*** con un 63%. Esta última era de esperare ya que es el nivel de comprensión que requiere un mayor esfuerzo.

En promedio la comprensión lectora de la muestra, para lo niveles mencionados anteriormente, que son los que se pueden medir de manera cuantitativa resulta en **75%**. Lo cual no es el mejor

resultado que se esperaría, pero considerando el tamaño de la muestra y la no eliminación de los sesgos se podría considerar un buen resultado.

Por otro lado las respuestas obtenidas para los niveles de ***Lectura crítica y Apreciación lectora*** tendrán la función de conocer el juicio y nivel de satisfacción del encuestado, con el objetivo de mejorar aquello que se considere necesario.

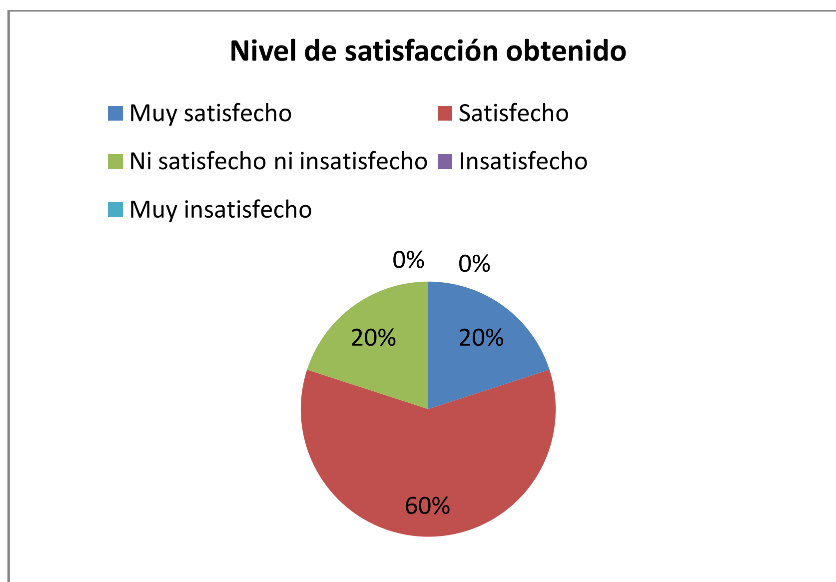


Figura 16 Niveles de satisfacción del paquete obtenidos en el cuestionario de evaluación.

El nivel de satisfacción predominante de la muestra es ***Satisfecho*** con un 60%, seguido por ***Muy satisfecho*** y ***Ni satisfecho ni insatisfecho*** ambos con un 20%. Lo que en general representa un resultado positivo para al paquete (Ver Figura 16).

Como parte de la evaluación de la lectura crítica, se les pidió a los participantes su opinión con respecto al paquete y se obtuvieron las siguientes respuestas:

1.- Quitar manejo de pruebas o sólo mencionarlo breve es esto y sirve para esto; y ampliar introducción de paquete de prueba un poco, la presentación está bien pero resulta impactante ver el número de páginas al inicio
2.- La sección de integración creo que está de sobra
3.- Me parece excelente el paquete. Al inicio un poco aburrido, pero después se vuelve muy interesante.
4.- En general bueno y claro de entender, aunque un poco confusa la parte de "6.5 Diseño de casos de prueba, escritura de pruebas unitarias y uso de una herramienta de cobertura de código", a través de los diferentes ejemplos de cobertura a veces perdía el hilo o la secuencia del ejemplo (aunque hay que tener en cuenta que solo lo ley una vez) .
5.- Es un paquete de fácil lectura que te guía de una forma ordenada a través de las distintas fases de desarrollo

También se preguntó por errores encontrados en el paquete y se propuso una posible solución para cada paso.

1.- Pág. 41, número 8 (ver numeración). No es correcto definir a JavaDoc como una "herramienta de comentarios automáticos".
2.- Tabla 10, la descripción viene en inglés
3.- En la parte que se hace referencia a los ejemplos, facilitaría ir al ejemplo, si se pone el número de la página de éste.

En cuanto al nivel de apreciación lectora se le pidió a los participantes que calificaran el formato, redacción y presentación del paquete y se obtuvieron los resultados de la Figura 17,

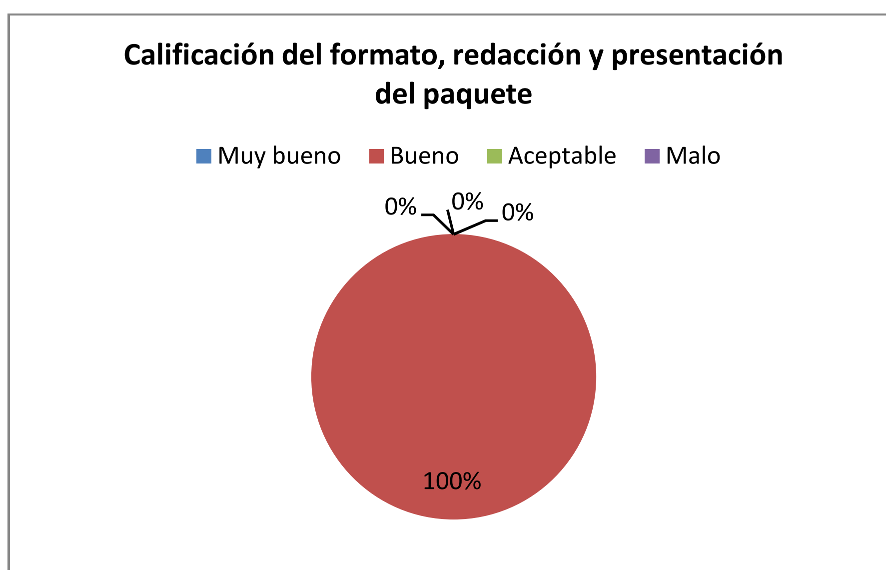


Figura 17 Calificación del formato, redacción y presentación del paquete obtenidos en el cuestionario de evaluación

Todos los participantes (100%) calificaron como ***Bueno*** estos atributos del paquete. Lo que en general representa un resultado positivo para al paquete.

6. Prácticas y actividades adicionales de apoyo al Paquete

Existen algunas otras prácticas y actividades que pueden mejorar la productividad de las tareas de construcción y pruebas unitarias. Sin embargo con el objetivo de no saturar de información el Paquete y/o confundir al lector, se optó por no incluirlas en la nueva versión.

Aún así se considera importante su lectura por lo que se decidió dejarlas como prácticas adicionales de apoyo al paquete, mismas que en un futuro podrían ser retomadas para realizar una versión extendida del paquete.

6.1 Técnica de Prevención de Defectos: FMEA – *Failure Modes and Effects Analysis*

FMEA es una técnica de prevención de defectos usada para identificar modos de fallo potenciales en el diseño de un producto, evaluar el riesgo de cada fallo potencial e implementar las acciones apropiadas para eliminar o mitigar aquellos modos de fallo (Marc & Robert, 2008) . Una vez identificados, esta información puede ser usada en proyectos futuros para evitar defectos.

Paso 1: Identificar y Describir el foco del área del producto objetivo
El foco del área para un producto de software puede ser la fase del ciclo de vida que se desea analizar, tales como requerimientos, diseño, construcción, pruebas, instalación, etc.

Paso 2: Crear una hoja de trabajo para el FMEA e introducir datos iniciales
El FMEA puede tener una estructura como la que se muestra en la figura 18.

Hoja de trabajo FMEA											
Proyecto: <Nombre del proyecto>											
Paso del proceso, Función o Tarea	Modo de Fallo Potencial	Efecto(s) potencial(es) del Fallo	Rango de Impacto	Causas Potenciales	Rango de Probabilidad	Controles actuales	Rango del defecto	Prioridad del Riesgo	Mejora	Propietario

Figura 18 Estructura básica de la hoja de trabajo para un FMEA

El foco del área se descompone por funcionalidad, escenarios, flujo lógico, tecnología, componentes, interfaces, etc. Como una analogía, si el foco del área es un proceso, durante la descomposición procesos individuales son identificados.

Paso 3: Determinar los modos de Fallo y añadirlos a la hoja de trabajo del FMEA

Un modo de fallo es un tipo de fallo que puede ocurrir. En sistemas de software, esto es evidenciado con síntomas tales como las pantallas azules, caídas del sistema, salidas incorrectas y corrupción de datos.

Identificando Modos de Fallo

Los modos de fallo potenciales pueden ser identificados por diversas fuentes, algunas de ellas son:

- Lluvia de ideas: Toma ventaja de la experiencia y conocimiento del equipo.
- Taxonomía de defectos: Usa el paso de "Escribir donde fue encontrado el defecto" para generalizar los modos de fallo potenciales

Paso 4: Califica el Impacto del Modo de Fallo, su probabilidad y su detectabilidad

Cada modo de fallo es analizado para determinar el impacto del peor caso, la probabilidad de las causas y la detectabilidad de los controles actuales.

Determinando efectos y evaluando el impacto potencial

Para cada efecto de los modos de fallo, el impacto potencial es estimado y cuantificado usando una escala como la que se muestra en la Tabla 21.

Rango del impacto	Criterio	Valor
Catastrófico	Todo el producto de software o sistema deja de trabajar apropiadamente (por ejemplo, el sistema se cae).	5
Crítico	Un área específica del producto de software deja de trabajar apropiadamente (por ejemplo, la red).	4
Moderado	Una parte específica del producto de software deja de trabajar apropiadamente (por ejemplo una característica de la red)	3
Menor	El producto de software trabaja apropiadamente pero tiene detalles menores, usabilidad etc.	2
Ninguno	Sin impacto	1

Tabla 21 Rango del impacto del fallo y criterio asociado

Determinando causas potenciales y evaluando la probabilidad

Para cada causa potencial, la probabilidad es estimada y cuantificada usando una escala como el que se muestra en la Tabla 22.

Rango de probabilidad	Criterio	Valor
Muy alto	> 1 en 2 (50%)	5
Alto	1 en 10 (10%)	4

Moderado	1 en 100 (1%)	3
Bajo	1 en 1,000 (0.1%)	2
Muy bajo	< 1 ien 10,000 (0.01%)	1

Tabla 22 Rango de probabilidad del fallo y criterio asociado

<u>*Determinando los controles actuales y evaluando la detectabilidad*</u>

Un modo de fallo que no puede ser detectado puede presentar un riesgo mayor que aquellos que pueden ser detectados. Si un modo de fallo puede ser detectado, al menos existe la oportunidad de eliminarlo o mitigarlo antes de que el producto o servicio se vea afectado (Ver Tabla 23)

Nota: Un control actual es la forma actual usada para detectar los errores.

Rango de Detectabilidad	Criterio	Valor
Muy bajo	Probabilidad muy baja (0-19%) de que los controles actuales detecten la causa	5
Bajo	Probabilidad baja (20-39%) de que los controles actuales detecten la causa	4
Medio	Probabilidad media (40-59%) de que los controles actuales detecten la causa	3
Alto	Probabilidad alta (60-79%) de que los controles actuales detecten la causa	2
Muy Alto	Probabilidad muy alta (80-100%) de que los controles actuales detecten la causa	1

Tabla 23 Rango de detectabilidad del fallo y criterio asociado.

Paso 5: Calcular el número de prioridad del riesgo para cada Modo de Fallo

El número de prioridad del riesgo (NPR) es un cálculo sencillo que consiste en el producto del valor del impacto, probabilidad y detectabilidad.

*NPR = Valor del impacto * Valor de la probabilidad * Valor de la detectabilidad*

Los modos de fallo con pocos o ningún riesgo pueden volverse de baja prioridad y se pueden atender sólo si los recursos y el tiempo lo permiten

Paso 6: Identificar los modos de fallo con los riesgos potenciales más elevados

Ordena las filas de la hoja de trabajo en orden descendiente basado en el número de prioridad del riesgo calculado. Los modos de fallo con los números de riesgo más altos usualmente representan una prioridad para el equipo.

Una optimización del análisis es realizar un ordenamiento primario basado en el número de prioridad del riesgo y un ordenamiento secundario basado en el valor de la probabilidad. Haciendo eso, el equipo puede enfocar sus recursos en eliminar las causas frecuentes de los modos de fallo de alto riesgo.

Paso 7: Definir un plan de acción para eliminar o mitigar las causas
Empezando con los modos de fallo de alto riesgo, hay que definir planes de acción para eliminar o mitigar las causas. Asignar acciones a los propietarios y fechas objetivo para cada acción.

Paso 8: Revalorar la prioridad de los riesgos después de que las acciones son implementadas

- Asumiendo que las acciones en realidad mitigan el impacto, reducen la probabilidad de una causa, o incrementan la detectabilidad, los rangos pueden cambiar y resultar en una reducción del número de prioridad de riesgos para el modo de fallo.
- Re-ordena la hoja del FMEA en orden descendente basado en el número de prioridad de riesgo para identificar los nuevos modos de fallo con los riesgos potenciales más altos.
- Repetir la ejecución de este ciclo continuamente puede ayudar al equipo a mejorar el producto.

Paso 9: Mantenimiento del FMEA

La hoja de trabajo del FMEA puede ser actualizada conforme se realicen cambios o mejoras al producto. Debido a que en el desarrollo de software, las mismas fallas pueden ser introducidas frecuentemente, el FMEA trata de evitar repetir estos mismos errores que pudieran causar más fallas. Como soporte para las pruebas, los resultados del FMEA pueden proveer entradas iniciales al plan de pruebas y al desarrollo de los casos de prueba.

Un ejemplo completo de un FMEA es proporcionado en el Anexo 2.

6.2 Base de conocimientos del Desarrollador

La taxonomía de defectos puede ayudar para conocer los tipos de problemas que ocurren en un proyecto, pero sólo enciende la luz roja de algunos aspectos que se deben de cuidar y no brinda una posible solución si se enfrenta nuevamente con el problema.

Debido a esto resulta de gran utilidad una base de conocimientos para el desarrollador. Es decir un sitio virtual en donde los programadores puedan encontrar la solución a problemas comunes, ya que en ocasiones suele ocurrir que el problema de un programador también puede ser el de otro, y la misma solución puede aplicarse a ambos, por lo que una única solución podría ahorrar tiempo y esfuerzo.

Un posible medio para reportar las soluciones a problemas puede ser un Sistema Administrador de Contenidos (CMS – de sus siglas en inglés *Content Management System)* o un simple wiki que contenga algunos de los siguientes datos:

- Nombre del Defecto
- Breve descripción del problema
- Posible solución o ejemplo del problema resuelto
- Comentarios acerca de la solución

Algunas sugerencias que pueden ayudar a mejorarlo son:

- Los ejemplos son mejores que explicaciones largas y engorrosas
- Trata de ser práctico, utilizar el modelo Copy-paste (Ribaud & Laporte, Experience Management for Very Small Entities: Improving the Copy-paste Model, pág. 5)
- Si el tópico necesita más información, un link o alguna especificación del tópico puede ser de ayuda
- Los programadores generalmente son expertos en la búsqueda de información en Internet.

6.3 Reglas de oro para Pruebas unitarias

La siguiente información fue obtenida de la documentación del SEI: Jones 1986, Boehm y Papaccio 1988, Robert Grady de Hewlett Packard, y Code complete segunda edición.

- Los programadores inyectan más defectos cuando diseñan mientras codifican (4.6 defectos por hora)que cuando diseñan antes de codificar (2.0 defectos por hora).
- Alrededor de un 80% de re-trabajo evitable viene de un 20% de los defectos.
- Cada hora que se invierte en prevención de defectos reduce el tiempo de reparación de 3 a 10 horas.
- El costo relativo de corregir defectos durante pruebas es 15 veces más grande que durante el diseño.
- Realizar pruebas sin medir la cobertura de código generalmente ejercita sólo alrededor de un 55% del código.
- Alrededor de un 7% de la reparación de defectos inyecta accidentalmente un defecto nuevo.

7. Conclusiones

Resulta clara la importancia de los estándares en las pequeñas organizaciones de software. Sin embargo como se mencionó en este trabajo, su adopción no resulta nada sencilla debido a diversos factores, como falta de recursos, complejidad de los estándares y el tiempo que se necesita invertir en la capacitación e implementación de los mismos.

Por ello en esta tesis se pretendió atacar algunos de esos problemas al elaborar una herramienta que redujera la brecha en la adopción de uno de los estándares internacionales más recientes para la mejora de procesos de desarrollo de software en pequeñas organizaciones, el estándar ISO/IEC 29110. Esta herramienta que tiene como nombre *Paquete de Puesta en Operación de Construcción y Pruebas unitarias* se enfoca específicamente en dichas actividades del estándar, dejando el resto a otras herramientas que tienen el mismo objetivo pero que atacan áreas y actividades diferentes del estándar *(Deployment Packages).*

Para el desarrollo de este trabajo se plantearon en un inicio una serie de objetivos, de los cuales a continuación se resume la conclusión para cada uno de ellos:

1. Elaborar y someter a aprobación la nueva versión del paquete como reporte técnico del estándar internacional ISO/IEC 29110 Perfil Básico:
   Conclusión: La aprobación del paquete, cuyo nombre original en Inglés es "Construction and Unit Testing Deployment Package - version 0.5" por parte de los miembros del WG24 se encuentra en proceso. Sin embargo la mayoría de los miembros involucrados ya han dado su voto aprobatorio.

2. Traducir el paquete al idioma español
   Conclusión: El paquete fue traducido al idioma español y las evaluaciones fueron realizadas en México en el mismo idioma.

3. Crear un sitio web para montar las evaluaciones
   Conclusión: El sitio y el marco de trabajo para las evaluaciones fue terminado, sin embargo debido a algunos contratiempos con el servidor dedicado no fue posible la configuración del sitio en línea en el tiempo estimado.

4. Aplicar las evaluaciones a por lo menos dos VSE
   Conclusión: Debido al tiempo limitado y a la disponibilidad de la audiencia objetivo inicial se decidió hacer un reajuste en el alcance de las evaluaciones y realizarlas a un conjunto de estudiantes de Maestría en Ciencias e Ingeniería de la computación, los cuales cumplen también el perfil requerido en cuanto a la experiencia y conocimientos.

Con respecto al resultado de las evaluaciones es importante recordar que se plantearon objetivos a largo y a corto plazo. Esto con la finalidad de tener un alcance factible para la tesis y no generalizar sobre los resultados obtenidos.

En cuanto a los objetivos a largo plazo, las conclusiones son las siguientes:

- Se creó un marco de trabajo para las evaluaciones que consiste en un proceso de tres etapas:
    - Aplicación de un cuestionario de exploración de conocimientos y experiencia previa en las áreas del paquete
    - Entrega del PPO de Construcción y pruebas unitarias
    - Aplicación de un cuestionario de evaluación de la comprensión lectora del paquete

Este marco quedó definido de manera conceptual para futuras evaluaciones a conjuntos representativos de la audiencia objetivo. Sin embargo como se estableció previamente en los objetivos de ésta tesis dicho alcance quedó fuera de éste trabajo de investigación debido al tiempo y esfuerzo requerido para poder completarlo.

En cuanto a los objetivos a corto plazo, las conclusiones son las siguientes:

- El marco fue aplicado satisfactoriamente a un subconjunto de la audiencia objetivo, a través de una prueba piloto que consistió en la evaluación completa de 5 participantes.

De los resultados obtenidos sobresalen los siguientes puntos:

- Se obtuvo un promedio de 75% en la comprensión lectora cuantitativa de los participantes
- Un 20 % de los participantes quedó muy satisfecho con el contenido del paquete, otro 60% quedó satisfecho y otro 20% no quedo ni satisfecho ni insatisfecho.
- Un 100% calificó como bueno el formato del paquete
- En 5 de las 7 tareas del paquete se detectó un incremento en el nivel de conocimientos con respecto a la evaluación de exploración.

Sin embargo no se puede generalizar sobre la audiencia objetivo ya que la muestra es muy pequeña, por lo que estos resultados sólo proporcionan una primera aproximación de la evaluación del paquete por elementos representativos de la audiencia. Aún así estos datos pintan un buen panorama para el PPO de Construcción de pruebas unitarias, lo que no significa que quede libre de errores, omisiones y posibles mejoras que podrían realizarse más adelante.

Como trabajo futuro quedan abiertas las siguientes cuestiones:

- Obtener una muestra más significativa para aplicar el marco de evaluación del PPO de Construcción y pruebas unitarias y tener así un panorama más amplio sobre las necesidades del paquete.
- Realizar proyectos piloto en los que se ponga a prueba el PPO de Construcción y pruebas unitarias o algún otro de los *Deployment Packages* que se encuentran disponibles y analizar los resultados

- Establecer un marco para las precondiciones y postcondicones de los *Deployment Packages* para que la aplicación de más de una de estas herramientas a la vez dentro de un proyecto de software minimice el esfuerzo.
- Mejorar y traducir el resto de los paquetes actuales al idioma español.

8. Anexo 1 – Paquete de Puesta en Operación de Construcción y Pruebas Unitarias Perfil Básico (Versión 0.5)

Paquete de Puesta en Operación de Construcción y Pruebas Unitarias Perfil Básico

Notas:
Este documento es de propiedad intelectual para la organización del autor. Sin embargo, la información que contiene es de libre uso. La distribución total o parcial de este documento está autorizada para su uso no comercial siempre y cuando la siguiente nota sea mencionada:

© nivel 5

El uso comercial de éste documento queda estrictamente prohibido. Éste documento es distribuido con el objetivo de mejorar el intercambio de información técnica o científica.

El autor no hace garantía de ningún tipo, ya sea explícita o implícita, con respecto a cualquier asunto incluyendo, pero no limitado a, garantía de adaptabilidad de uso o comercialización, exclusividad, o resultados obtenidos del uso del material.

El proceso descrito en este Paquete de Puesta en Operación no pretende excluir o desaprobar el uso de procesos adicionales que las VSE puedan encontrar útiles.

Autores	JAVIER FLORES – Universidad Nacional Autónoma de México (UNAM), (México) ANA VAZQUEZ – Nivel 5th (México)
Editores	R. CHAMPAGNE – École de Technologie Supérieure (ETS), (Canada) C. Y. LAPORTE – École de technologie supérieure (ÉTS), (Canada)
Fecha de creación	29/06/2011
Última modificación	21/11/11

Versión	0.5

Historial de Versiones

Fecha (dd-mm-yyyy)	**Versión**	**Descripción**	**Autor**
29-06-2011	0.5	Creación del Documento	Javier Flores

Abreviaciones/Acrónimos

Abre./Acro.	**Definiciones**
PPO	*Paquete de Puesta en Operación (En inglés conocido como Deployment Package*) - Un conjunto de artefactos desarrollados para facilitar la implementación de un conjunto de prácticas, del marco seleccionado, en una VSE.
VSE	VSE - Very Small Entity – (Pequeña organización) una empresa, organización, departamento o proyecto de a lo más 25 personas.
VSEs	Very Small Entities – (Pequeñas organizaciones)
UI	Interfaz de usuario
LT	Líder Técnico
AN	Analista
DIS	Diseñador
PR	Programador
AP	Administrador del Proyecto

Tabla de Contenidos

1. Descripción Técnica 5
Propósito de éste documento 5
¿Por qué es importante la construcción de software y pruebas unitarias? 5
2. Definiciones 7
Términos Genéricos 7
Términos específicos 7
3. Su relación con ISO/IEC 29110 9
4. Descripción de Procesos, Actividades, Tareas, Pasos, Roles y Productos. 11
Sub-tarea: Definir estándares de construcción 11
Sub-tarea: Reportar taxonomía de defectos de etapas previas 13
Asignar tareas a los miembros del equipo de trabajo 16
Construir o actualizar componentes de software 21
Diseñar o actualizar casos de pruebas unitarias y aplicarlos 24
Corregir los defectos 31
Actualizar el registro de trazabilidad 33
Descripción de los roles 33
Descripción de productos 34
Descripción de artefactos 37
5. Plantillas 38
5.1 Plantilla de construcción para Java 38
6. Ejemplos 40
6.1 Ejemplo de un estándar genérico de construcción 40
6.2 Ejemplo de Pseudocódigo 44
6.3 Ejemplo de Diagrama de Flujo 45
6.4 Jerarquía de criterios de cobertura, del más débil al más fuerte 46
6.5 Diseño de casos de prueba, escritura de pruebas unitarias y uso de una herramienta de cobertura de código 48
6.6 Corregir los defectos 52
6.7 Ejemplos de Test runners 54
6.8 Ejemplo de una macro para aserciones en C++ 55
6.9 Ciclo de vida de las pruebas unitarias 56
7. Listas de verificación 57
Listas de verificación de tareas 57
7.1 Asignar tareas a los miembros del equipo de trabajo 57
7.2 Construir o actualizar los componentes de software 57
7.3 Diseñar o actualizar casos de pruebas unitarias y aplicarlos 57

7.4 Corregir los defectos 57
Listas de verificación de Soporte 58
7.5 Revisión de código 58
7.6 Lo que el arquitecto y el diseñador deben proveer 59
8. Herramientas 60
8.1 Matriz de trazabilidad 60
8.2 Herramientas de Cobertura de código 62
8.3 Marcos de trabajo para pruebas unitarias (Frameworks) 62
9. Referencias a otros Estándares y Modelos 63
ISO 9001 Matriz de Referencia 63
ISO/IEC 12207 Matriz de Referencia 64
CMMI Matriz de referencia 66
10. Referencias 68
11. Formulario de Evaluación 69

1. Descripción Técnica

Propósito de éste documento

Este Paquete de Puesta en Operación PPO (En inglés conocido como *Deployment Package - DP*) sirve de apoyo al *Perfil Básico* como está definido en el estándar ISO/IEC 29110 Parte 5-1-2: Guía de Ingeniería y Gestión. El perfil básico es un perfil del Grupo de Perfiles Genéricos. Éste grupo está compuesto de cuatro perfiles: Perfil de Entrada, Perfil Básico, Perfil Intermedio y Perfil Avanzado. El grupo es aplicable a las VSE que no desarrollan software crítico. El grupo de perfiles genéricos no implica ningún dominio de aplicaciones específicas.

Un PPO es un conjunto de artefactos desarrollados para facilitar la implementación de un conjunto de prácticas en las VSE. Un PPO no es un modelo de referencia de procesos (e.g no es prescriptivo o mandatorio). Los elementos de un PPO típico son: Descripción de procesos, actividades, tareas, roles, productos, plantillas, listas de verificación, ejemplos, referencias y referencias a estándares, modelos y herramientas.

El contenido de éste documento es meramente informativo.

Este documento ha sido producido por Javier Flores Flores (UNAM, México) y Ana Vázquez (Nivel número 5) más allá de su participación en el JTC1/SC7/WG24.

¿Por qué es importante la construcción de software y pruebas unitarias?

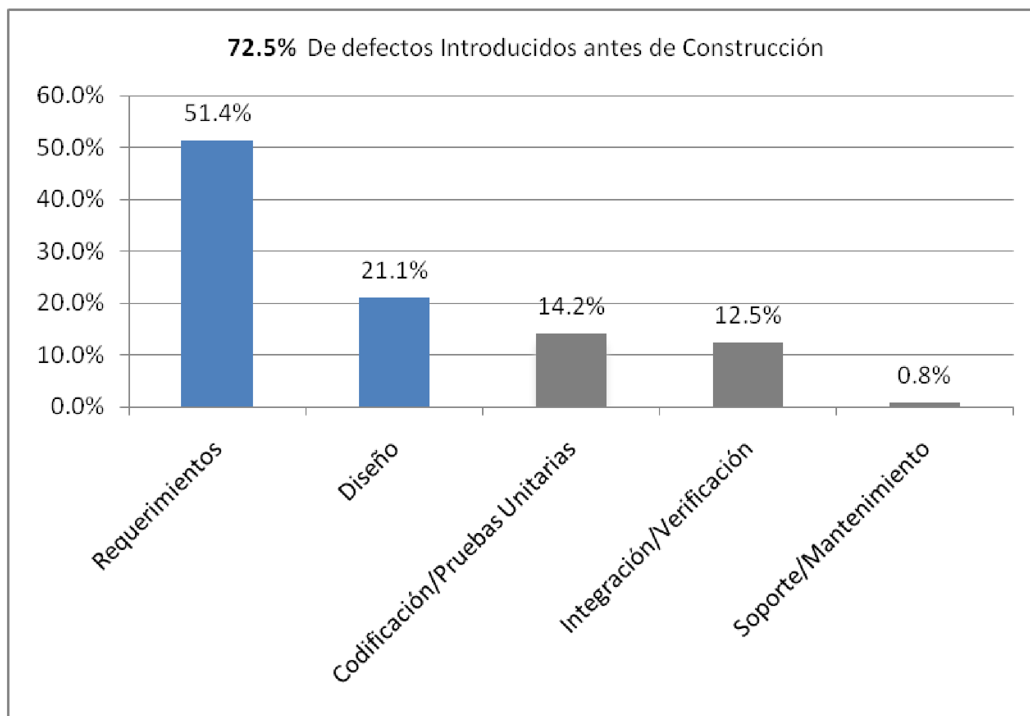
Todas las etapas en el desarrollo de software son importantes, pero la actividad central es la construcción de software y aquí hay algunas razones:

- En algunos proyectos reales las etapas de requerimientos y diseño no se realizan. Pero no importa que tan apretada se encuentre la agenda, ***la construcción de software siempre debe realizarse***.
- En muchos proyectos, ***la única documentación disponible para el programador es el código***. Las especificaciones de requerimientos y los documentos de diseño pueden estar desactualizados, pero el código fuente debe estar siempre al día.
- ***80% del costo de desarrollo es consumido por los programadores*** de software identificando y corrigiendo defectos. (NIST, Instituto Nacional de Estándares y Tecnología – USA).

Por otro lado las pruebas unitarias representan también una parte importante, ya que son el primer nivel en la jerarquía de pruebas y tan rápido como el programador desarrolla una unidad de código, el siguiente paso es ver si ésta funciona adecuadamente., y mientras más tarde se encuentren defectos más grande es el costo de repararlos (ver Tabla 1)

Defectos	Cuando son detectados				
Cuando son introducidos	**Requerimientos**	**Arquitectura**	**Construcción**	**Pruebas de sistema**	**Post entrega**
Requerimientos	1	3	5-10	10-100	10-100
Arquitectura	---	1	15	25-100	25-100
Construcción	---	---	1	10-25	10-25

Tabla 1 Representación del costo promedio de corregir defectos basados en cuando son introducidos y cuando son detectados. [Code Complete]



Gráfica 1. Orígenes de los defectos de software (Selby 2007[1]). Adecuación.

[1] Selby, P., Selby, R.W., Sistemas de Ingeniería orientados a Mediciones usando técnicas Six Sigma para mejorar la detección de defectos de software, Procedimientos del Simposio número 17, INCOSE, Junio 2007, San Diego.

2. Definiciones

En ésta sección el lector encontrará dos conjuntos de definiciones. El primer conjunto define los términos usados en todos los PPO, e.g Términos genéricos. El segundo conjunto define los términos usados en este PPO, e.g términos específicos.

Términos Genéricos

Proceso: Conjunto de actividades interrelacionadas que transforman entradas en salidas [ISO/IEC 12207].

Actividad: Un conjunto de tareas cohesivas de un proceso [ISO/IEC 12207].

Tarea: Acción requerida, recomendada o permisible que intenta contribuir al logro de una o más salidas de un proceso [ISO/IEC 12207].

Sub-Tarea: Cuando una tarea esta completa, es dividida en sub-tareas.

Paso: En un PPO, una tarea es descompuesta en una secuencia de pasos.

Rol: Una función definida para ser realizada por un miembro del equipo del proyecto, tales como pruebas, inspección, codificación [ISO/IEC 24765].

Producto: Pieza de información o entregable que puede ser producida (no de manera obligatoria) por una o más tareas. (e.g. documentación de diseño, código fuente).

Artefacto: Información, que no se menciona en el ISO/IEC 29110 Parte 5, pero que puede ayudar a la VSE durante la ejecución del proyecto.

Términos específicos

Componente: Conjunto de servicios funcionales de software que cuando son implementados, representan un conjunto de funciones bien definidas que son distinguibles por un único nombre [ISO/IEC 29881:2008].

Defecto: Un problema que si no es corregido puede causar que una aplicación falle o genere resultados incorrectos [ISO/IEC 20926].

Trazabilidad: Grado en que dos o más productos del procesos de desarrollo pueden tener alguna relación, especialmente productos que tienen una relación predecesor-sucesor o maestro-subordinado.

Prueba unitaria: Pruebas de rutinas y módulos individuales realizadas por el programador o por un tester independiente [ISO/IEC 24765]

Cobertura de Código: Medida usada en pruebas de software. Describe el grado en el que el código de un programa ha sido probado [Practical Software Testing]

3. Su relación con ISO/IEC 29110

Este Paquete de Puesta en Operación cubre las actividades relacionadas con la Construcción de software y pruebas unitarias del reporte técnico ISO/IEC 29110 Parte 5-1-2 para las VSE - Perfil Básico [ISO/IEC29110].

Las actividades de construcción deben ser planeadas durante la actividad de planeación del proyecto y deben ser descritas en el plan del proyecto. Si no es el caso, el administrador del proyecto debe realizar ésta actividad antes de empezar la construcción. (Ver el PPO de Administración de Proyectos).

En esta sección, el lector encontrará una lista de procesos de Implementación de Software, actividades, tareas y roles de la parte 5 que están directamente relacionados con este paquete.

- **Proceso:** SI - Implementación de Software
  - **Actividad:** SI.2[2] Análisis de Requerimientos de Software
    - **Tareas y Roles:**

Tareas	Roles[3]
SI.2.2 Documentar o actualizar la *especificación de Requerimientos.* Identificar y consultar fuentes de información (clientes, usuarios, sistemas previos, documentos, etc.) con el objetivo de obtener nuevos requerimientos. Analizar los requerimientos identificados para determinar el alcance y la factibilidad. Generar o actualizar la *Especificación de Requerimientos.*	AN, CLI

[2] Estos números hacen referencia a procesos, actividades y tareas de ISO/IEC 29110 Parte 5-1-2

[3] Los roles son definidos en la siguiente sección. Los roles también son definidos en ISO/IEC 29110 Parte 5-1-2

- **Proceso:** SI - Implementación de Software
    - **Actividad:** SI.4 Construcción de Software
        - **Tareas y Roles:**

Tareas	Roles
SI.4.1 Asignar tareas relacionadas con su rol a los miembros del equipo de trabajo, de acuerdo al *Plan del proyecto* actual.	LT, PR
SI.4.3 Construir o actualizar *Componentes de Software* basados en la parte detallada del *Diseño de Software.*	PR
SI.4.4 Diseñar y actualizar casos de pruebas unitarias y aplicarlas para verificar que el *Componente de Software* implementa la parte detallada del *Diseño de Software.*	PR
SI.4.5 Corregir los defectos encontrados hasta que las pruebas unitarias hayan pasado satisfactoriamente (alcanzando un criterio de salida).	PR
SI.4.6 Actualizar el registro de trazabilidad incorporando los componentes de software construidos o modificados.	PR

4. Descripción de Procesos, Actividades, Tareas, Pasos, Roles y Productos.

- **Proceso:** SI - Implementación de Software
  - **Actividad:** SI.2 Análisis de Requerimientos de Software
    - **Tareas y Roles:**

Tareas	Roles
SI.2.2 Documentar o actualizar la *especificación de Requerimientos.* Identificar y consultar fuentes de información (clientes, usuarios, sistemas previos, documentos, etc.) con el objetivo de obtener nuevos requerimientos. Analizar los requerimientos identificados para determinar el alcance y la factibilidad. Generar o actualizar la *Especificación de Requerimientos.*	AN, CLI

Esta tarea está relacionada con las siguientes sub-tareas:

- Definir estándares de construcción
- Reportar taxonomía de defectos de etapas previas

Sub-tarea: Definir estándares de construcción

Objetivo:	Servir de guía en la codificación de software para producir código fácil de mantener dentro o fuera del proyecto.
Justificación:	Los requerimientos de mantenimiento deben ser acordados como parte de la especificación de requerimientos. Aún generalmente estos no son acordados explícitamente, pero siempre son esperados. La falta de estándares de codificación será percibida por colegas cuando intenten modificar el código, dentro del equipo o fuera de él. Algunos componentes pueden llegar a ser reemplazados por nuevos por la falta de su comprensión, si el mantenimiento es realizado por el equipo de desarrollo, el costo del proyecto se incrementará, en cambio si es realizado por el cliente entonces él absorberá dicho costo.

	Los estándares de codificación deben ser usados en al menos aquellos componentes que realizan funcionalidades esenciales. Los estándares de codificación no están especificados de manera explícita en el 29110-5, sin embargo su uso puede incrementar la productividad del proyecto y la calidad del producto. *Nota: Un estándar general de construcción es proporcionado en la sección de ejemplos*
Roles:	Administrador del proyecto
	Líder técnico
	Programador
Artefactos:	Estándar de construcción
Pasos:	1. Planear la sub-tarea
	2. Obtener estándares disponibles
	3. Seleccionar los estándares
	4. Adoptar los estándares
	5. Verificar la adopción de los estándares
Descripción de los pasos:	***Paso 1. Planear la sub-tarea*** De acuerdo al progreso del proyecto, el administrador del proyecto decide si incluye en el plan estos pasos. La asignación del esfuerzo es muy importante porque la definición de estándar puede no tener un fin adecuado, y si el estándar no es adoptado entonces puede resultar inútil. Con lo que respecta al estándar, es deseable tenerlos listos antes de empezar la construcción, aún así esto no es siempre posible. ***Paso 2. Obtener estándares disponibles*** Verifica si tu cliente cuenta con estándares de construcción, en caso contrario búscalos en Internet o en otras fuentes disponibles. Evita definir los estándares desde cero, en la mayoría de los proyectos esto se encuentra fuera del alcance, crearlos puede tomar mucho tiempo y esfuerzo. ***Paso 3. Seleccionar los estándares*** Si tu cliente no posee estándares entonces pide a los programadores que seleccionen alguno de los encontrados, o una combinación de ellos. Independientemente si se consigue o no un estándar, la manera más fácil de lidiar con ello es implementar algunos componentes y usarlos como ejemplos, si se tiene el tiempo suficiente, entonces

	se puede crear un estándar de construcción propio. *Nota 1: Un estándar general de codificación es proporcionado en la sección de ejemplos.* *Nota 2: Una plantilla de construcción para Java es proporcionada en la sección de plantillas* ***Paso 4. Adoptar los estándares*** Pide a los programadores que adopten los estándares de construcción de ahora en adelante, especialmente en aquellos componentes que desempeñan funcionalidades esenciales. ***Paso 5. Verificar la adopción de los estándares*** A través de otro programador verifica la adopción de los estándares de construcción en al menos aquellos componentes que desempeñan funcionalidades esenciales.

Sub-tarea: Reportar taxonomía de defectos de etapas previas

Objetivo:	Reportar defectos de etapas previas al área correspondiente
Justificación:	Como se muestra en la gráfica 1 más de un 70% de los defectos son introducidos antes de la construcción del software. Por lo tanto existe una gran probabilidad de encontrar defectos en etapas previas a la etapa de Construcción. Por ello la mejor estrategia es reportar los defectos al área encargada antes de que las correcciones se vuelvan más costosas. Esta tarea no está especificada en el 29110-5, aún así el invertir esfuerzo en ella puede ayudar a incrementar la productividad del proyecto y la calidad del producto.
Roles:	Líder Técnico
	Programador
Artefactos:	Taxonomía de defectos
Pasos:	1. Confirmar el defecto
(Programador)	2. Escribir una descripción breve del defecto
	3. Escribir dónde fue encontrado el defecto
	4. Escribir las posibles causas
	5. Escribir la extensión del daño
Pasos:	1. Verificar el reporte del defecto

<table>
<tr><td rowspan="2">(Líder Técnico)</td><td>2. Escribir una estrategia de mitigación</td></tr>
<tr><td>3. Reportar el defecto al área encargada</td></tr>
<tr><td>Descripción de los pasos:

(Programador)</td><td>Paso 1. Confirmar el defecto

Si encuentras inconsistencias en el diseño detallado, la arquitectura de software o una parte de los requerimientos, explica los detalles a tu Líder Técnico.

Si tu Líder Técnico está de acuerdo en que lo que encontraste es un defecto, continúa con los siguientes pasos, de otra forma abandona la sub-tarea. Recuerda que el no reportar un defecto en el momento apropiado significa que después de codificar, en algún momento tendrás que realizar los cambios de cualquier forma, y la mayoría de los casos son más costosos.

Paso 2. Escribir una descripción breve del defecto

Escribe una descripción breve del defecto, incluyendo cómo encontraste el defecto y bajo qué circunstancias lo hallaste.

Paso 3. Escribir dónde fue encontrado el defecto

Escribir dónde fue encontrado el defecto dentro del ciclo de vida
• Requerimientos
• Arquitectura de Software
• Diseño Detallado

Paso 4. Escribir las posibles causas

Al escribir información sobre la causa del defecto es posible encontrar indicios de la raíz del problema

• Factores sistemáticos: Guías y procedimientos; cultura organizacional; información de dominio específico tales como documentación, código, herramientas, etc.
• Factores humanos: Una persona comete errores por razones humanas: Omisión, falta de aplicación, distracción, errores de búsqueda, soluciones incorrectas, etc.
• Desconocido

Paso 5. Escribir la extensión del daño

Cuál es el efecto de propagación

• Función
• Objeto
• Proceso</td></tr>
</table>

	• Compatibilidad • Aplicación • Máquina • Servidor • Cliente • Red • Otro
Descripción de los pasos: ***(Líder Técnico)***	***Paso 1.*** **Verificar el reporte del defecto** Verificar el reporte realizado por el programador. Si hay alguna inconsistencia pedirle al programador que clarifique los detalles. ***Paso 2.*** **Escribir una estrategia de mitigación** Si conoce alguna estrategia de mitigación para el defecto encontrado, añádalo al reporte. Algunas estrategias de mitigación pueden ser: **Tipo de Mitigación:** • Nueva herramienta • Hardware • Entrenamiento • Administración del personal • Comunicación (grupos/individuos) • Acceso al conocimiento • Cambio de proceso ***Paso 3.*** **Reportar el defecto al área encargada** Una vez que el reporte este completo envíalo al área encargada y si es posible, reasigna las tareas del programador que encontró el defecto mientras el área correspondiente responde a la solicitud.

- **Proceso:** SI - Implementación de Software
  - **Actividad:** SI.4 Construcción de Software
    - **Tareas y Roles:**

Tareas	Roles
SI.4.1 Asignar tareas relacionadas con su rol a los miembros del equipo de trabajo, de acuerdo al *Plan del proyecto* actual.	LT, PR
SI.4.3 Construir o actualizar *Componentes de Software* basados en la parte detallada del *Diseño de Software.*	PR
SI.4.4 Diseñar y actualizar casos de pruebas unitarias y aplicarlas para verificar que el *Componente de Software* implementa la parte detallada del *Diseño de Software.*	PR
SI.4.5 Corregir los defectos encontrados hasta que las pruebas unitarias hayan pasado satisfactoriamente (alcanzando un criterio de salida).	PR
SI.4.6 Actualizar el registro de trazabilidad incorporando los componentes de software construidos o modificados.	PR

Asignar tareas a los miembros del equipo de trabajo

Nota: Las tareas están ligadas a un rol, de acuerdo al Plan del Proyecto

Objetivo:	Definir la secuencia de integración y asignar tareas a los miembros del equipo de trabajo.
Justificación:	Generalmente en la primera versión del Plan del proyecto se han identificado la mayoría de los componentes a construir, aún así en ésta etapa del proyecto, cuando la Arquitectura de Software y el Diseño Detallado están completos, el Plan del Proyecto debe ser actualizado para incluir la construcción y pruebas unitarias, ya sea en detalle o de manera general, de todos los componentes que deben de ser producidos. La secuencia de construcción de los componentes debe ser coordinada con la secuencia de integración para tener los componentes (ya probados de manera unitaria) listos para ser integrados en el momento correcto. La definición de la secuencia de construcción no se encuentra

	directamente en el 29110-5, aún así dedicar cierto esfuerzo para definirla puede ayudar a optimizar el calendario de construcción.
Roles:	Administrador del proyecto
	Líder Técnico
Productos:	Plan del Proyecto (Actualizado)
Pasos:	1. Obtener la documentación del Diseño de Software
	2. Seleccionar la estrategia de integración
	3. Detallar el calendario del proyecto
	4. Asignar tareas a los miembros del equipo de trabajo.
	5. Definir un criterio de salida para las pruebas unitarias
Descripción de Pasos:	***Paso 1. Obtener la documentación del Diseño de Software*** • Obtener la documentación del diseño de software del repositorio del proyecto, el diseño de software detallado de bajo nivel incluye el detalle de los componentes de software a construir. • Obtener el registro de trazabilidad del repositorio. ***Paso 2. Seleccionar la estrategia de integración*** Existen diferentes estrategias para determinar la secuencia de integración, pero éstas son más heurísticas que algorítmicas. Los enfoques de integración más ampliamente aceptados son: - Big bang: Integra todas las partes al mismo tiempo, de tal forma que el software en su totalidad es ensamblado y probado en un solo paso (éste enfoque puede ser muy riesgoso debido al nivel de entropía introducido). - Bottom-up: Los módulos de los niveles inferiores son integrados primero, después la integración continúa desplazándose hacia arriba a través de la estructura de control. Éste proceso es repetido hasta que el componente que se encuentra en el tope de la jerarquía es integrado. -Top-down: Los módulos son integrados desplazándose hacia abajo a través de la estructura de control. Éste proceso se repite hasta que los componentes que se encuentran hasta debajo de la jerarquía son integrados. *Nota: Implementar los enfoques de Bottom-up o Top-down en su forma pura en algunas ocasiones es radical. Una mejor opción es escoger un enfoque híbrido*

Algunos enfoques híbridos son:

Integración orientada a Riesgos:

La integración orientada a riesgos pretende integrar los componentes que se encuentran en el tope (clases de alto nivel) y en el nivel más bajo (interfaces de dispositivos y clases de utilería) primero, dejando los componentes de nivel medio al último. La idea es priorizar de acuerdo al nivel de riesgo asociado a cada componente.

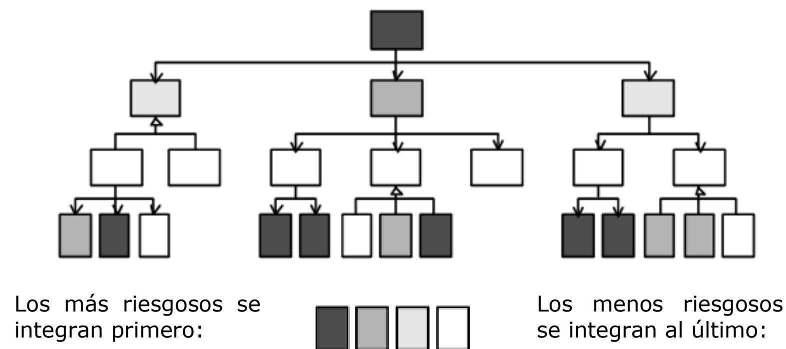


Figura 2 Estrategia de integración orientada a riesgos. [Code complete]

Integración orientada a Características

La idea es integrar clases en grupos que desempeñen características identificables. Cada característica integrada trae consigo un incremento adicional a la funcionalidad. Una ventaja de esta estrategia es que se adapta fácilmente con el diseño orientado a objetos.

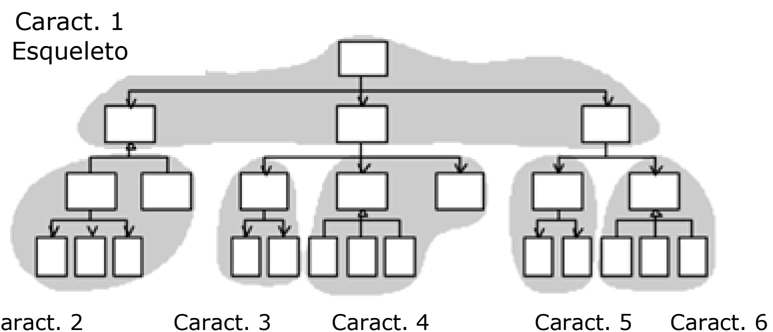


Figura 3 Estrategia de Integración orientada a características. [Code complete]

Una vez analizadas las estrategias, al final es tu decisión si usas una combinación de dos o más enfoques o una adaptación específica para tu proyecto.

	Paso 3. Detallar el calendario del proyecto
	
	Detalla el calendario del proyecto incluyendo las actividades para desarrollar o modificar todos los componentes identificados, de acuerdo a una estrategia de integración seleccionada.
	
	Haz todo lo posible para mantener los acuerdos relacionados con el costo y tiempo asignados al proyecto, si es necesario modificarlos, entonces es necesario tramitar una solicitud de cambios. Para mayores detalles sobre el calendario del proyecto, revisa el PPO de Administración de Proyectos *(Project Management Deployment Package).*
	
	Paso 4. Asignar tareas a los miembros del equipo de trabajo
	
	• Asignar tareas a los miembros del equipo de trabajo
	  o Asignar tareas para la codificación de los componentes de software
	  o Asignar tareas para desarrollar los casos de prueba y la prueba de los componentes de software
	• Informar a los miembros sus tareas respectivas
	
	Para asignar las tareas a la gente apropiada existen algunos aspectos sobe los programadores que debes tomar en cuenta. Siempre pregunta los antecedentes de tu equipo:
	
	• Experiencia
	• Habilidades
	• Conocimiento
	
	Esto te ayudará a tener a la persona indicada en la actividad adecuada. También toma en cuenta las siguientes consideraciones:
	
	• Siempre trata de asignar los componentes y tareas más críticas a las personas con mayor experiencia.
	• Deja los componentes y actividades más sencillas a los programadores novatos; eventualmente se convertirán en programadores expertos
	• Trata de identificar las habilidades de cada programador y saca provecho de ello.
	
	6. Definir el criterio de salida para las pruebas unitarias
	
	Probar todos los casos posibles de un componente no es factible incluso para proyectos pequeños. Por lo tanto debes decidir un criterio de salida que les diga a tus programadores cuando deben de dejar de hacer pruebas o cuando las pruebas son suficientes para un

determinado tipo de componente.

Un buen comienzo es definir un porcentaje para algún criterio de cobertura de código (ver paso1 de la tarea "Diseñar o actualizar casos de pruebas unitarias y aplicarlos") dependiendo del riesgo del código bajo prueba. Este puede ser dividido en tres categorías:

Tipo de Componente	Descripción
Alto riesgo	El código puede causar daño severo (borrar información, herir a alguien, dar respuestas erróneas poco obvias que puedan costarle al usuario mucho dinero), o tener demasiados usuarios (el costo de incluso defectos menores se multiplicaría), o es probable que tenga muchos errores que en su conjunto sumen un alto costo (Un algoritmo mal diseñado, una interfaz pobre o poco entendible, o simplemente has encontrado un numero inusual de problemas).
Mediano riesgo	Los defectos encontrados individualmente no son necesariamente críticos, pero el tener muchos de ellos puede causar un desplazamiento en el calendario. Existe una buena razón para corregirlos lo más pronto y con el menor costo posible. Sin embargo se debe considerar que el esfuerzo invertido en hacer un análisis más profundo puede ser usado en otras tareas. Esta decisión depende del Administrador del proyecto o del Líder técnico.
Bajo riesgo	Es poco probable que el código tenga defectos lo suficientemente importantes para detener o retrasar una entrega, aun considerando todos los defectos. Este tipo de defectos representan una molestia en funcionalidades no esenciales, y tienen soluciones sencillas y obvias.

Tabla 2 Categorías del código bajo prueba. [How to Misuse code coverage]

No existe un algoritmo que pueda decirte que categoría le corresponde a un componente. En algunas ocasiones, algún código de mediano riesgo puede ser de alto riesgo con respecto a cierto tipo de fallas y lo mismo puede ocurrir con el código de bajo riesgo. Por lo tanto es necesario analizar el impacto de cada componente y elegir adecuadamente.

Una vez que se ha definido el riesgo del componente es conveniente usar un criterio de cobertura de código más fuerte para probar componentes con un alto riesgo.

Tipo de componente	Criterio de cobertura recomendado

Alto riesgo	Decisión-Condición
Medio riesgo	Decisión
Bajo riesgo	Sentencia

Table 3 Tipo de cobertura recomendado para cada tipo de componente

Una porcentaje de cobertura de código comúnmente seleccionado por algunas compañías de prestigio es 85% (Marick 1999). Sin embargo no siempre es la cantidad correcta ya que cada proyecto posee restricciones diferentes. La elección de un porcentaje adecuado depende de diferentes circunstancias; como la importancia del proyecto, el calendario y los recursos estimados.

Alcanzar el 100% de cobertura de código generalmente resulta muy difícil y en algunas ocasiones el esfuerzo para alcanzarlo resulta muy costoso. Aún así el obtener un porcentaje elevado puede darte cierta seguridad de que tu componente trabaja apropiadamente aún si no está completamente probado.

Construir o actualizar componentes de software

Nota: Los componentes están basados en la parte detallada del diseño de software.

Objetivo:	Producir los componentes de software como fueron diseñados
Justificación:	La calidad en la construcción afecta substancialmente a la calidad del software y la mejor manera de incrementar la calidad es mejorando las prácticas y técnicas de los programadores. Es usual que los programadores se sientan confiados de producir componentes sin la necesidad de un enfoque sistemático. Sin embargo estos enfoques siguen siendo útiles para la construcción de componentes complejos o críticos. Existen diferentes enfoques para la generación de componentes, en este documento son usados dos de los enfoques más ampliamente aceptados, el enfoque de Pseudocódigo y el de Diagrama de flujo *Nota: En algunas ocasiones existen inconsistencias que vienen de etapas previas (Requerimientos, Diseño detallado). Si éste es el caso, la mejor opción es reportarlos antes de que se construya algún componente inadecuado. (Ver sub-tarea "reportar taxonomía de defectos de etapas previas").*
Roles:	Programador
Productos:	Componentes de Software
Pasos:	1. Entender el diseño detallado del componente y su contribución

<table>
<tr><td></td><td>2. Buscar funcionalidades disponibles en las librerías estándar.</td></tr>
<tr><td></td><td>3. Definir la lógica del componente</td></tr>
<tr><td></td><td>4. Codificar el componente de acuerdo al estándar de construcción.</td></tr>
<tr><td></td><td>5. Verificar el componente</td></tr>
<tr><td>Descripción de los pasos:</td><td>Paso 1. Entender el diseño detallado del componente y su contribución

a) Verificar la contribución

Verifica la contribución del componente y ve si la mejor opción para solucionar el problema es la creación de un nuevo componente o la adaptación de uno previo.

b) Detalle suficiente

Si el diseño detallado de software no está lo suficientemente claro el Programador puede complementarlo con la ayuda del Diseñador.
El Diseño Detallado de Software debe incluir detalles del componente para facilitar la construcción y pruebas dentro del ambiente de desarrollo:
- Provee un diseño detallado (Diagramas de clase, diagramas de actividad, diagrama entidad-relación, etc.)
- Especifica el formato de entrada/salida de los datos
- Provee la especificación de las necesidades de almacenamiento de datos
- Define el formato de las estructuras de datos requeridas
- Define los campos de datos y el propósito de cada elemento de datos requerido.

Paso 2. Buscar funcionalidades disponibles en las librerías estándar.

Verifica si alguna o todas las funcionalidades del componente ya se encuentran disponibles en las librerías del lenguaje, plataforma o herramientas que estás usando. Mejora tu productividad reutilizando código. Muchos algoritmos ya han sido creados, probados y mejorados, así que no inventes la rueda, sólo úsala.

Paso 3. Definir la lógica del componente

Existen diferentes enfoques para la creación de componentes, dos de los más ampliamente aceptados son el enfoque de</td></tr>
</table>

	Pseudocódigo y el de Diagrama de Flujo. Los pasos fueron adaptados de Code Complete (Ver referencia). ***a) Proceso de programación por Pseudocódigo*** Uno de las mejores formas de abstraer una idea es haciendo un simple dibujo de ella a un nivel muy elevado. Después refina la idea y empieza a escribir algunas líneas de pseudocódigo que describan una posible solución a un nivel medio (en tu lenguaje natural, e.g Español), después continúa refinando el pseudocódigo hasta que alcances un nivel de diseño de bajo nivel (Muy cercano al código fuente). De tal forma que convertir el pseudocódigo en código fuente resulte sencillo. Siempre trata tantas ideas como puedas en el pseudocódigo antes de que empieces a codificar. Porque una vez que empieces a escribir código, generalmente te involucras emocionalmente con tu código y se vuelve cada vez más difícil desechar un mal diseño y empezar de nuevo. *Nota: Un ejemplo de Pseudocódigo es proporcionado en la sección de ejemplos.* **b) Diagrama de Flujo** Otra forma de abstraer una idea es haciendo un gráfico o alguna representación simbólica del algoritmo que tienes en mente. Cada paso en el algoritmo es presentado por un símbolo y contiene una descripción corta del proceso. Los símbolos del Diagrama de flujo se encuentras ligados uno con otro por medio de flechas que muestran la dirección del flujo del algoritmo. Puedes empezar definiendo un estado inicial e ir añadiendo pasos, decisiones o ciclos al proceso hasta que alcances el final del algoritmo, Después puedes refinar cada paso tanto como sea necesario hasta que convertirlo en código fuente resulte sencillo. *Nota: Un ejemplo de Diagrama de Flujo es proporcionado en la sección de ejemplos.* ***Paso 4. Codificar el componente de acuerdo al estándar de construcción.*** Codificar puede ser un proceso mecánico una vez que el algoritmo está listo. Enfocándote más en aspectos de configuración y del lenguaje que en la lógica del componente. Algunas partes del trabajo previo pueden ser reusables, por ejemplo si usas el enfoque de Pseudocódigo, puedes usar el pseudocódigo de alto nivel como comentarios y el de bajo nivel para codificar la lógica del componente.

	Para éste tipo de aspectos es importante seguir el estándar de construcción cuando se codifica, porque no sólo hace que tu código sea más limpio, sino que se vuelve un código estandarizado y fácil de mantener (*ver la sub-tarea "Definir estándares de construcción").* Si tu componente necesita de una interfaz, lo recomendable es codificarla de acuerdo a un estándar de Interfaz. Si dicho estándar no existe o no lo conoces, solicítalo a tu líder técnico (Ver *la lista de verificación "Definir el estándar de interfaz").* **Verificar si el código necesita ser dividido** Puedes considerar la creación de una subrutina si: • Algunos de los pasos en el proceso son muy repetitivos y en vez de copiar y pegar grandes porciones de código sólo haces la llamada a la subrutina apropiada que realiza el trabajo. • Algunos pasos en el algoritmo son partes de un sub-proceso independiente que puede ser usado por otro componente. Crear una subrutina para que realice el trabajo no cambia la lógica, pero hace que el código sea más claro y fácil de mantener. ***Paso 5. Verificar el componente*** La revisión de código es una inspección sistemática en la que los desarrolladores de software revisan su código con el objetivo de encontrar y corregir defectos superficiales al inicio de la fase de desarrollo, mejorando la calidad del código fuente y las habilidades del programador. *Nota: En la sección de listas de verificación se proporciona una lista para revisión de código.* **Compilar el código:** Deja que el compilador revise variables no declaradas, conflictos en el nombrado y cosas similares.

Diseñar o actualizar casos de pruebas unitarias y aplicarlos

Objetivo:	Encontrar y corregir defectos introducidos durante la construcción a través del diseño y aplicación de casos de pruebas unitarias.
Justificación:	Existen dos enfoques fundamentales para las pruebas de software, llamados: de caja negra y de caja blanca. Las pruebas de caja negra prueban los requerimientos funcionales, sin necesidad del

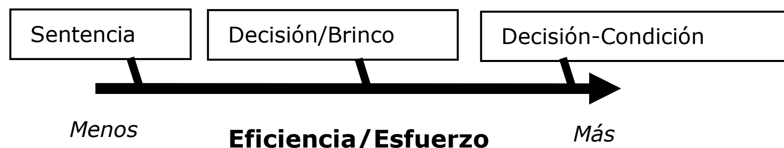
	conocimiento de la estructura interna del componente. Por su parte las pruebas de caja blanca prueban la estructura interna y la lógica del componente. Estos componentes o unidades son las piezas más pequeñas del software y en la mayoría de los casos resulta más fácil y menos costoso buscar defectos dentro ellas que en todo el sistema. Debido a esto es importante que las pruebas unitarias sean realizadas tan pronto como las unidades se encuentren listas. Aún si no es siempre posible, es altamente recomendable que los programadores realicen las pruebas de caja blanca, ya que conocen el código y la estructura de los componentes que ellos mismos codifican. Por ello, para ser prácticos en este paquete, se asume esa última sugerencia y el enfoque seleccionado son las pruebas de caja blanca. Un beneficio importante de las pruebas unitarias es que permiten trabajar con cierto nivel de paralelismo, permitiendo probar y depurar simultáneamente. Además con la ayuda de herramientas y marcos de trabajo, la eficiencia de estas pruebas puede ser mejorada considerablemente. *Nota: Las pruebas unitarias son útiles para todo tipo de software. Sin embargo el software implementado usando un lenguaje orientado a objetos introduce consideraciones adicionales que deben ser tomas en cuenta cuando se prueba dicho tipo de software. Para mayor información al respecto puedes consultar [Introduction to Software Testing]* (ver referencias).
Roles:	Programador
	Administrador del proyecto
Productos:	Componentes de Software [probados]
Artefactos:	Casos de prueba, Conjunto de pruebas unitarias
Pasos:	1. Obtener el criterio de salida
	2. Diseñar los casos de prueba
	3. Codificar las pruebas unitarias
	4. Ejecutar las pruebas unitarias
	5. Analizar los resultados
Descripción de los pasos:	***Paso 1. Obtener el criterio de salida*** El criterio de salida está compuesto por uno o más criterios de cobertura de código[4] y un cierto porcentaje a alcanzar. Estos deben estar previamente definidos por el Administrador del proyecto.

[4] Medida usada en las pruebas de software. Describe el grado en el que el código fuente de un programa ha sido probado.

Criterios de Cobertura

Los criterios de cobertura (criterios de adecuación para algunos autores) son un marco de trabajo para pruebas de caja blanca que sirven como regla de paro para determinar si se han creado o no suficientes casos de prueba.

Entre los criterios de cobertura comunes se encuentran; Sentencia, Decisión (o brinco) y Decisión-Condición. La cobertura de sentencia es considerada la más débil de éstos criterios, en el sentido de que es la que requiere el menor esfuerzo pero al mismo tiempo es la menos eficiente a la hora de revelar defectos.



a) Cobertura de sentencia

Al satisfacer este criterio se asegura la ejecución de cada línea de código sin importar la ruta[5]. Es el criterio de cobertura más débil porque algunos defectos se esconden en una ruta en particular, la cual no es necesariamente ejecutada al ser el único propósito el ejecutar cada línea de código.

Nota: Este es el mínimo nivel de pruebas que debe ser cubierto.

b) Cobertura de Decisión/Brinco

Este criterio requiere que cada resultado posible de cada decisión sea ejecutada al menos una vez (no para cada condición individual contenida en un predicado compuesto e.g. *if(a&b), a&b* como un todo, en vez de satisfacer las condiciones *a* y *b* de manera separada).

Nota: Este podría ser un buen objetivo para componentes de complejidad media.

c) Cobertura de Decisión-Condición

[5] Ruta o camino: Una secuencia de ejecución de sentencia que comienza con una entrada y termina con una salida (Lee Copeland, 2004).

	Al satisfacer este criterio no solo se cubre la cobertura de decisión, además asegura que cada condición dentro de una decisión tome cada resultado posible al menos una vez, y la decisión como un todo tome cada resultado posible al menos una vez (para cada condición individual contenida en un predicado compuesto e.g. *f(a&b),* se satisface las condiciones *a* y *b* de manera separada). *Note: Es recomendable para unidades complejas y críticas.* Existen criterios más fuertes como: Cobertura de Condiciones múltiples, MDC, etc. Pero se encuentran fuera del alcance de este paquete. **Porcentaje para el criterio de cobertura seleccionado** Dependiendo de la dificultad o la importancia del componente, el Administrador del proyecto debe decidir un criterio de cobertura conveniente. i.e. Un componente de complejidad media: Se requiere cobertura del criterio de Decisión de 85%. Esto significa que una vez que los casos de prueba sean diseñados y ejecutados, se necesita que al menos un 85% de los brincos (o decisiones) sean cubiertos. Cuando se cumpla este porcentaje entonces se puede decir que la pruebas están completas para dicho objetivo. ***Paso 2. Diseñar los casos de prueba*** Un caso de prueba es un conjunto específico de variables de entrada por medio del cual el programador determina si el componente trabaja adecuadamente, basándose en si las salidas obtenidas son o no las esperadas. Cada caso de prueba debe tener: ***ID del caso de prueba:*** Para tener control del número de casos de prueba creados. ***Descripción:*** Describe el propósito de la prueba. La descripción puede tener información adicional como el valor de verdad de algunas decisiones o condiciones. También puede incluir la ruta cubierta en caso de que uses un gráfico de control de flujo (GCF). Todo esto depende de la estrategia que estés usando para obtener los casos de prueba. ***Entradas:*** La configuración de los valores de las entradas que forman el caso de prueba. ***Salidas esperadas:*** Los resultados esperados para la

	configuración de las entradas. *Nota1: Es necesario un campo extra para indicar si el caso de prueba pasa o no la prueba, pero permanece vacío hasta su ejecución, donde esta información es obtenida.* ***Pasa/Falla:*** Usa P o F para indicar si el caso de prueba pasa o falla la prueba ***Paso 3. Codificar las pruebas unitarias*** Una prueba unitaria es un script (típicamente una función o método, dependiendo de la naturaleza de la implementación) que prueba uno o más casos de prueba de un componente específico Para codificar las pruebas unitarias, la mejor opción es usar un marco de trabajo (framework) de pruebas unitarias para el lenguaje que estés usando en la implementación. Si no existe un framework para dicho lenguaje, otra opción es construir tu propio marco de trabajo. ***Sin* un marco de trabajo (framework) de pruebas unitarias externo** Si no existe un framework para el lenguaje que estas usando, puedes escribir tus propias rutinas de pruebas. Para hacer esto necesitas código que verifique si una llamada al componente bajo ciertas condiciones retorna *verdadero* (si todo trabaja como lo esperado) o *falso* (si la llamada regresa una salida inesperada o algún error). Algunas rutinas de este tipo se llaman *aserciones* y usualmente toman dos argumentos: Una expresión booleana que describe si la suposición esperada es verdadera, y un mensaje que mostrar en caso contrario. La idea es que tus scripts de pruebas usen estas *aserciones* para verificar si el caso de prueba detecta un defecto. Si el lenguaje que estas usando no trae soporte para el manejo de aserciones, estas son relativamente fáciles de escribir. *Nota: Un ejemplo de una macro para aserciones es provisto en la sección de ejemplos.* ***Con* un marco de trabajo (framework) de pruebas unitarias externo** Estas herramientas son dependientes del lenguaje y te permiten escribir y correr pruebas unitarias con el objetivo de encontrar defectos en la unidad bajo prueba. Casi todas ellas cuentan con una suite de aserciones que ayudan en la realización de las pruebas.

La idea general es escribir *scripts* que contengan una o más aserciones que reciban la salida esperada y la unidad bajo prueba con una cierta configuración de valores de entrada, de tal manera que si alguna de las aserciones falla, puedas saber cual lo hizo y puedas comparar la salida esperada con la salida obtenida.

Uno de los objetivos de este tipo de herramientas es que una vez que hayas escrito un conjunto de pruebas unitarias, si existen cambios en la implementación, al volver a ejecutar las pruebas unitarias puedas verificar que no se hayan alterado funcionalidades que anteriormente estaban trabajando correctamente.

Otra ventaja importante de este tipo de herramientas es que no hay necesidad de eliminar nada, ya que desde el inicio se mantiene la implementación y las pruebas unitarias de manera separada.

Nota: En la sección de herramientas hay enlaces a los frameworks de pruebas unitarias más populares para diferentes lenguajes.

Paso 4. Ejecutar las pruebas unitarias

Nota: Aislar lo más que se pueda las unidades bajo prueba resulta importante para evitar comportamientos inadecuados debido a dependencias entre los componentes.

Sin un marco de trabajo (framework) externo

Una vez que hayas codificado las pruebas unitarias, puedes crear un mecanismo para correrlas. Puede ser un programa aparte cuyo único propósito sea el correr las pruebas unitarias, o una función o método dentro del código de la implementación con el mismo objetivo.

***Con* un marco de trabajo (framework) externo**

Los frameworks de pruebas unitarias generalmente vienen provistos con *Test runners* que básicamente son aplicaciones gráficas o de línea de comandos que son usadas para correr las pruebas y reportar los resultados.

a) Test Runner de línea de comandos

Estas aplicaciones son diseñadas para ser usadas desde la línea de comandos de un SO o desde algún archivo batch o shell scripts. Resultan de gran utilidad cuando las pruebas son ejecutadas remotamente o cuando se usa algun build script como “make” o “ant”. Si el conjunto de pruebas es muy grande puedes dejarlas correr una noche y ver los resultados al siguiente día.

	Nota: Un ejemplo es provisto en la sección de ejemplos **b) Test Runner gráfico** Un Test Runner gráfico es típicamente una aplicación de escritorio o una parte de un IDE[6] (un plug-in) cuya función es la ejecución de las pruebas de una manera más sencilla y cómoda. La característica más común de este tipo de herramientas es un indicador de progreso en tiempo real. Algunas de ellas también incluyen un contador del número de defectos encontrados y una barra de colores para indicar el progreso, la cual generalmente comienza en verde y se torna roja conforme detecta algún error en las pruebas. *Nota: Un ejemplo es provisto en la sección de ejemplos.* ***Paso 5. Analizar los resultados*** Dependiendo de cómo se ejecuten las pruebas unitarias, puedes ya sea analizar los resultados almacenados en un archivo de registro o directamente en la consola en el caso de que uses un Test runner de línea de comandos, o visualizarlos gráficamente en el caso de que uses una aplicación gráfica. En cualquier caso el siguiente paso es determinar si alguna de las pruebas falla. Si esto ocurre, debes realizar las correcciones pertinentes ya sea en el código de la implementación o en el código de las pruebas[7] unitarias y volver a ejecutarlas nuevamente. Una vez que todas las pruebas unitarias han pasado, debes revisar el criterio de salida definido por el Administrador del proyecto y ver si cuentas con suficientes casos de prueba o si es necesaria la creación de nuevos casos. Cabe mencionar que es altamente recomendable usar herramientas de Cobertura de Código para ver que tanto código ha sido cubierto por tus pruebas y que porcentaje has alcanzado del criterio seleccionado. *Nota: Algunas herramientas populares son listadas en la sección de herramientas, la mayoría de ellas consideran al menos los criterios de cobertura de sentencia y decisión.*

[6] IDE (Integrated Development Environment). Ambiente integrado de desarrollo de software que provee ciertas facilidades a los programadores, tales como: editores de código, compiladores o intérpretes, depuradores etc.

[7] En algunas ocasiones los errores no se encuentran en la implementación sino en el código de las pruebas

Corregir los defectos

Note: Los defectos son corregidos constantemente mientras no se alcance el criterio de salida seleccionado.

Objetivo:	Corregir los defectos encontrados por las pruebas unitarias.
Justificación:	Corregir los defectos encontrados por las pruebas unitarias por la persona que se encuentra a cargo de la construcción del componente, es el método más rápido y menos costoso para corregir los defectos.
Roles:	Programador
Productos:	Componentes de Software [corregidos]
Pasos:	1. Confirmar el defecto
	2. Determinar la naturaleza y ubicación del defecto
	3. Corregir el defecto
	4. Verificar las correcciones

Descripción de los pasos:	***Paso 1. Confirmar el defecto***
	
	Verifica que lo que encontraste sea un defecto en el código de la implementación (el propósito de los casos de prueba) y no un defecto en el código de la prueba.
	
	Paso 2. Determinar la ubicación del defecto
	
	Prácticas para localizar defectos:
	• ***Usar listas de revisión de código:*** Si no encuentras el error, puedes apoyarte en las listas de revisión de código para darte una idea de dónde buscar.
	Nota: Un ejemplo de una lista de revisión de código provisto en la sección de ejemplos
	• ***Pasar el código a través del depurador***
	Una vez que la rutina compila, revisa línea por línea a través del depurador. Asegúrate que cada línea se ejecute de acuerdo a lo esperado.
	• ***Encuentra la fuente del error.*** Trata de reproducirlo por diferentes maneras para determinar su causa exacta. Algunos consejos pueden ser:
	    • Reduce la región del código sospechoso
	    • Usa diferentes datos
	    • Refina los casos de prueba

Figure 4. Reproduce un error por diferentes medios para determinar su causa exacta. [Code complete]

- ***Usa la experimentación como último recurso:*** El error más común para algunos programadores es tratar de resolver un problema haciendo cambios experimentales al programa, sin embargo esto generalmente inyecta más defectos.

Paso 3. Corregir el defecto

Una vez que has encontrado el origen del defecto, guarda el código original y después realiza los cambios correspondientes.

Nota: Otro error común es reparar los síntomas del error o solo una instancia de éste, en vez de corregir el error en sí.

Paso 4. Verificar las correcciones

Ejecuta nuevamente las pruebas unitarias para verificar que las correcciones hechas trabajen apropiadamente.

Repite desde el paso 1 cada vez que una prueba unitaria encuentre un defecto. Si las pruebas unitarias no encuentran más defectos lo siguiente alcanzar el criterio de salida seleccionado para tu componente.

Actualizar el registro de trazabilidad

Nota: Se Incorporan los componentes de software construidos o modificados

Objetivo:	Asegurarse que todos los componentes de software puedan ser trazados a un elemento de diseño una vez que sean construidos.
Justificación:	El registro de trazabilidad debe ser desarrollado en las fases previas del proyecto para asegurar la trazabilidad de los requerimientos al diseño de los componentes. Esta tarea es realizada para asegurar la trazabilidad entre el diseño y la construcción de los componentes.
Roles:	Programador
Productos:	Registro de trazabilidad [Actualizado]
Pasos:	1. Actualizar el registro de trazabilidad
Descripción de los pasos:	***Paso 1. Actualizar el registro de trazabilidad*** Este registro debe ser actualizado con la siguiente información: - Identificador único de componentes de software - Identificación de casos de prueba (opcional) - Fecha de verificación (i.e Fecha en que el componente de software ha sido probado y no se encontraron nuevos defectos)

Descripción de los roles

A continuación se presenta una lista alfabética de roles, abreviaciones y listas de competencias como se encuentran definidas en el ISO 29110 Parte 5-1-2.

	Roles	***Abreviaciones***	***Competencias***
1.	Analista	AN	Conocimiento y experiencia que permitan deducir, especificar y analizar los requerimientos Conocimiento en el diseño de interfaces de usuario y criterios ergonómicos. Conocimiento en técnicas de revisión. Conocimiento en técnicas de edición. Experiencia en el desarrollo y mantenimiento de software.
2.	Cliente	CLI	Conocimiento de los procesos del cliente y habilidad para explicar los requerimientos del cliente El cliente (representante) debe tener la autoridad para aprobar los requerimientos y sus cambios.

			El cliente incluye representantes de usuarios con el objetivo de asegurar que el ambiente operacional sea el indicado. Conocimiento y experiencia en el dominio de la aplicación.
3.	Programador	PR	Conocimiento y/o experiencia en programación, integración y pruebas unitarias. Conocimiento de técnicas de revisión. Conocimiento de técnicas de edición. Experiencia en el desarrollo y mantenimiento de software.
4.	Líder Técnico	LT	Conocimiento y experiencia en el dominio de procesos de software.
5.	Administrador del proyecto	AP	Capacidad de liderazgo con experiencia en la toma de decisiones, planeación, administración de personal, delegación y supervisión, finanzas y desarrollo de software.

Descripción de productos

A continuación se presenta una lista alfabética de las entradas, salidas y productos de procesos internos, sus descripciones, posibles estados y la fuente del producto.

	Nombre	**Descripción**	**Fuente**

1.	*Plan del proyecto*	Presenta como los procesos del proyecto y sus actividades serán ejecutadas para asegurar el éxito de la completitud proyecto, y la calidad de los productos entregables. Incluye los siguientes elementos que pueden tener las siguientes características:
		- *Descripción del Producto*
		    ○ Propósito
		    ○ Requerimientos generales del cliente
		- Descripción del alcance sobre lo que se incluye y lo que no
		- *Objetivos* del proyecto
		- *Entregables* – lista de productos para ser entregados al cliente.
		- *Tareas, incluyendo* verificación, validación y revisiones con el cliente y el equipo de trabajo, para asegurar la calidad de los productos. Las tareas pueden ser representadas como una Estructura de Descomposición del Trabajo (EDT).
		- *Relaciones y Dependencias de las Tareas*
		- *Duración Estimada* de las tareas
		- *Recursos* (humanos, materiales, equipo y herramientas) incluyen el entrenamiento
			Administración del proyecto

		requerido, y el calendario cuando los recursos son necesarios.
		- *Composición* del equipo de trabajo
		- *Calendario* de las áreas del proyecto, la fecha de inicio esperado y la fecha de término, para cada tarea.
		- *Esfuerzo estimado* y costo
		- *Identificación de los riesgos del proyecto*
		- Estrategia para el *control de versiones*
		  - Herramientas para el repositorio de productos o mecanismos identificados
		  - Mecanismos de acceso y localización para el repositorio especificado.
		  - Identificación de versiones y definición de controles
		  - Mecanismos de respaldo y recuperación definidos
		  - Especificación de mecanismos de almacenamiento, manejo y entrega (incluyendo archivo y recuperación)
		- *Instrucciones de entrega*
		  - Elementos requeridos para la liberación del producto identificado (i.e, hardware, software, documentación, etc.)
		  - Requerimientos de entrega
		  - Orden secuencial de tareas a ser realizadas
		  - Liberaciones aplicables identificadas
		  - Identifica todos los componentes de software entregados con la información de versiones
		  - Identifica cualquier procedimiento de respaldo o recuperación necesario
		
		Los estados aplicables son: Verificado, Validado, Modificado y Revisado.	

2.	*Componente de Software*	Un conjunto de unidades de código relacionadas. Los estados aplicables son: Probado, Corregido y Registrado en línea base	Implementación de Software
3.	*Diseño de Software*	Este documento incluye información textual y gráfica de la estructura del software. La estructura puede incluir las siguientes partes: Diseño de software arquitectónico de alto nivel – Describe la estructura total del software: - Identifica los componentes requeridos de software - Identifica las relaciones entre los componentes de software	Implementación de Software

		- Las consideraciones son dadas a cualquier requisito:
		  - Características de desempeño de software
		  - Hardware, software e interfaces humanas
		  - Características de seguridad
		  - Requerimientos de diseño de base de datos
		  - Manejo de errores y atributos de recuperación
		
		Diseño de software detallado a bajo nivel – Incluye detalles de los componentes de software para facilitar su construcción y prueba dentro del ambiente de desarrollo.
		- Provee el diseño detallado (Puede ser representado por un prototipo, diagrama de flujo, diagrama entidad-relación, pseudocódigo, etc.)
		- Provee un formato de entrada/salida para datos
		- Provee especificaciones para necesidades de almacenamiento de datos
		- Establece convenciones de nombrado de datos
		- Define el formato de las estructuras de datos requeridas
		- Define los campos de datos y el propósito de cada elemento de los datos requeridos
		- Provee las especificaciones de la estructura del programa
		
		Los estados aplicables son: verificado y registrado en línea base	

4.	*Registro de trazabilidad*	Documenta las relaciones entre los requerimientos incluidos en la *Especificación de Requerimientos*, elementos del *Diseño de Software,* los *Componentes de Software,* y *Casos de prueba.* Puede incluir: - Identifica los requerimientos de la *Especificación de Requerimientos* a ser trazados - Provee mapeo hacia adelante y hacia atrás de los requerimientos a los elementos del *Diseño de Software*, *Componentes de Software y Casos de prueba.* Los estados aplicables son: Verificado, Registrado en línea base y Actualizado.	Implementación de Software

Descripción de artefactos

A continuación se muestran una lista alfabética de los artefactos que pueden ser producidos para facilitar la documentación del proyecto. Los artefactos no son requeridos por la parte 5, son opcionales.

	Nombre	Descripción
1.	Estándares de Construcción	Provee convenciones, reglas y estilos para: - Organización de código fuente (sentencias, rutinas, clases y otras estructuras) - Documentación de código - Módulos y archivos - Variables y constantes - Expresiones - Estructuras de control - Funciones - Manejo de condiciones de error Entre otras.
2.	Casos de prueba	Un conjunto de entradas para realizar pruebas, condiciones de ejecución y resultados esperados desarrollados con un objetivo en particular, tales como entrenar una ruta específica de un programa o verificar el cumplimiento de un requerimiento específico [IEEE 1012-2004].
3.	Conjunto de pruebas unitarias	Conjunto de rutinas escritas en un lenguaje específico diseñadas para probar los casos de prueba
4.	Taxonomía de defectos	Es un método que reduce el número de defectos en el producto aprendiendo de los tipos de errores que se comenten en el proceso de desarrollo, de tal manera que puedan ser analizados para mejorar el proceso de reducir o eliminar la probabilidad de que se cometa el mismo tipo de error en el futuro.

5. Plantillas

5.1 Plantilla de construcción para Java

```
/******************************************************************************
* Copyright (c) 20XX [Nombre de la Organización], Inc. Todos los derechos reservados.
******************************************************************************
    Número de Versión - $Revisión$
    Última actualización   - $Fecha$
    Actualizado por     - $Autor$

    Resumen del propósito del componente

    Diseño de bajo nivel, discusiones de diseño físico, dependencias,
    suposiciones, detalles de implementación, notas, etc.

/******************************************************************************
    Paquetes y sentencias de importación
*/

package com.miEmpresa.miPaquete;
import com.algunPaquete;

/******************************************************************************
    Usa estas divisiones para dividir el módulo en agrupaciones lógicas.
*/
/******************************************************************************
    Definiciones de Clase
*/
/*=============================================================================
    Resumen del objetivo de la clase
 ............................................................................
    Descripción de la clase
=============================================================================*/
public class JavaClass extends SuperClass
    implements AlgunaInterface
    {
    /*---------------------------------------------------------------------
        Constructores
     ..................................................................
       Descripción de los constructores
    ---------------------------------------------------------------------*/
    // Resumen del Constructor 1
    public JavaClass( )
```

```
        {

        }
    /*====================================================================
        Funciones miembro Públicas
    */
    /*--------------------------------------------------------------------
        Resumen de funciones miembro
     ....................................................................
        Descripción de funciones
        Descripción de excepciones y valores de retorno
    --------------------------------------------------------------------*/
    public TIPO_DATO_RETORNO
    algunMetodo(
        TIPO_PARAM1 param1,     // Descripción del parámetro 1
        TIPO_PARAM2 param2      // Descripción del parámetro 2
        )
        throws algunaExcepcion
        {

        }
    /*====================================================================
        Funciones miembro Privadas
    */
    /*====================================================================
        Datos miembro Protegidos
    */
    protected boolean unValorBooleano;

    /*====================================================================
        Datos miembro Privados
    */
    private int unValorEntero;
    }
/******************************************************************************/
```

6. Ejemplos

6.1 Ejemplo de un estándar genérico de construcción

Formato

1	Todos los archivos fuente deben estar basados en una plantilla de construcción apropiada.
2	Si no existe un estilo definido en el código o el estilo difiere del estándar de construcción, dale formato al archivo para que esté acorde al estándar.

3	Evita crear líneas de más de 79 caracteres de longitud. Pero si son necesarias, identa la siguiente línea un nivel. Si la nueva línea también excede los 79 caracteres inserta un nuevo salto de línea después del último operador dentro del límite de los 79 caracteres, pero no identes la siguiente línea. Continúa de esta forma hasta que todas las líneas de la sentencia sean de menos de 79 caracteres en longitud.
	`If (PreliminarySize = 0 And CalculatedMeanSizeInLoc(Module) <> 0) Or`
	`    (PreliminarySize = CalculatedMeanSizeInLoc(Module)) Then`
	`   Call CalibrationData.GetProjectPhaseFromId(`
	`       ProjectCharacteristics.CurrentProjectPhase, ProjectPhase)`
	`End If`

4	Usa **líneas en blanco** deliberadamente para separar y organizar el código, 1 dentro de funciones o comentarios, 2 entre funciones, 3 o 4 para dividir secciones.

5	**Usa espacios** para separar de manera visible las variables o funciones.
	`If ( 3 == x)`
	`func1 ( param1, param2 );`
	`int array [ MAX_NUM_COSAS ];`

Comentarios

1	Usa comentarios para describir las intenciones del programador, describiendo el **"porqué"** en vez del **"como" o "qué"**.
2	Solo para **piezas críticas de código** debes explicar en detalle cómo funciona. Pero en ese caso añade una **marca especial**.
3	La mejor forma de comentar código es hacerlo en **lenguaje natural** (Español, Inglés), no más código o pseudocódigo.
4	Otros programadores deben entender los comentarios (información **útil**).
5	Trata de uno usar comentarios redundantes o innecesarios.

6	Comenta cada **bloque de código** con una descripción breve de su intención.
7	Identa los **comentarios al mismo nivel de identación** que el código que comentan (porque no son ni más ni menos importantes que el código de la implementación).
8	Si es posible, usa una **herramienta de comentarios automáticos** como **JavaDoc** que permita examinar la documentación a nivel de código de manera separada del código de la implementación.
9	El **Rango numérico** de los datos debe ser comentado.
10	Las limitaciones de entrada o salida de datos deben ser comentadas.

Módulos y archivos

1	Cuando se nombre un archivo la **primera letra** debe ser **mayúscula**, así como también la **primera letra** de cualquier palabra o **acrónimo** que se incluya en el nombre del archivo. Usa letras **minúsculas** para cualquier otro carácter alfabético (**usa** sólo caracteres **alfanuméricos**).
	`NombreArchivo.java`
	`NombreArchivo2.cpp`

2	Archivos con **propósitos** similares deben ser **nombrados** similarmente.
3	**Comienza** todos los archivos fuente con el comentario estándar para la **cabecera**, el cual describe el **propósito** del archivo y una **breve sinopsis** de su contenido.
4	Colocar más de **un componente** o módulo dentro de un solo archivo fuente **no es recomendable.**
5	Trata de **usar rutas relativas** para prevenir incluir archivos en ambientes no deseados.

Variables

1	Usa cada variable para exactamente un solo propósito.

2	Usa **mayúscula** para **el primer carácter** de una palabra o **acrónimo** dentro del nombre de una variable; usa minúscula para cualquier otro carácter alfabético. **El primer carácter** del nombre de una variable debe ser una letra en **minúscula.**
	`balance // Nombre de variable con una palabra`
	`mesTotal // Nombre de variable con dos palabras`
	`exportarDoc // Nombre de variable con acrónimo`
	Cada variable debe ser precedida con un **comentario breve** que describa el **propósito** de la variable. Porque incluso los nombres más descriptivos pueden ser confusos en algunas ocasiones.

3	**Trata de NO usar la notación del estilo Húngaro** para ligar el tipo de variable al nombre, ya que puede ser una distracción y el propósito original del nombre de

	la variable puede perderse. Notación Húngara: `float   fbalance`
4	**Inicializa** una variable cuando sea declarada, porque asegura un valor conocido para la variable.

Constantes

1	Usa caracteres **alfanuméricos en mayúsculas** para los **nombres de las constantes.** Separa las palabras del nombre de una constante por **guiones bajos.** `ESTO_ES_UNA_CONSTANTE`
2	Trata de no usar constantes numéricas sin nombre (**números mágicos**)**,** incluso si sólo es usada una sola vez.

3	Las cadenas de texto que son mostradas al usuario siempre deben ser almacenadas en **cadenas constantes** o tablas de cadenas. Esto les permite ser traducidas a otro idioma o actualizadas cuando se necesite.
	`OPTION1 = “Selecciona una herramienta”`
	`OPTION1 = “Select a tool”`

Expresiones y sentencias

1	En lenguajes donde los operadores de asignación y comparación difieren, hay que tener cuidado con la ambigüedad que se puede generar cuando se realizan asignaciones dentro de expresiones condicionales. Caso de ambigüedad: `If ( valor = 3 + i )`
2	Trata de tener un **orden** en las comparaciones ya que resulta más fácil de entender. Por ejemplo usa **<** y **<=**, en vez de > o >=.

Estructuras de Control

1	**Identa** bloques de código usando “llaves o delimitadores afines”. Idéntalos al **mismo nivel** que el código contenido dentro el bloque.
	`if ( true == exito )`
	`{`
	`// Haz algo.`
	`}`

2	Usa **nombres descriptivos** para los índices de los ciclos, tales como “fil” y “col”. Si el ciclo es simple y pequeño, los índices tradicionales tales como ‘i’, ‘j’ y ‘k’ son aceptables.

3	Usar sentencias **break o return** para romper ciclos **no es recomendable.**
4	**Usa de preferencia ciclos** `for` cuando sea apropiado, ya que hace a los ciclos más fáciles de entender y mantener, además de sirven como método de prevención para no caer en ciclos infinitos.
5	Si estas incluyendo otras **sentencias** en las **cabeceras** de los ciclos `for`, posiblemente necesites usar un ciclo `while`.

Funciones

1	Sigue el **mismo formato** que en el **nombrado de variables** cuando nombres a las funciones, con la única diferencia de que el **primer carácter** del nombre de la función debe estar en **mayúscula.**

2	Para una function cuyo **propósito** principal sea **realizar una acción,** usa un nombre con la forma **verbo-objeto**. Si el método pertenece a un objeto entonces solo usa un **verbo**.
	`ImprimirDocumento( documento );     //Nombre de función del tipo : verbo-objeto`
	`documento.Imprimir();             //Nombre de función del tipo: verbo`

3	Para una función cuyo propósito principal es retornar un valor, usa un nombre que describa el valor retornado. `if ( ARCHIVO_ABIERTO == file.Estado() )`
4	Cada declaración de función debe ser precedida con un comentario que describa la **intención de la función**, el **propósito de cada argumento**, y el **valor de retorno** de la función. *Nota: Javadoc tiene sus propias etiquetas para realizar esta acción.*
5	Usar **múltiples sentencias de retorno** como medio de salida para una función **no es recomendable**.

6.2 Ejemplo de Pseudocódigo

Especificación

Un procedimiento debe computar la prima de seguro para un conductor de automóvil, basándose en la siguiente información:

- La edad de la persona (entero).
- El sexo de la persona ("M" para masculino, "F" para femenino).
- El estado civil de la persona (casado o no casado).

Las reglas del negocio para computar la prima son las siguientes:

- Una prima básica de $500 aplica a cualquiera.
- Si la persona es un hombre soltero de menos de 25 años de edad, un costo adicional de $1500 es añadido a la prima básica.
- Si la persona es casada o es mujer, un descuento de $200 es aplicado a la prima básica.

  Si la persona tiene entre 45 y 64 años, un descuento de $100 es aplicado a la prima básica

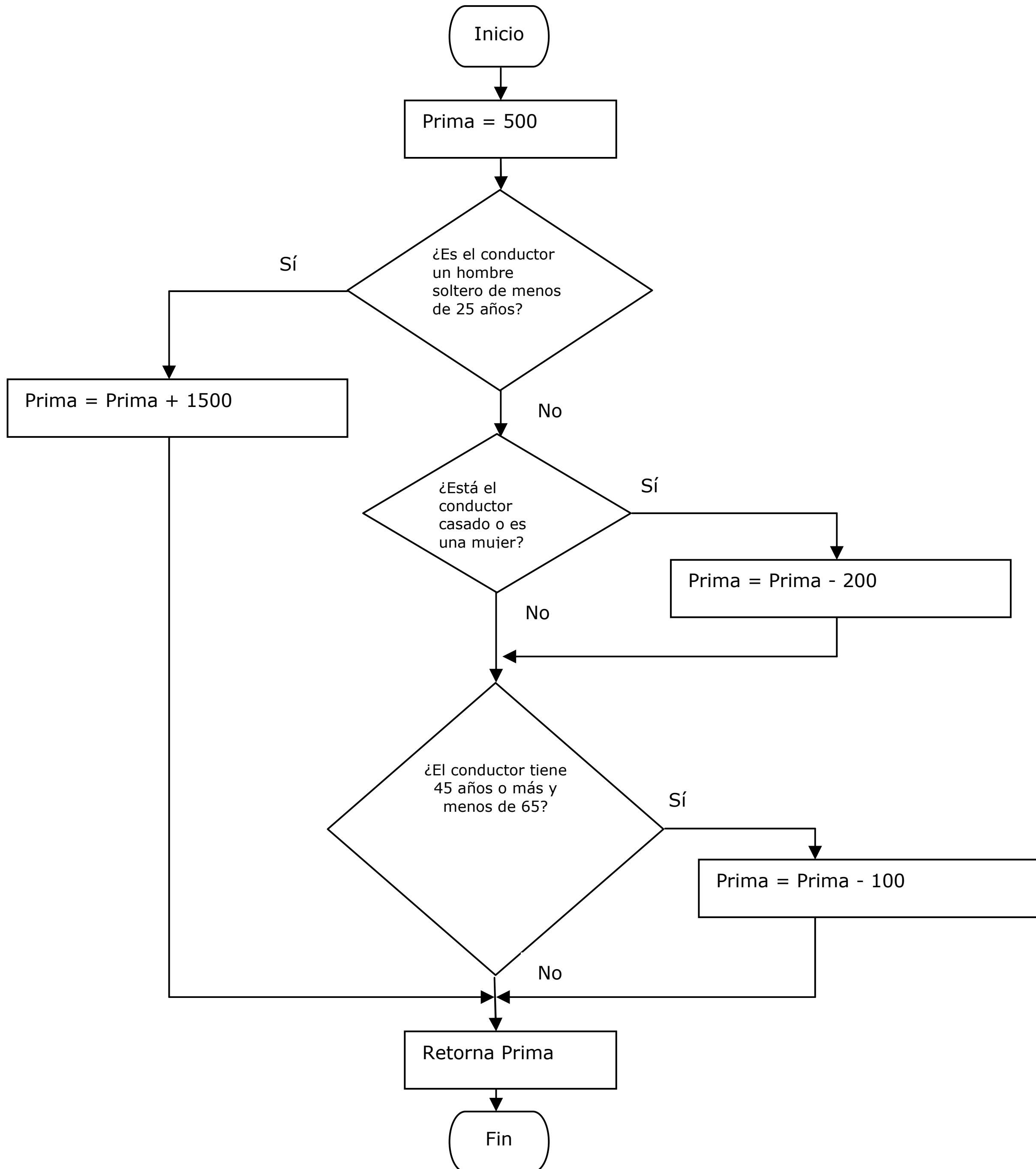
El procedimiento debe regresar la prima correspondiente.

Pseudocódigo basado en la especificación anterior

```
Inicializa el costo de la prima básica en $500
Si el conductor es un hombre soltero menor de 25 años de edad
        Un costo de $1500 es añadido a la prima básica
En otro caso
        Si el conductor es casado o es una mujer
                Un descuento de $200 es aplicado a la prima básica
        Si el conductor tiene 45 años o más y menos de 65 años
                Un descuento de $100 es aplicado a la prima básica
Retorna la prima
```

6.3 Ejemplo de Diagrama de Flujo

Este diagrama de flujo corresponde a la especificación provista en el ejemplo 6.2



6.4 Jerarquía de criterios de cobertura, del más débil al más fuerte

Rutina

```
x = 0
if(edad < 65 && casado == true)
{
       x = 2
       y = 10 + edad
}
return (edad/x)
```

Mientras más fuerte es el criterio, mayor es el número defectos que serán revelados por las pruebas.

Casos de prueba para la cobertura de sentencia: Todas las sentencias tienen que ser ejecutadas al menos una vez; para este caso sólo es necesario un caso de prueba para alcanzar 100% de cobertura de sentencia.

ID del caso de prueba	Descripción	Entrada		Salida esperada	Pasa/Falla
		Edad	Casado		
CP1	Un conductor casado de 30 años	30	Verdadero	15	P

Nota: Con este caso de prueba ejecutas todas las líneas de código, pero el caso en el que la decisión es falsa nunca es probado, por lo tanto no detecta el error aritmético edad/x cuando x=0.

Casos de prueba para la cobertura de decisión/brinco: Cada decisión debe tomar cada posible resultado al menos una vez.

Decisión: `edad < 65 && casado == true`

ID del caso de prueba	Descripción	Entrada		Resultado de la Decisión *(Predicado compuesto como un todo)*	Salida esperada	Pasa/ Falla
		Edad	Casado			
CP1	Un conductor casado de 30 años	30	Verdadero	Verdadero	15	P
CP2	Un conductor casado de 75 años	75	Verdadero	Falso	0	F

Notas:

- *El caso de prueba CP2 detectó un defecto que la cobertura de sentencia no.*
- *El valor para casado = Falso nunca es considerado y aún así 100% de la cobertura de decisión es alcanzada. Esto pasa por que no fueron analizados todos los valores posibles para cada condición. De tal forma que si un defecto se esconde en la condición 2 (casado == true), este puede no ser detectado.*

Casos de prueba para la cobertura de Decisión-Condición

Cada decisión debe tomar cada resultado posible al menos una sola vez, y cada condición dentro de las decisiones debe tomar cada resultado posible al menos una sola vez.

Decisión: `edad < 65 && casado == true`

Condición 1: `edad < 65`
Condición 2: `casado == true`

ID del caso de prueba	Descripción	Entrada		Resultado de condición 1	Resultado de condición 2	Resultado de la decisión	Salida esperada	Pasa/ Falla
		Edad	Casado					
CP1	Un conductor casado de 30 años de edad.	30	Verdadero	Verdadero	Verdadero	Verdadero	15	P
CP2	Un conductor casado de 75 años de edad	75	Verdadero	Falso	Verdadero	Falso	0	F
CP3	Un conductor soltero de 30 años de edad	30	Falso	Verdadero	Falso	Falso	0	F

Nota: Si la condición 2 tuviera un defecto lógico en su resultado falso, entonces el caso de prueba CP3 lo hubiera detectado. De cualquier forma este no es el caso

El criterio descrito anteriormente NO requiere de la cobertura de todas las posibles combinaciones entre las condiciones. Aún así, considerando que la mayoría de los compiladores dejan de evaluar una expresión *"and"* tan rápido como determinan que un operador es *falso*, o una expresión *"or"* tan rápido como determinan que un operador es *verdadero.* Algunas condiciones no serán evaluadas sin importar el valor que tengan.

Para resolver este problema es recomendable probar los operadores *"and" y "or"* en el siguiente orden:

```
if( a && b && c && d )
```

Entradas	Valores (V=verdadero, F=falso)
a,b,c,d	V V V V
a,b,c,d	F V V V
a,b,c,d	V F V V
a,b,c,d	V V F V
a,b,c,d	V V V F

```
if( a || b || c || d )
```

Entradas	Valores (V=verdadero, F=falso)
a,b,c,d	F F F F
a,b,c,d	V F F F
a,b,c,d	F V F F
a,b,c,d	F F V F
a,b,c,d	F F F V

De esta forma aseguras que cada condición tome un valor verdadero y uno falso dentro de la ejecución en el programa.

En el caso de los operadores "and", el primer conjunto (V V V V) servirá para evaluar el resultado verdadero de la decisión completa y cualquiera de los otros casos funcionará para evaluar el resultado falso.

En el caso de los operadores "or", el conjunto (F F F F) servirá para evaluar el resultado falso de la decisión completa y cualquiera de los otros casos funcionará para evaluar el resultado verdadero.

Nota: Si la decisión está compuesta de una mezcla entre operadores "and" y operadores "or", puedes usar paréntesis "(,)" para determinar la jerarquía de operadores que mejor te convenga.

6.5 Diseño de casos de prueba, escritura de pruebas unitarias y uso de una herramienta de cobertura de código

Rutina

```
public int getPrima() {

        int prima = 500;

        if ((edad < 25) && !casado && sexo.equals("M"))
        {
                prima += 1500;
        } else
        {
                if (casado || sexo.equals("F"))
                {
                        prima -= 200;
                }
                if ((edad >= 45) && (edad < 65))
                {
                        prima -= 100;
                }
        }
        return prima;
}
```

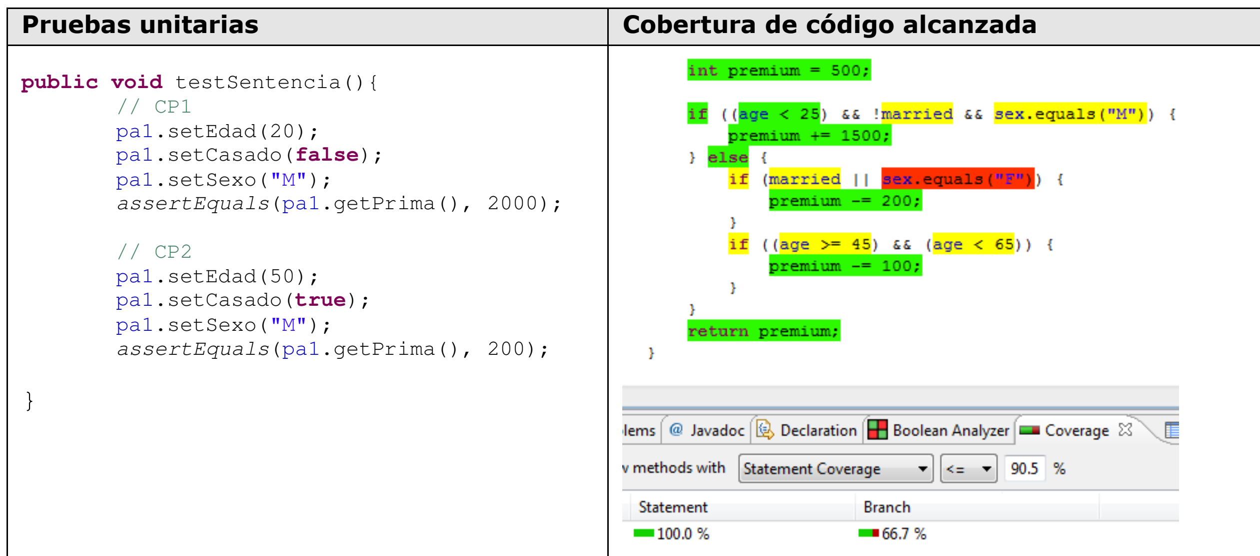
Nota: Las pruebas unitarias fueron escritas usando aserciones con los valores de entrada de cada caso de prueba y las salidas esperadas. Por conveniencia el lenguaje elegido es Java, el framework de pruebas unitarias es Junit y la herramienta de cobertura de código es Codecover.

Cobertura de Sentencia

Para probar todas las sentencias en este caso es necesario probar los valores verdadero y falso de la primera decisión (primer if).

ID del caso de prueba	Descripción	Entradas	Salida esperada
CP1	Un hombre soltero de 20 años	Edad = 20, Soltero, Masculino	2000
CP2	Un hombre casado de 50 años	Edad= 50, Casado, Masculino	200

Tabla 4 Casos de prueba requeridos para satisfacer 100% de cobertura de sentencia.



Pruebas unitarias	Cobertura de código alcanzada

```
public void testSentencia(){
        // CP1
        pa1.setEdad(20);
        pa1.setCasado(false);
        pa1.setSexo("M");
        assertEquals(pa1.getPrima(), 2000);

        // CP2
        pa1.setEdad(50);
        pa1.setCasado(true);
        pa1.setSexo("M");
        assertEquals(pa1.getPrima(), 200);

}
```

```
int premium = 500;

if ((age < 25) && !married && sex.equals("M")) {
    premium += 1500;
} else {
    if (married || sex.equals("F")) {
        premium -= 200;
    }
    if ((age >= 45) && (age < 65)) {
        premium -= 100;
    }
}
return premium;
}
```


Tabla 5 Las pruebas unitarias satisfacen 100% de la cobertura de sentencia (*statement coverage - en inglés*). El código en colores tiene el siguiente significado: Verde – completamente cubierto, Amarillo – parcialmente cubierto, Rojo – nunca ejecutado (Codecover, plugin para Eclipse Java IDE)

Como se puede ver en la figura anterior, todas las sentencias fueron ejecutadas, pero no todos los posibles resultados de cada decisión o condición.

Cobertura de Decisión/Brinco

Las decisiones de la rutina son:

D1	`if ((edad < 25) && !casado && sexo.equals("M"))`
D2	`if (casado || sexo.equals("F"))`
D3	`if ((edad >= 45) && (edad < 65))`

Decisión	Resultado verdadero	Resultado falso
D1	Un conductor masculino soltero con menos de 25 años de edad.	Un conductor de 25 años de edad o más, ó un conductor casado, ó un conductor femenino.
D2	Un conductor casado o un conductor femenino.	Un conductor masculino soltero.
D3	Un conductor entre los 45 y 64 años de edad	Un conductor con menos de 45 años o más de 64

Tabla 6 Situaciones de entrada para ejercitar todos los resultados posibles de las decisiones

ID del caso de prueba	Descripción *(Prueba de los resultados)*	Entradas	Salida esperada
CP1	D1V	Edad = 20, Soltero, masculino	2000
CP2	D1F,D2V,D3V	Edad = 45, Casado, masculino (irrelevante)	200
CP3	D1F,D2F,D3F	Edad = 35, Soltero, masculino	500

Tabla 7 Casos de prueba obtenidos. Nota: D#V significa probar el resultado verdadero de la decisión correspondiente. D#F significa probar el resultado falso de la decisión correspondiente.

<table>
<tr><th>Pruebas unitarias</th><th>Cobertura de código alcanzada</th></tr>
<tr><td>

```
public void testDecision(){
        // CP1
        pa1.setEdad(20);
        pa1.setCasado(false);
        pa1.setSexo("M");
        assertEquals(pa1.getPrima(), 2000);

        // CP2
        pa1.setEdad(45);
        pa1.setCasado(true);
        pa1.setSexo("M");
        assertEquals(pa1.getPrima(), 200);

        // CP3
        pa1.setEdad(35);
        pa1.setCasado(false);
        pa1.setSexo("M");
        assertEquals(pa1.getPrima(), 500);
}
```

</td><td>

```
    int premium = 500;

    if ((age < 25) && !married && sex.equals("M")) {
        premium += 1500;
    } else {
        if (married || sex.equals("F")) {
            premium -= 200;
        }
        if ((age >= 45) && (age < 65)) {
            premium -= 100;
        }
    }
    return premium;
}
```

ms | Javadoc | Declaration | Boolean Analyzer | Coverage
methods with Statement Coverage <= 90.5 %
Statement 100.0 % Branch 100.0 %

</td></tr>
</table>

Tabla 8 Las pruebas unitarias satisfacen 100% de la cobertura de decisión/brinco (branch coverage – en inglés). El código en colores tiene el siguiente significado: Verde – completamente cubierto y Amarillo – parcialmente cubierto (Codecover, plugin para Eclipse Java IDE).

Como se puede observar en la figura anterior, las condiciones de las decisiones están parcialmente cubiertas (código coloreado de amarillo), esto debido a que aunque todos los posibles resultados de las decisiones fueron probados, no así todas los posibles resultados de las condiciones.

Cobertura de Decisión-Condición

D1	`if ((edad < 25) && !casado && sexo.equals("M"))`
D2	`if (casado || sexo.equals("F"))`
D3	`if ((edad >= 45) && (edad < 65))`

Las condiciones de las decisiones son las siguientes:

D1	C1	`edad < 25`
D1	C2	`!casado`
D1	C3	`sexo.equals("M")`
D2	C4	`casado`
D2	C5	`sexo.equals("F")`
D3	C6	`edad >= 45`
D3	C7	`edad < 65`

Condición	Resultado verdadero	Resultado falso
C1	Un conductor con menos de 25 años	Un conductor de 25 años o más
C2	Un conductor soltero	Un conductor casado
C3	Un conductor masculino	Un conductor femenino
C4	Un conductor casado	Un conductor soltero

C5	Un conductor femenino	Un conductor masculino
C6	Un conductor de 45 años de edad o más	Un conductor con menos de 45 años de edad
C7	Un conductor con menos de 65 años de edad	Un conductor con 65 años de edad o más

Tabla 9 Situaciones de entrada para ejecutar todos los posibles resultados de las condiciones

ID del caso de prueba	Descripción (*Probar los resultados*)		Entrada	Salida esperada
	Decisiones	Condiciones		
CP1	D1V	C1V,C2V,C3V	Edad= 20, Soltero, *masculino*	2000
CP2	D1F	C1F,C2V,C3V	Edad= 50, Soltero, *masculino*	400
CP3	D1F	C1V,C2F,C3V	Edad= 20, Casado, *masculino*	300
CP4	D1F	C1V,C2V,C3F	Edad= 20, Soltero, Femenino	300
CP2	D2F	C4F,C5F	Edad= 50, Soltero, masculino	400
CP3	D2V	C4V,C5F	Edad= 20, Casado, masculino	200
CP4	D2V	C4F,C5V	Edad= 20, Soltero, Femenino	200
CP2	D3V	C6V,C7V	Edad= 50, Soltero, masculino	400
CP5	D3F	C6F,C7V	Edad= 30, Soltero, *masculino*	500
CP6	D3F	C6V,C7F	Edad= 70, Soltero, *masculino*	500

Tabla 10 Casos de prueba obtenidos. Nota: C#T significa probar el resultado verdadero de la condición relacionada. C#F significa probar el resultado falso de la decisión relacionada.

Nota 1: Para obtener los casos de prueba se siguió el consejo dado en el ejemplo 6.4 para este tipo de cobertura.

Nota 2: Recuerda que algunas veces la misma configuración de entradas puede probar más de un caso de prueba. Por lo tanto siempre trata de optimizar el número de casos de prueba.

Pruebas unitarias

```
public void testDecisionCondicion(){
        // CP1
        pa1.setEdad(20);
        pa1.setCasado(false);
        pa1.setSexo("M");
        assertEquals(pa1.getPrima(), 2000);

        // CP2
        pa1.setEdad(50);
        pa1.setCasado(false);
        pa1.setSexo("M");
        assertEquals(pa1.getPrima(), 400);

        // CP3
        pa1.setEdad(20);
        pa1.setCasado(true);
        pa1.setSexo("M");
        assertEquals(pa1.getPrima(), 300);

        // CP4
        pa1.setEdad(20);
        pa1.setCasado(false);
```

```
        pa1.setSexo("F");
        assertEquals(pa1.getPrima(), 300);

        // CP5
        pa1.setEdad(30);
        pa1.setCasado(false);
        pa1.setSexo("M");
        assertEquals(pa1.getPrima(), 500);

        // CP6
        pa1.setEdad(70);
        pa1.setCasado(false);
        pa1.setSexo("M");
        assertEquals(pa1.getPrima(), 500);

}
```

Tabla 11 Las pruebas unitarias satisfacen 100% de cobertura de Decisión-Condición. Pero la herramienta de cobertura de código usada no mide este criterio (Codecover). Aún así existen otras herramientas que sí lo hacen.

Con estas conjunto de pruebas unitarias cada condición y decisión toma todos los posibles resultados al menos una sola vez.

Como se pude ver para satisfacer criterios más fuertes generalmente se necesitan más casos de prueba. Pero los componentes quedan mejor probados y con menos huecos en donde los defectos puedan esconderse.

6.6 Corregir los defectos

El siguiente cambio al código fuente del ejemplo 6.5 inyecta un defecto:

Rutina

```
1 public int getPrima() {
2       int prima = 500;
3       if ((edad < 25) && !casado && sexo.equals("M")) {
4               prima += 1500;
5       } else {
6               if (casado || sexo.equals("F")) {
7                       prima -= 200;
8               }
9               if ((edad > 45) && (edad < 65)) {
10                      prima -= 100;
11              }
12      }
13      return prima;
14 }
```

Cuando las pruebas unitarias diseñadas para la cobertura de Decisión son ejecutadas un error es revelado y una de las pruebas falla.

ID del caso de prueba	Descripción	Entradas	Salida esperada	Pasa/ Falla
CP1	Un conductor masculino y soltero de 20 años de edad	Edad = 20, Soltero, masculino	2000	P
CP2	Un conductor masculino y casado de 45 años de edad	**Edad = 45**, Casado, masculino	200	**F**
CP3	Un conductor masculino y soltero de 35 años de edad	Edad = 35, Soltero, masculino	500	P

Pruebas unitarias	Resultado del Test runner para Junit

```java
public void testDecision(){
        // CP1
        pa1.setEdad(20);
        pa1.setCasado(false);
        pa1.setSexo("M");
        assertEquals(pa1.getPrima(), 2000);

        // CP2
        pa1.setEdad(45);
        pa1.setCasado(true);
        pa1.setSexo("M");
        assertEquals(pa1.getPrima(), 200);

        // CP3
        pa1.setEdad(35);
        pa1.setCasado(false);
        pa1.setSexo("M");
        assertEquals(pa1.getPrima(), 500);
}
```

JUnit | Package Explorer

Finished after 0.119 seconds

Runs: 4/4 Errors: 0 Failures: 1

testStatement - log240.assurance.PersonneAssureeTestBoiteBlanc

testDecision - log240.assurance.PersonneAssureeTestBoiteBlanch

testCondition - log240.assurance.PersonneAssureeTestBoiteBlanc

testDecisionCondition - log240.assurance.PersonneAssureeTestBc

Tabla 12 Al ejecutar las pruebas unitarias el Test runner muestra un defecto. Resultado esperado: 200 Resultado obtenido: 300

Una vez que se detecta que el error está en la implementación y no en la prueba unitaria. El siguiente paso es localizar el defecto.

```
Line 9          if ((edad > 45) && (edad < 65))
```

Ya encontrado el defecto, debes verificar si no existe otro defecto similar en el código vecino. Si no es el caso entonces debes realizar las correcciones pertinentes y volver a ejecutar nuevamente el conjunto de pruebas unitarias para ver si las correcciones tuvieron efecto y no se alteró ninguna funcionalidad previa.

```
Line 9          if ((edad >= 45) && (edad < 65))
```

ID del caso de prueba	Descripción	Entradas	Salida esperada	Pasa/ Falla
CP1	Un conductor masculino y soltero de 20 años de edad	Edad = 20, Soltero, masculino	2000	P

CP2	Un conductor masculino y casado de 45 años de edad	Edad = 45, Casado, masculino	200	P
CP3	Un conductor masculino y soltero de 35 años de edad	Edad = 35, Soltero, masculino	500	P

<table>
<tr><th>Pruebas unitarias</th><th>Resultado del Test runner para Junit</th></tr>
<tr><td><pre>
public void testDecision(){
        // CP1
        pa1.setEdad(20);
        pa1.setCasado(false);
        pa1.setSexo("M");
        assertEquals(pa1.getPrima(), 2000);

        // CP2
        pa1.setEdad(45);
        pa1.setCasado(true);
        pa1.setSexo("M");
        assertEquals(pa1.getPrima(), 200);

        // CP3
        pa1.setEdad(35);
        pa1.setCasado(false);
        pa1.setSexo("M");
        assertEquals(pa1.getPrima(), 500);
}
</pre></td><td>JUnit | Package Explorer
Finished after 0.135 seconds
Runs: 4/4 Errors: 0 Failures: 0
testStatement - log240.assurance.PersonneAssureeTestBoiteBlanc
testDecision - log240.assurance.PersonneAssureeTestBoiteBlanch
testCondition - log240.assurance.PersonneAssureeTestBoiteBlanc
testDecisionCondition - log240.assurance.PersonneAssureeTestBo</td></tr>
</table>

Tabla 13 Una vez corregido el error. Se vuelven a ejecutar las pruebas unitarias y todas pasan de manera exitosa.

Ahora que tu componente ha pasado todas las pruebas, debes decidir si continúas creando más casos de prueba o si tienes suficientes. Esto depende del criterio de salida que tu Administrador de proyecto haya definido.

6.7 Ejemplos de Test runners

Test runner de línea de comandos

El siguiente es un ejemplo de ejecución de una prueba unitaria en rubyUnit desde línea de comandos.

```
>ruby testrunner.rb c:/ejemplos/pruebas/MisPruebasUnitarias.rb
Loaded suite MisPruebasUnitarias
Started
....Finished in 0.014 seconds.4 tests, 5 assertions, 0 failures, 0 errors
>Exit code: 0
```

La primera línea es la ejecución de las pruebas desde el prompt. En este ejemplo se corre un conjunto de pruebas unitarias definidas en MisPruebasUnitarias. Las siguientes dos líneas son el comienzo de la ejecución. Las últimas dos líneas son un resumen del resultado obtenido. Típicamente el código de salida contiene el número de pruebas que fallaron.

Test runner gráfico

Las siguientes son capturas de pantalla de dos Test runners gráficos.

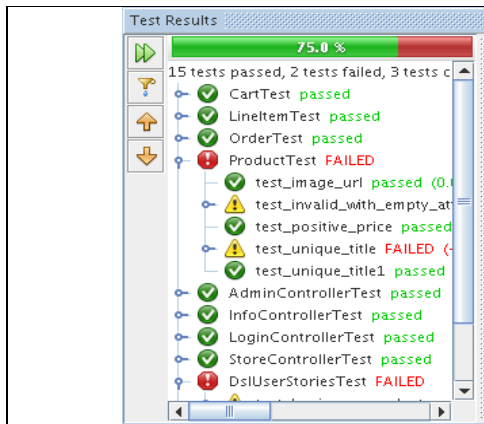


Figure 5. Test runner para Ruby on Rails

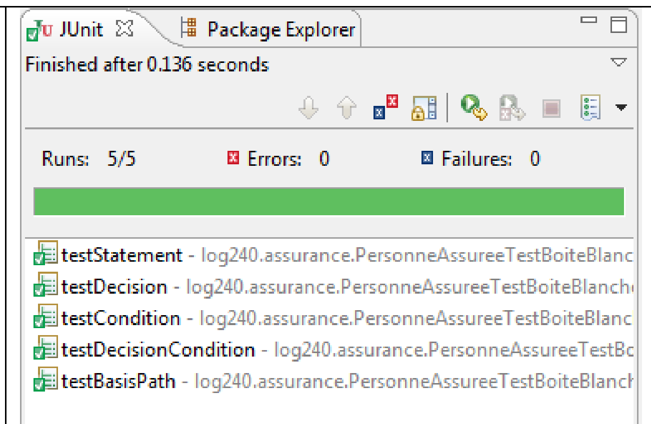
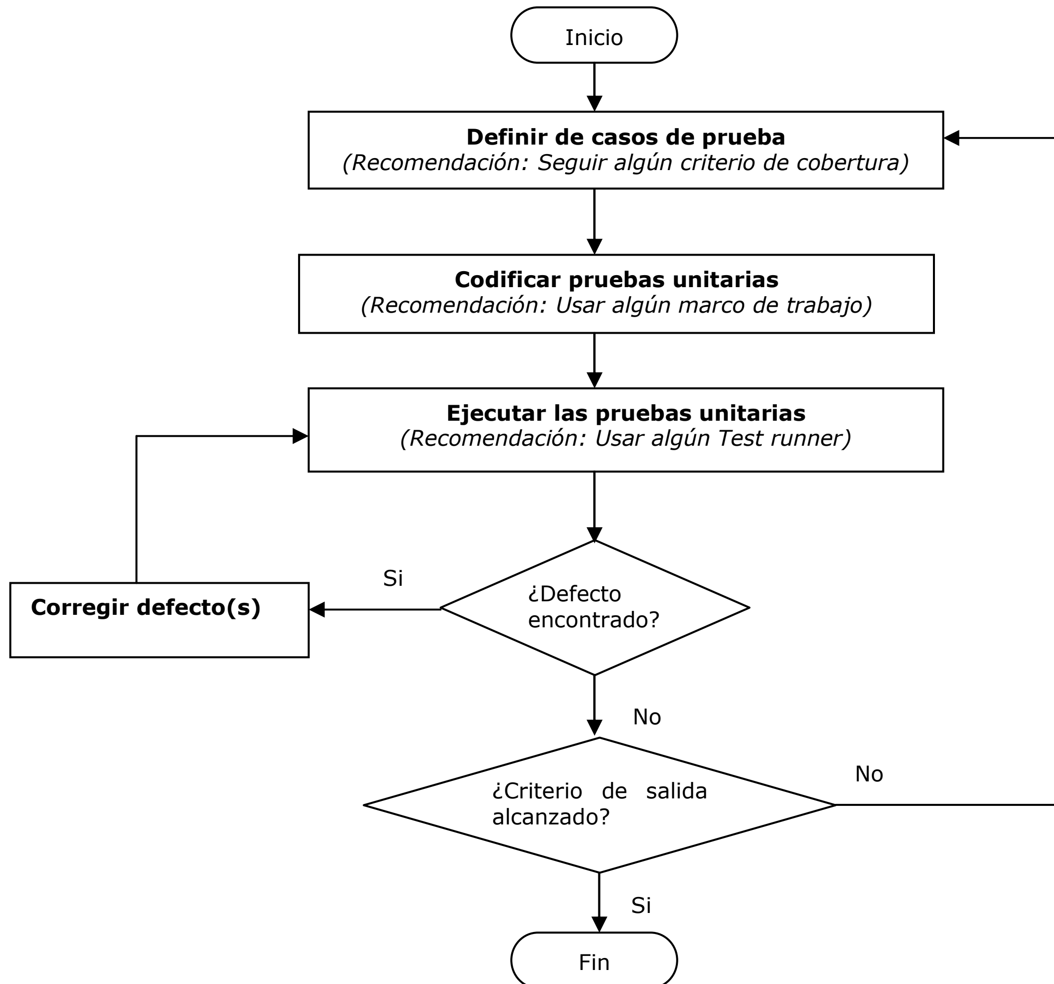


Figure 6. Test runner para Junit4 en Eclipse Java EE IDE

6.8 Ejemplo de una macro para aserciones en C++

```
#define ASSERT ( condicion, mensaje ){
      if( !(condicion) ){
            fprintf (stderr, "La Asercion %s ha fallado: %s\n",
                     #condicion, mensaje);
            exit( SALIDA_POR_FALLO );
        }
}
```

6.9 Ciclo de vida de las pruebas unitarias



7. Listas de verificación

Listas de verificación de tareas

7.1 Asignar tareas a los miembros del equipo de trabajo

☐	¿Cuentas ya con la documentación del diseño de software?
☐	¿Has seleccionado ya una estrategia de integración?
☐	¿Has detallado ya el calendario del proyecto?
☐	¿Has asignado ya las tareas a los miembros del equipo de trabajo?
☐	¿Has definido ya un criterio de salida para las pruebas unitarias?

7.2 Construir o actualizar los componentes de software

☐	¿Has entendido el diseño detallado del componente y su contribución?
☐	¿Has buscado funcionalidades disponibles en las librerías estándar?
☐	¿Has definido la lógica del componente?
☐	¿Has seguido el estándar de construcción mientras codificas?
☐	¿Has verificado el componente?

7.3 Diseñar o actualizar casos de pruebas unitarias y aplicarlos

☐	¿Cuentas con un criterio de salida?
☐	¿Has diseñado los casos de prueba?
☐	¿Has codificado las pruebas unitarias?
☐	¿Has ejecutado las pruebas unitarias?
☐	¿Has analizado los resultados?

7.4 Corregir los defectos

☐	¿Has confirmado el defecto?
☐	¿Has determinado la ubicación del defecto?
☐	¿Has buscado errores similares alrededor del código donde encontraste el defecto?
☐	¿Has corregido el error desde el origen y no solo un parche?
☐	¿Has verificado las correcciones?
☐	¿Tus correcciones cambian drásticamente el diseño del componente? (Si es así, coméntaselo a tu Líder técnico)

Listas de verificación de Soporte

7.5 Revisión de código

Nota: Esta lista de verificación puede ser adaptada para aplicarse a la mayoría de los lenguajes de programación. Una etiqueta es añadida enfrente de cada elemento de la lista para ayudar a reconocer los registros.

Etiqueta - Elemento		**Descripción**
☐	**RC1 Completo**	- Verifica que todas las funcionalidades en el diseño hayan sido codificadas y que todas las funciones y procedimientos necesarios hayan sido implementados
☐	**RC2 Lógica**	- Verifica que el flujo del programa y que todos los procedimientos y funciones sean consistentes con el diseño detallado.
☐	**RC3 Ciclos**	- Asegúrate de que cada ciclo sea apropiadamente inicializado y terminado. - Verifica que cada ciclo sea ejecutado el número correcto de veces.
☐	**RC4 Llamadas**	- Revisa cada llamada a función y procedimiento para asegurarte que es compatible con el estándar de construcción
☐	**RC5 Declaraciones**	Verifica cada variable y parámetro: - Tiene exactamente una declaración - Es usada solo dentro del alcance declarado - Es escrita correctamente en todo donde es usada
☐	**RC6 Inicializaciones**	- Revisa que cada variable sea inicializada
☐	**RC7 Limites**	- Verifica todas las variables, arreglos e índices para asegurarte que su uso no excede los límites declarados
☐	**RC8 Inicio-fin**	- Revisa que cada par de llaves, paréntesis y equivalentes sean cerrados correctamente, incluyendo casos donde *ifs* anidados puedan malinterpretarse.
☐	**RC9 Booleanos**	- Revisa las condiciones booleanas
☐	**RC10 Formato**	- Verifica que el formato de cada línea esté acorde al estándar de construcción.
☐	**RC11 Apuntadores**	- Revisa que todos los apuntadores sean usados apropiadamente
☐	**RC12 Entradas-salidas**	- Revisa todos los formatos de entrada y salida
☐	**RC13 Escritura**	- Asegúrate que cada variable o parámetro sea escrito apropiadamente.
☐	**RC14 Comentarios**	- Cerciórate que todos los comentarios sean acorde al estándar.
☐	**RC15 Cálculos**	- Verifica que todas las operaciones aritméticas y ecuaciones expresen lo que algoritmo indica
☐	**RC16 Base de datos**	- Asegúrate que todas las conexiones sean cerradas. - Verifica que la conexión del controlador sea escrita correctamente.

7.6 Lo que el arquitecto y el diseñador deben proveer

Requerimientos

☐	¿Debe algún requerimiento ser especificado en más detalle antes de su construcción?
☐	¿Son las entradas especificadas apropiadamente, incluyendo su origen, precisión, rango de valores y frecuencia?
☐	¿Son las salidas especificadas apropiadamente, incluyendo su destino, precisión, rango de valores, frecuencia y formato (incluyendo páginas web, reportes, etc.)?
☐	¿Son especificadas todas las interfaces externas de software y hardware?
☐	¿Es especificado el nivel máximo de memoria y/o almacenamiento?
☐	¿Son especificadas las consideraciones de tiempo? Tales como tiempo de respuesta esperado, tiempo de procesamiento, tasa de transferencia, etc.
☐	¿Es especificado el nivel de seguridad deseado?

Arquitectura y Diseño

☐	¿Es el diseño de datos o las clases más críticas descritas y justificadas?
☐	¿Es especificada la organización y el contenido de la base de datos?
☐	¿Es descrita alguna estrategia para el diseño de la interfaz de usuario?
☐	¿Es descrita y justificada alguna estrategia para el manejo de elementos de Entrada/Salida?
☐	¿Son descritos los requerimientos de seguridad en la arquitectura?
☐	¿Se provee una estrategia coherente para el manejo de errores?

Estándar de interfaz de usuario

☐	¿Cuentas ya con algún estándar de interfaz?
☐	¿Has seleccionado ya un estándar de interfaz?
☐	¿Cuentas ya con la aprobación del cliente?
☐	¿Tus programadores ya han adoptado el estándar?
☐	¿Has verificado la adopción del estándar?

8. Herramientas

8.1 Matriz de trazabilidad

- Objetivos:
  - Mantener el vínculo que existe entre cada requerimiento y su seguimiento a través de su descomposición, implementación y prueba (verificación)
  - Asegurar que todos los requerimientos tengan seguimiento y que sólo lo que sea requerido sea desarrollado.
  - Útil cuando se realizan evaluaciones de impacto de los requerimientos, diseño u otros cambios configurados en los elementos.

Nota: Una matriz de trazabilidad es una herramienta ampliamente usada para implementar el registro de la trazabilidad

Matriz de Trazabilidad

Fecha (dd-mm-aa): ______________

Nombre del proyecto:____________________________

	Nombre	**Firma**	**Fecha (dd-mm-aa)**
Verificado por:	______________________	______________________	______________
Aprovado por:	______________________	______________________	______________

Numero de Identificación	Texto de la necesidad	Texto del requerimiento	Método de Verificación	Título o ID del Caso de Uso	Título o ID del Componente de Código	Título o ID del Caso de Prueba	Fecha de Verificación	Nombre de la persona que realiza la verificación	Resultado de la Verificación

Instrucciones	
La tabla anterior debe ser creada en una hoja de cálculo o una base de datos de tal forma que pueda ser organizada por columnas para tener una trazabilidad bidireccional. Los identificadores únicos de cada fila deben ser asignados de manera jerárquica para que los elementos de bajo nivel (más detallados) puedan ser trazados a elementos de más alto nivel.	
Número de Identificación (ID)	Identificador único por medio del cual el requerimiento puede ser referenciado.
Texto de la necesidad	Describe brevemente cual es la necesidad que se tiene
Texto del requerimiento	Describe el requerimiento que pretende cubrir la necesidad suscitada.

Método de verificación	Métodos de verificación para el seguimiento del requerimiento: Prueba (P), Demostración (D), Análisis (A), Simulación (S), Inspección (I)
Título o ID del caso de uso	Título o ID del Caso de Uso que atiende al requerimiento
Título o ID del componente de código	Título o ID del Componente de código que atiende al requerimiento
Título o ID del caso de prueba	Título o ID del caso de prueba diseñado para probar el componente que atiende al requerimiento
Fecha de verificación	Fecha en la que se realizó la verificación
Nombre de la persona que realiza la verificación	Nombre de la persona que realiza la verificación del requerimiento. Es diferente de la persona que verifica la matriz de trazabilidad
Resultado de la verificación	El resultado de la verificación puede ser: (E) Éxito o (F) Fracaso, dependiendo del resultado esperado en la verificación.

Directrices La trazabilidad de requerimientos debe:

- Asegurar la trazabilidad para cada nivel de descomposición realizada en el proyecto. En particular:
    - Asegurar que cada requerimiento de bajo nivel pueda ser trazado a un requerimiento de alto nivel o a su fuente original.
    - Asegurar que cada diseño, implementación, y elemento de prueba pueda ser trazado a un requerimiento.
    - Asegurar que cada requerimiento tenga una representación en diseño e implementación
    - Asegurar que cada requerimiento sea probado/verificado
- Asegurar que la trazabilidad sea usada en la realización de evaluaciones de impacto de cambios de requerimientos o planes de proyecto, actividades y productos de trabajo.
- Tener mantenimiento y ser actualizada cuando ocurran cambios
- Ser consultada durante la preparación de evaluaciones de impacto para cada cambio propuesto al proyecto.
- Ser planeada, ya que el mantener los vínculos y referencias es una labor intensiva que debe ser monitoreada y debe ser asignada a un miembro del equipo del proyecto
- Ser resguardada como un documento electrónico

8.2 Herramientas de Cobertura de código

Vínculos a las herramientas más populares de cobertura de código:

Herramienta	Lenguaje	Fuente
CodeCover	Java	http://codecover.org/
Bullseye	C/C++	http://www.bullseye.com/
Cobertura	Java	http://cobertura.sourceforge.net/
Ncover	C# .NET	http://www.ncover.com/
PHPUnit – Xdebug	PHP	http://xdebug.org/

8.3 Marcos de trabajo para pruebas unitarias (Frameworks)

Vínculos a los frameworks de pruebas unitarias más populares:

Herramienta	Lenguaje	Fuente
JUnit	Java	http://www.junit.org/
NUnit	C# .NET	http://www.nunit.org/
DUnit	Delphi	http://dunit.sourceforge.net/
CppUnit 2	C++	https://launchpad.net/cppunit2
PHPUnit	PHP	http://phpunit.sourceforge.net/
PyUnit	Python	http://pyunit.sourceforge.net/

9. Referencias a otros Estándares y Modelos

Esta sección proporciona referencias de éste Paquete de Puesta en Operación a una selección de estándares ISO e ISO/IEC y a la Integración de Modelos de Madurez de Capacidades (CMMI®[8] - Capability Maturity Model Integration[SM]) versión 1.2, del Instituto de Ingeniería de Software (SEI – Software Engineering Institute).

Notas:

- Esta sección es puramente informativa
- Sólo las áreas cubiertas por el Paquete de Puesta en Operación son listadas en cada tabla
- Las tablas usan la siguiente convención:
  - Totalmente cubierta = T
  - Parcialmente cubierta = P
  - No cubierta = N

ISO 9001 Matriz de Referencia

Título de la tarea y paso	Cobertura T/P/N	Cláusula de ISO 9001	Comentarios
SI.2.2 Documentación o actualización de la Especificación de Requerimientos.	P	7.3.2 Diseño y desarrollo de entradas d) Otros requerimientos esenciales para el diseño y desarrollo	
Sub-tarea: Definir estándares de construcción.			En la mayoría de los casos no son especificados por los clientes, pero son esenciales para algunos componentes.
SI.4.1 Asignar tareas a los miembros del equipo de trabajo relacionadas con su rol, de acuerdo al Plan del proyecto actual.	P	7.3.1 Diseñar y desarrollar la planeación	Sólo incluye tareas de comunicación

[SM] Integración CMM es un servicio de la universidad de Carnegie Mellon.

® El Modelo de Integración de Capacidades y Madurez, CMMI está registrado en la oficina de patentes norteamericana por la universidad de Carnegie Mellon.

SI.4.3 Construir o actualizar *Componentes de Software* basados en la parte detallada del *Diseño de Software.*	P	7.3.3 Diseñar y desarrollar salidas. a) Conocer los requerimientos de entrada para el diseño y desarrollo.	
SI.4.4 Diseñar o actualizar casos de pruebas unitarias y aplicarlos para verificar que los *Componentes de Software* implementen la parte detallada del *Diseño de Software.*	P	7.3.4 Revisión del Diseño y desarrollo a) para evaluar la habilidad de los resultados del diseño y desarrollo en el cumplimiento de los requerimientos, b) Identificar cualquier problema y proponer las acciones necesarias	
SI.4.5 Corregir los defectos encontrados hasta que todas las pruebas unitarias pasen satisfactoriamente (alcanzando un criterio de salida).	P	7.3.4 Revisión del Diseño y desarrollo a) para evaluar la habilidad de los resultados del diseño y desarrollo en el cumplimiento de los requerimientos, b) Identificar cualquier problema y proponer las acciones necesarias	
SI.4.6 Actualizar el Registro de Trazabilidad incorporando los *Componentes de Software* construidos o modificados.	P	7.3.7 Administrar los cambios en el diseño y desarrollo	

ISO/IEC 12207 Matriz de Referencia

Título de la tarea y paso	**Cobertura T/P/N**	**Cláusula de ISO/IEC 12207**	**Comentarios**
SI.2.2 Documentación o actualización de la Especificación de Requerimientos.	P	7.1.2.3.1 Análisis de requerimientos de Software 7.1.2.3.1.1 El ejecutor debe establecer y documentar requerimientos de software (incluyendo las especificaciones de características de calidad) descritas a continuación: b) Interfaces externas al componente de software k) Requerimientos de mantenimiento para el usuario.	a) Sólo cubre requerimientos relacionados con la fase de construcción b) Está relacionado con interfaces de usuario y con estándares de construcción.

Sub-tarea: Definir estándares de construcción.		7.1.5.3.1 Construcción de Software 7.1.5.3.1.5 El ejecutor debe evaluar el código y los resultados de las pruebas considerando los criterios listados a continuación. Los resultados de las evaluaciones deben ser documentadas. e) Uso de métodos y estándares apropiados para codificación.	Los estándares de construcción serán útiles cuando se realice la sección 7.1.5.3.1.5
SI.4.1 Asignar tareas a los miembros del equipo de trabajo relacionadas con su rol, de acuerdo al Plan del proyecto actual.	P	6.3.1.3.3 Activación del proyecto 6.3.1.3.3.3 El administrador debe iniciar la implementación y el conjunto de criterios, ejerciendo control sobre el proyecto.	Esta tarea sólo activa la fase de construcción.
SI.4.3 Construir o actualizar *Componentes de Software* basados en la parte detallada del *Diseño de Software.*	P	7.1.5.3.1 Construcción de Software 7.1.5.3.1.1 El ejecutor debe desarrollar y documentar lo siguiente: a) Cada unidad de software y base de datos	
SI.4.4 Diseñar o actualizar casos de pruebas unitarias y aplicarlos para verificar que los *Componentes de Software* implementen la parte detallada del *Diseño de Software.*	P	7.1.5.3.1 Construcción de Software 7.1.5.3.1.1 El ejecutor debe desarrollar y documentar lo siguiente: b) Procedimientos de prueba y datos para probar cada unidad de software y base de datos.	
SI.4.5 Corregir los defectos encontrados hasta que todas las pruebas unitarias pasen satisfactoriamente (alcanzando un criterio de salida).	P	7.1.5.3.1 Construcción de Software 7.1.5.3.1.2 El ejecutor debe probar cada componente de software y de base de datos para asegurar que satisface los requerimientos. Los resultados de prueba deben ser documentados.	
SI.4.6 Actualizar el Registro de Trazabilidad incorporando los *Componentes de Software* construidos o modificados.	P	7.1.5.3.1 Construcción de Software 7.1.5.3.1.5 El ejecutor debe evaluar el código y los resultados de las pruebas considerando los criterios listados a continuación. Los resultados de las evaluaciones deben ser documentadas. a) Trazabilidad a los requerimientos y diseño del componente de software.	

CMMI Matriz de referencia

Título de la tarea y paso	Cobertura T/P/N	PA/Objetivo/ Práctica de CMMI V1.2	Comentarios
SI.2.2 Documentación o actualización de la Especificación de Requerimientos.	P	Desarrollo de Requerimientos (DR) SG 1 Desarrollo de los requerimientos del cliente SP 1.2 Desarrollo de los requerimientos del cliente	Las restricciones para verificación y validación no son cubiertas.
Sub-tarea: Definir estándares de construcción.			Los stakeholders de la fase de construcción deben proporcionarlos.
SI.4.1 Asignar tareas a los miembros del equipo de trabajo relacionadas con su rol, de acuerdo al Plan del proyecto actual.	P	Soluciones Técnicas (ST) SG 3 Implementar el Diseño del Producto GP 2.8 Monitorear y Controlar el Proceso	Sólo activa la ejecución del plan.
SI.4.3 Construir o actualizar *Componentes de Software* basados en la parte detallada del *Diseño de Software.*	P	Soluciones Técnicas (ST) SG 3 Implementar el Diseño del Producto SP 3.1 Implementar el Diseño	Cubre las sub-prácticas 1. Usar métodos efectivos para implementar los componentes del producto y 2. Adherirse a estándares aplicables y criterios.
SI.4.4 Diseñar o actualizar casos de pruebas unitarias y aplicarlos para verificar que los *Componentes de Software* implementen la parte detallada del *Diseño de Software.*	P	Soluciones Técnicas (ST) SG 3 Implementar el Diseño del Producto SP 3.1 Implementar el Diseño	Cubre la sub-práctica 4. Desarrollar las pruebas unitarias del componente del producto apropiadamente

SI.4.5 Corregir los defectos encontrados hasta que todas las pruebas unitarias pasen satisfactoriamente (alcanzando un criterio de salida).	P	Soluciones Técnicas (ST) SG 3 Implementar el Diseño del Producto SP 3.1 Implementar el Diseño	Cubre la sub-práctica 4. Desarrollar las pruebas unitarias del componente del producto apropiadamente
SI.4.6 Actualizar el Registro de Trazabilidad incorporando los *Componentes de Software* construidos o modificados.	P	Administración de Requerimientos (AREQ) SG 1 Administrar Requerimientos SP 1.4 Mantener trazabilidad bidireccional de requerimientos	Sólo cubre la trazabilidad de los componentes construidos.

10. Referencias

Clave	Referencia
[Code Complete]	Steve McConnell, Code Complete, Second Edition, Redmond, Washington, Microsoft Press, 2004.
[Art of Software testing]	Glenford J. Myers, The Art of Software Testing, Second Edition, 2004.
[Practitioner's Guide]	Lee Copeland, A Practitioner's Guide to Software Test Design, 2004
[Defect Prevention]	Marc McDonald, The Practical Guide To Defect Prevention, 2008
[Introduction to Software Testing]	Paul Ammann & Jeff Offutt, Introduction to Software testing, 2008
[Testing Computer Software]	Cem Kaner, Testing Computer Software
[Pratical Software Testing]	Ilene Burnstein, Practical Software Testing, 2002
[SE Support Activities for VSE]	Vicent Ribaud, Software Engineering Support Activities for Very Small Entities, 2010
[Application of ISES in VSE]	Claude Y. Laporte, The application of International Software Engineering Standards in Very Small Enterprises, 2008
[A SE Lifecycle Standard for VSEs]	Claude Y. Laporte, A Software Engineering Lifecycle Standard for Very Small Enterprises, 2008
[Misuse Code Coverage]	Brian Marick, How to Misuse Code Coverage, 1999
[IEEE 1012-2004]	IEEE 1012-2004 IEEE Standard for Software Verification and Validation, IEEE Computer Society
[ISO/IEC 12207]	ISO/IEC 12207:2008 Systems and software engineering – Software life cycle processes.
[ISO/IEC 29110-5-1-2]	ISO/IEC TR 29110-5-1-2:2011, Software Engineering—Lifecycle Profiles for Very Small Entities (VSEs) – Part 5-1-2: Management and Engineering Guide – Basic VSE Profile
[ISO/IEC 24765]	ISO/IEC 24765:2011 Systems and software engineering vocabulary
[ISO/IEC 20926]	ISO/IEC 20926:2003 Software engineering -- IFPUG 4.1 Unadjusted functional size measurement method -- Counting practices manual
[ISO/IEC 29881:2008]	ISO/IEC 29881:2008 Information technology--Software and systems engineering--FiSMA 1.1 functional size measurement method,
[IEEE 1233-1998]	IEEE Guide for Developing System Requirements Specifications

11. Formulario de Evaluación

Paquete de Puesta en Operación: Construcción y pruebas unitarias Versión 0.5 Tus respuestas nos ayudaran a mejorar el Paquete, tus comentarios y sugerencias son bien recibidas.
1. ¿Qué tan satisfecho estás con el CONTENIDO de éste Paquete de Puesta en Operación? e *Muy Satisfecho* e *Satisfecho* e *Ni Satisfecho ni Insatisfecho* e *Insatisfecho* e *Muy Insatisfecho*
2. La secuencia en la que los tópicos son discutidos, ¿Son lógicos y fáciles de seguir? e *Muy Satisfecho* e *Satisfecho* e *Ni Satisfecho ni Insatisfecho* e *Insatisfecho* e *Muy Insatisfecho*
3. ¿Qué tan satisfecho estuviste con la APARIENCIA/FORMATO de éste Paquete de Puesta en Operación? e *Muy Satisfecho* e *Satisfecho* e *Ni Satisfecho ni Insatisfecho* e *Insatisfecho* e *Muy Insatisfecho*
4. ¿Existe algún tópico innecesario dentro del paquete? (Por favor descríbelo)
5. ¿Qué otro tópico crees que deba incluir el paquete? (Por favor descríbelo) • Tópico propuesto: • Justificación para el nuevo tópico:
1. ¿Existe algún error en el Paquete? • Por favor indícalo: • Descripción del error: • Ubicación del error (sección #, figura #, tabla #) :
7. Otros cometarios o sugerencias:
8. ¿Recomendarías este Paquete de Puesta en Operación a algún colega de otra VSE? e *Definitivamente* e *Probablemente* e *No estoy seguro* e *Probablemente No* e *Definitivamente No*

Opcional

- Nombre: ____
- Correo electrónico: ___________________________________

Manda este formulario a: thexaviermail@gmail.com o Avumex2003@yahoo.com.mx o hanna.oktaba@ciencias.unam.mx

9. Anexo 2 – Ejemplo de FMEA

Siempre se elige el riesgo potencial más alto de un modo de fallo potencial para determinar la prioridad de mitigación. Sin embargo del siguiente ejemplo aunque ambos modos de fallo tienen la misma calificación de riesgo se atienden primero los que generen más efectos potenciales (Marc & Robert, 2008).

Hoja de trabajo FMEA	**Producto: Pastel de Leche**									
Responsable:	Juan Pérez									
Fecha de creación	12/12/12									
Paso del proceso, Función o Tarea	**Modo de fallo potencial**	**Efectos potenciales de fallo**	**Rango de impacto**	**Causas potenciales**	**Rango de probabilidad**	**Controles actuales**	**Rango de detectabilidad**	**Prioridad del riesgo**	**Mejora**	**Propietario**
Añadir leche al pastel de leche	Cantidad incorrecta de leche	Pastel demasiado seco o demasiado húmedo	3	La taza medidora tiene marcas pequeñas	5	Ninguno	3	45	Usar impresiones más largas de las medidas de las tazas	Pedro López
			3	Marcas borrosas en la taza medidora	3	Inspección Visual	2	18	Remplazar las tazas que tienen las medidas borrosas	
			3	Derramo de leche	2	Ninguno	4	24	Entrenar a los panaderos	
	La harina continua en la taza medidora	Muy poca leche – Pastel muy seco	3	Descuido del empleado	3	Entrenamiento	5	45	Cambiar estándares, procedimientos de operación y mejorar El programa de entrenamiento.	
		Grumos en la leche	3	Descuido del empleado	1	Entrenamiento	5	15		

Tabla 24 Ejemplo de un FMEA para la preparación de un pastel de leche.

10. Anexo 3 – Cuestionario de Exploración

Encuesta: Evaluación de la comprensión del Paquete de Puesta en Operación de Construcción y Pruebas unitarias del estándar ISO/IEC 29110 Perfil básico.

Universidad Nacional Autónoma de México
______de __________de _______

Cuestionario 1 de 2 (Exploración de conocimientos y experiencia)

Instrucciones: Contesta las siguientes preguntas tachando con una "X" sobre el símbolo ○ de la respuesta que consideres más adecuada. Las preguntas que tengan el símbolo ☐ entre sus opciones pueden tener más de una respuesta.

1. ***¿Cuánto tiempo llevas como programador de software?***

○ Menos de 1 año

○ 1 – 3 años

○ 4 – 10 años

○ Más de 10 años

2. ***¿Qué roles o papeles desempeñas en tu organización?* Marca todos los que desempeñes.**

☐ Programador
☐ Analista
☐ Diseñador
☐ Líder Técnico
☐ Administrador de Proyectos

3. ***¿En un proyecto común, que porcentaje de tu tiempo le dedicas a la programación de software?***

○ Menos de 30%

○ De 30% a 60%

○ Más de 60%

4. ***¿Qué opinas acerca de la realización de las pruebas unitarias?***

○ Son indispensables para todo proyecto

○ Pueden ser útiles en algunas ocasiones

○ Representan una pérdida de tiempo

○ No sé, desconozco el tema

5. ***¿Consideras que tú como programador necesitas mejorar, aprender o repasar prácticas de construcción de software?***

○ Sí, para desarrollarme profesionalmente

○ Sí, para ser más productivo en mi organización

○ No, considero que mis conocimientos son suficientes

6. ***¿Sigues algún estándar de construcción de código (formato de codificación, comentarios, estructuras de control, nombres de variables, constantes, etc.)?***

○ Sí, en la organización contamos con un estándar

○ Sí, yo sigo mi propio estándar

○ No sigo ningún estándar

7. ***¿Qué haces cuando encuentras algún defecto en los requerimientos, en el diseño o en la arquitectura?***

○ Lo corrijo yo mismo

○ Lo reporto a mi líder técnico o a mi superior a cargo

○ Lo ignoro

8. ***¿Qué estrategia de integración de componentes de software es la que sigues generalmente?***

○ Bottom-up (Ascendente)

○ Top-down (Descendente)

○ Otra

○ No sigo ninguna estrategia de integración

○ Desconozco que es una estrategia de integración

9. ***¿Te indican de alguna forma la dificultad de los componentes de software que te asignan?***

○ Sí, de manera escrita

○ Sí, de manera verbal

○ No me mencionan nada

10. ***¿Has usado el enfoque de Pseudocódigo para la generación de componentes de software?***

- ○ Es el enfoque que uso habitualmente
- ○ Lo he usado en ocasiones
- ○ Lo conozco pero no lo uso
- ○ Nunca he oído de él

11. ***¿Has usado el enfoque de Diagramas de flujo para la generación de componentes de software?***

- ○ Es el enfoque que uso habitualmente
- ○ Lo he usado en ocasiones
- ○ Lo conozco pero no lo uso
- ○ Nunca he oído de él

12. ***¿Has usado listas de verificación de código para buscar defectos en tus componentes?***

- ○ Las uso habitualmente
- ○ Las he usado en ocasiones
- ○ Las conozco pero no las uso
- ○ Nunca he oído de ellas

13. ***Cuando detectas una anomalía o defecto en tu código ¿cómo lo localizas?***

- ○ Con listas de verificación
- ○ A través del depurador
- ○ A prueba y error
- ○ Todas las anteriores

14. ***Después de que creas un componente de software ¿Existe alguna forma de que éste pueda ser mapeado a un elemento del diseño?***

- ○ Sí
- ○ No
- ○ Lo desconozco

15. ¿Sabes en qué consisten las pruebas de caja blanca?

○ Sí

○ Tengo una idea

○ No

16. ¿Has realizado pruebas unitarias al código que generas?

○ Sí

○ En ocasiones

○ Nunca

○ Desconozco que son las pruebas unitarias

Si tu respuesta de la pregunta anterior fue: *"Sí" o "En ocasiones"* contesta las siguientes preguntas, de lo contrario puedes dejarlas en blanco y saltarte directamente a la parte de información adicional.

17. ¿Cuáles de los siguientes pasos ejecutas cuando realizas las pruebas unitarias?

☐ Diseñar los casos de prueba

☐ Codificar las pruebas unitarias

☐ Ejecutar las pruebas unitarias

☐ Todos los anteriores

18. ¿Cómo decides cuando debes dejar de realizar pruebas unitarias a un componente?

○ Cuando de acuerdo a mi experiencia considero que he realizado suficientes

○ Cuando he cubierto el criterio que el Administrador de Proyecto me ha indicado

○ Cuando todas las pruebas que se me han ocurrido han pasado satisfactoriamente

○ Cuando he agotado el tiempo que tenía destinado para realizar las pruebas

19. ¿Sabes en qué consisten los criterios de cobertura de código?

○ Sí

○ Tengo una idea

○ No

20. ¿Has usado algún framework de pruebas unitarias para codificar las pruebas?

○ Sí

○ No

○ No sé que es un framework de pruebas unitarias

21. ¿Sabes qué un Test runner?

○ Sí

○ Tengo una idea

○ No

Información adicional:

Si tuviste algún problema con alguna pregunta, por favor escribe el número de la(s) pregunta(s) y una pequeña descripción sobre el problema que tuviste.

____

____

____

____

____

____

Si tienes algún comentario o sugerencia que pueda mejorar el cuestionario por favor escríbelo en las siguientes líneas

____

____

____

____

____

11. Anexo 4 Cuestionario de evaluación – Con respuestas

A continuación se presentan las preguntas que conforman la evaluación de la comprensión lectora que se realizó a los 5 participantes de la muestra. Cada color en la formulación de la pregunta representa un nivel de comprensión lectora que se atiende:

- ***Comprensión literal:***
  El lector hace valer sus destrezas de memoria y reconocimiento, al recordar detalles, reconocer ideas principales, relaciones causa-efecto etc.
- ***Reorganización de la información:***
  El lector ordena las ideas mediante procesos de clasificación y síntesis
- ***Comprensión inferencial:***
  El lector liga al texto su experiencia personal para realizar conjeturas e hipótesis, lo que le permite interpretar el texto al hacer deducciones de información e ideas que no aparecen de manera explícita en el texto.
- ***Lectura crítica o juicio valorativo***:
  Permite la reflexión sobre el contenido del texto. Para ello, el lector necesita establecer una relación entre la información del texto y los conocimientos que ha obtenido de otras fuentes, y evaluar las afirmaciones del texto contrastándolas con su propio conocimiento.
- ***Apreciación lectora***:
  Este nivel hace referencia al impacto psicológico y estético del texto en el lector, incluyendo el formato y estilo que el autor le da al texto

Los números que se encuentran a un lado de las respuestas corresponden al porcentaje y número de participantes que eligieron dicha respuesta como correcta. Los valores coloreados de azul significan que más del 50% de los participantes contestaron correctamente la pregunta, mientras que los valores coloreados de rojo significan que el 50% o menos de los participantes contestaron correctamente la pregunta.

Encuesta: Evaluación de la comprensión del Paquete de Puesta en Operación de Construcción y Pruebas unitarias del estándar ISO/IEC 29110 Perfil básico.

Universidad Nacional Autónoma de México
15 de Agosto de 2011

Cuestionario 2 de 2 (Evaluación de la comprensión lectora)

Instrucciones: Contesta las siguientes preguntas tachando con una “X” sobre el símbolo ○ de la respuesta que consideres más adecuada.

1. Con la falta de estándares de construcción el siguiente problema puede suscitarse:	
○ Comprender y modificar ciertas partes de código puede resultar complicado.	100% - 5 respuestas

○ Algunas partes del código pueden no funcionar correctamente.	
○ La complejidad del algoritmo aumenta	

2. Si no se tiene el tiempo suficiente para construir un estándar de construcción propio lo más recomendable es:	
○ Exigirle al cliente un estándar de construcción	
○ Actualizar el calendario del proyecto para asignar el tiempo necesario para construir un estándar propio	20% - 1 respuesta
○ Implementar algún componente y usarlo de ejemplo para la construcción del resto de los componentes	60% - 3 respuestas
○ No usar un estándar de construcción, al menos para ese proyecto	20% - 1 respuesta

3. Lo más importante de un estándar de construcción es:	
○ Su creación	
○ Su tamaño	
○ Su adopción	100% - 5 respuestas
○ Su distribución	

4. ¿Cuál es el orden que deben seguir los siguientes pasos para definir un estándar de construcción? ***a. Verificar la adopción de los estándares*** ***b. Planear la sub-tarea*** ***c. Obtener estándares disponibles*** ***d. Adoptar los estándares*** ***e. Seleccionar los estándares***	
○ a,b,c,d,e	
○ e,b,c,d,a	
○ b,c,d,e,a	
○ b,c,e,d,a	100% - 5 respuestas

5. Suponiendo que encuentras un defecto en la arquitectura de software y no puedes continuar con tu trabajo si el problema no es corregido. ¿Cuál de las siguientes medidas es la que tomas?	
○ Buscar alguna otra tarea independiente hasta que el problema sea solucionado	
○ Buscar una solución y corregir el defecto tu mismo	

○ Reportar el defecto a tu líder técnico	100% - 5 respuestas
○ Reportar el defecto directamente con el arquitecto a cargo	

6. ¿Cuál es el primer paso a seguir para reportar un defecto de una etapa previa?	
○ Escribir una descripción breve del defecto	60% - 3 respuestas
○ Escribir las posibles casusas del defecto	
○ Escribir dónde fue encontrado el defecto	
○ Confirmar el defecto	40% - 2 respuestas

7. ¿Cuáles de los siguientes pasos de la sub-tarea: reportar taxonomía de defectos de etapas previas, le corresponde a tu líder técnico? ***a. Confirmar el defecto*** ***b. Verificar el reporte del defecto*** ***c. Escribir una estrategia de mitigación*** ***d. Escribir las posibles causas*** ***e. Reportar el defecto al área encargada***	
○ a,c,d	
○ b,c,e	60% - 3 respuestas
○ a,c,e	20% - 1 respuesta
○ b,d,e	20% - 1 respuesta

8. ¿Cómo se conoce a la estrategia de integración de componentes de software en donde los módulos son integrados desplazándose hacia abajo a través de la estructura de control, repitiendo este proceso hasta que los componentes que se encuentran hasta abajo de la jerarquía son integrados?	
○ Top-down	80% - 4 respuestas
○ Bottom-up	
○ Big-bang	
○ Top-up	20% - 1 Respuestas

9. Suponiendo que tienes un pequeño sistema que se divide en cuatro componentes (todos con la misma importancia) y cada uno de ellos desempeña una función específica y bien definida. ¿Cuál de las siguientes estrategias de integración es la que más te conviene?	
○ Integración orientada a riesgos	40% - 2 respuestas
○ Integración orientada a características	60% - 3 respuestas

○ Bottom-up	
○ Top-down	

10. ¿Cuáles de los siguientes aspectos son los que más toma en cuenta el Administrador del proyecto para la asignación de tus tareas? ***a. Experiencia*** ***b. Habilidades*** ***c. Edad*** ***d. Conocimiento*** ***e. Salario***	
○ a,b,e	
○ c,d,e	
○ a,c,d	
○ a,b,d	100% - 5 respuestas

11. Es una herramienta que permite una inspección sistemática en la que los desarrolladores de software revisan su código con el objetivo de encontrar y corregir defectos superficiales al inicio de la fase de desarrollo	
○ Compilador	
○ Lista de revisión de código	100% - 5 respuestas
○ Catálogo de errores	
○ Taxonomía de defectos	

12. Antes de comenzar con la construcción del componente es necesario que:	
○ Comprendas los requerimientos no funcionales del componente	20% - 1 respuestas
○ Se tenga una lista con los lenguajes de programación que mejor puedan solucionar el problema	
○ Se entienda el diseño detallado del componente y su contribución	40% - 2 respuestas
○ Se tengan listas las pruebas unitarias	40% - 2 respuestas

13. Suponiendo que tienes a cargo la construcción de un componente crítico, el cual requiere de un algoritmo complejo (pero común) para cierto funcionamiento ¿Cuál de las siguientes medidas es la que tomas?	
○ Buscas el algoritmo en alguna biblioteca del lenguaje que usas o en otras fuentes confiables	100% - 5 respuestas

○ Propones tu propio algoritmo para conocer a detalle su funcionamiento	
○ Pides ayuda y junto con otro compañero diseñas el algoritmo	
○ Buscas un algoritmo más sencillo pero que es un poco menos eficiente	

14. Dos de los enfoques más ampliamente aceptados para la construcción de la lógica de los componentes de software son: ***a. Pseudocódigo*** ***b. Listas de verificación*** ***c. Programación estructurada*** ***d. Diagrama de flujo***	
○ a,b	20% - 1 respuestas
○ a,d	40% - 2 respuestas
○ a,c	40% - 2 respuestas
○ c,d	

15. ¿Cuál es el objetivo de las pruebas unitarias?	
○ Encontrar y corregir defectos introducidos durante la construcción a través del diseño y aplicación de casos de prueba.	60% - 3 respuestas
○ Encontrar defectos introducidos durante la construcción a través del diseño y aplicación de casos de prueba	20% - 1 respuestas
○ Corregir defectos introducidos durante la construcción a través del diseño de casos de prueba	
○Encontrar y corregir defectos introducidos durante la integración a través del diseño de casos de prueba	20% - 1 respuestas

16. En caso de que tu componente necesite de una interfaz de usuario ¿Cuál de las siguientes opciones consideras más recomendable?	
○ Buscas una plantilla amigable en la web	20% - 1 respuesta
○ Utilizar la interfaz del proyecto anterior	
○ Codificar la interfaz de acuerdo al estándar de interfaz	60% - 3 respuestas
○ Diseñar tu propia interfaz para tu componente en específico	20% - 1 respuesta

17. ¿Por qué es recomendable que los programadores realicen las pruebas unitarias de caja blanca?	
○ Porque pueden aprender fácilmente a codificar las pruebas unitarias	20% - 1 respuesta
○ Porque conocen el código, la estructura interna y la lógica de los componentes que ellos mismos codifican.	80% - 4 respuestas
○ Porque conocen perfectamente los requerimientos funcionales del componente	
○ Por su experiencia y conocimiento en el uso de frameworks	

18. Sirven como regla de paro para determinar si se han creado o no suficientes casos de prueba	
○ Frameworks de pruebas unitarias	
○ Pruebas unitarias	20% - 1 respuesta
○ Criterios de cobertura	80% - 4 respuestas
○ Aserciones	

19. Dada la siguiente pieza de código, ¿qué criterio de cobertura es el que mejor se adapta considerando los aspectos de eficiencia y esfuerzo invertido?

```
public static void Raices(double A, double B, double C)
{
    //Ax^2 + Bx + C = 0   considerando a Factor >= 0
    double factor=Math.pow(B,2) - 4*A*C;
    double re1=0,im1=0;
    String X1="",X2="";

      re1 = (-1*B)/(2*A);
      im1= (Math.sqrt(factor))/(2*A);
      X1 = "" + (re1+im1) + " + " + (im1-im1) + " i";
      X2 = "" + (re1-im1)  + " - " + (im1-im1) + " i";

    System.out.println("Tu polinomio es: "+A+"x^2 + ("+B+"x) + ("+C+") = 0");
    System.out.println("\nLas raices del polinomio son:\n");
    System.out.println("X1 = "+X1+"\n"+"X2 = "+X2+"\n");
}
```

Opción	
○ Brinco	
○ Decisión-Condición	60% - 3 respuestas
○ Decisión	
○ Sentencia	40% - 2 respuestas

20. Tipo de rutinas que usualmente toman dos argumentos: una expresión booleana que describe si la suposición esperada es verdadera y un mensaje que mostrar en caso contrario. Se usan para verificar si el caso de prueba detecta un defecto.	
○ Scripts de línea de comandos	
○ Pruebas unitarias	
○ Aserciones	100% - 5 respuestas
○ Test runner	

21. Algunas de las ventajas de un Framework de pruebas unitarias son: ***a. Mantienen desde el inicio separado el código de la implementación del código de las pruebas.*** ***b. Si existen cambios en la implementación, al ejecutar las pruebas nuevamente se revisan que todo continúe trabajando como lo previsto.*** ***c. Cuentan con una suite de aserciones que ayudan en la realización de las pruebas*** ***d. Son independientes del lenguaje de programación***	
○ Ninguna de las anteriores	
○ a,c,d	20% 1 respuesta
○ a,b,c	80% - 4 respuestas
○ Todas las anteriores	

22. El uso de herramientas de Cobertura de código es altamente recomendable porque:	
○ Proporciona los casos de prueba necesarios para satisfacer cada criterio de cobertura	60% - 3 respuestas
○ Reduce significativamente el número de casos de prueba	
○ Permite saber que tanto porciento de código ha sido cubierto por cada criterio de cobertura	40% - 2 respuestas
○ Permite saber que tanto porciento del componente ha sido construido	

23. Corregir los defectos encontrados por las pruebas unitarias, por la persona que se encuentra a cargo de la construcción del componente, es el método más rápido y menos costoso para corregir los defectos.

○ Verdadero	100% - 5 respuestas
○ Falso	

24. Cuál de las siguientes NO es una buena práctica para la localización de defectos:	
○ Reducir la región del código sospechoso	
○ Usar diferentes tipos de datos para reproducir el error	
○ Refinar los casos de prueba	
○ Hacer cambios experimentales al código de la implementación	100% - 5 respuestas

25. Suponiendo que tus pruebas unitarias ya no encuentran más defectos ¿Cuál sería tu siguiente paso?	
○ Liberar el componente	
○ Revisar que se haya alcanzado el criterio de salida definido para el componente, en caso contrario diseñar más casos de prueba.	60% - 3 respuestas
○ Escribir un reporte de las pruebas unitarias realizadas	40% - 2 respuestas
○ Diseñar todos los casos de prueba restantes para satisfacer el criterio de Decisión-Condición.	

26. Asegurar una trayectoria congruente entre los requerimientos, el diseño y la construcción de los componentes es el objetivo de:	
○ El diseño detallado	
○ El análisis de los requerimientos	
○ La construcción de software	
○ El registro de trazabilidad	100% - 5 respuestas

27. ¿Qué tan satisfecho estás con el CONTENIDO de éste Paquete de Puesta en Operación?	
○ Muy satisfecho	20% - 1 respuesta
○ Satisfecho	60% - 3 respuestas

○ Ni satisfecho ni insatisfecho	20% - 1 respuesta
○ Nada satisfecho	

28. ¿Cuál es tu opinión acerca del paquete y que temas consideras que deban quitarse, ampliarse o añadirse?
1.- Quitar manejo de pruebas o sólo mencionarlo breve es esto y sirve para esto; y ampliar introducción de paquete de prueba un poco, la presentación está bien pero resulta impactante ver el número de páginas al inicio
2.- La sección de integración creo que está de sobra
3.- Me parece excelente el paquete. Al inicio un poco aburrido, pero después se vuelve muy interesante.
4.- En general bueno y claro de entender, aunque un poco confusa la parte de "6.5 Diseño de casos de prueba, escritura de pruebas unitarias y uso de una herramienta de cobertura de código", a través de los diferentes ejemplos de cobertura a veces perdía el hilo o la secuencia del ejemplo (aunque hay que tener en cuenta que solo lo ley una vez) .
5.- Es un paquete de fácil lectura que te guía de una forma ordenada a través de las distintas fases de desarrollo

29. ¿Qué calificación le darías al formato, redacción y presentación del paquete?	
○ Muy bueno	
○ Bueno	100% - 5 respuestas
○ Aceptable	
○ Malo	
○ Muy malo	

30. ¿Encontraste algún error en el paquete? Por favor describe el error y su ubicación
Tabla 10, la descripción viene en inglés

12. Referencias

Anacleto, A. (2004). *Experiences gained from applying ISO/IEC 15504 to small software companies in Brazil.* Lisbon,Portugal.

Backstrom, C. H. (1981). *Survey Research.* New York.

Claes, W., & Per, R. (2000). *Experimentation in Software Engineering, An Introduction.* Boston: Springer.

De la villa, M. (2004). *Modelos de evaluación y Mejora de Procesos: Análisis Comparativo.* Universidad de Huelva.

IEEE, I. . (2008). *ISO/IEC 12207 Systems and software engineering — Software life cycle processes.* Software & Systems Engineering Standards Committee.

INTECO. (2008). *Mejora de los Procesos de Desarrollo Software Visión Práctica.* Universidad de Sevilla, España.

Kasunic, M. (2005). *Designing an Effective Survey.* Pittsburgh: Carnegie Mellon Software Engineering Institute.

Laporte, C. Y. (2008). *A Software Engineering Lifecycle Standard for Very Small Enterprises.* Dublin: EuroSPI.

Laporte, C. Y. (s.f.). *Deployment Packages and Eclipse Process Framework Project*. Recuperado el 10 de Febrero de 2011, de http://profs.etsmtl.ca/claporte/English/VSE/index.html

Laporte, C. Y. (2008 - I). *The Application of International Software Engineering Standards in Very Small Enterprises.* Montreal, QC, Canadá: Software Quality Professional Journal.

Laporte, C., April, A., & Renault, A. (2006). *Applying ISO/IEC Software Engineering Standards in Small Settings: Historical Perspectives and Initial Achievements, Proceedings of SPICE Conference.* Luxembourg.

Marc, M., & Robert, M. (2008). *The practical guide to defect prevention.* Washington: Microsoft.

Myers, G. J. (2004). *The Art of Software Testing.* New Jersey.

OECD. (2002). *OECD Small and Medium Enterprise Outlook.* Paris, Francia: Organisation for Economic Co-operation and Development.

Oktaba, H. (2005). *Modelo de Procesos para la Industria de Software, MoProSoft.* México, D.F.: Universidad Nacional Autónoma de México.

Pérez Zorrilla, J. (2005). Evaluación de la comprensión lectora: Dificultades y limitaciones. *Revista de Educación* , 121-138.

Ribaud, V., & Laporte, C. Y. *Experience Management for Very Small Entities: Improving the Copy-paste Model.* International Journal on Advances in Software.

Ribaud, V., Saliou, P., & O´Connor, R. V. (2010). *Software Engineering Support Activities for Very Small Entities.* Grenoble, France.

SCAMPI. (2001). *Standar CMMI Appraisal Method for Process Improvement.* Pittsburgh: Software Engineering Institute.

SPICE. (2003). *Software Process Improvement Capability Determination. ISO/IEC 15504-2 Interational Standar.* Switzerland: International Organization for Standardization.

Vazquez, A. (s.f.). *Public Site of the ISO/IEC JTC1/SC7 Working Group 24 ISO/IEC 29110 - Life Cycle Profiles for Very Small Entities (VSEs).* Recuperado en Noviembre de 2010, de http://profs.etsmtl.ca/claporte/English/VSE/Deploy%20Pack/DP-Construction_rev4.doc

Wilks, S. (1962). *Mathematical Statistics.*

Deployment Package
Construction and Unit Testing
Basic Profile

Notes:

This document is the intellectual propriety of its author's organization. However, information contained in this document is free of use. The distribution of all or parts of this document is authorized for non commercial use as long as the following legal notice is mentioned:

© 5th level

Commercial use of this document is strictly forbidden. This document is distributed in order to enhance exchange of technical and scientific information.

This material is furnished on an "as-is" basis. The author(s) make(s) no warranties of any kind, either expressed or implied, as to any matter including, but not limited to, warranty of fitness for purpose or merchantability, exclusivity, or results obtained from use of the material.

The processes described in this Deployment Package are not intended to preclude or discourage the use of additional processes that Very Small Entities may find useful.

Author	JAVIER FLORES – Universidad Nacional Autónoma de México (UNAM), (México) ANA VAZQUEZ – 5th level (México)
Editors	R. CHAMPAGNE – École de Technologie Supérieure (ETS), (Canada) C. Y. LAPORTE – École de technologie supérieure (ÉTS), (Canada)
Creation date	29/06/11
Last update	25/02/13
Version	0.5

Version History

Date (yyyy-mm-dd)	Version	Description	Author
2011-06-29	0.5	Document Creation	Javier Flores

Abbreviations/Acronyms

Abre./Acro.	Definitions
DP	Deployment Package - a set of artefacts developed to facilitate the implementation of a set of practices, of the selected framework, in a Very Small Entity.
VSE	Very Small Entity – an enterprise, organization, department or project having up to 25 people.
VSEs	Very Small Entities
TL	Technical Leader
AN	Analyst
DES	Designer
PR	Programmer
PM	Project Manager

Table of Contents

1. Technical Description 5

Purpose of this document 5

Why are Construction and Unit Testing Important? 5

2. Definitions 7

Generic Terms 7

Specific Terms 7

3. Relationships with ISO/IEC 29110 8

4. Description of Processes, Activities, Tasks, Steps, Roles and Products . 10

Sub-task: Define construction standards 10

Sub-task: Report defect taxonomy of previous phases 12

Assign tasks to the members of the work team 15

Construct or update software components 19

Design or update unit test cases and apply them 22

Correct the defects 28

Update the Traceability Record 29

Role Description 30

Product Description 31

Artefact Description 33

5. Template 34

5.1 Java construction template 34

6. Example 36

6.1 Example of a general construction standard 36

Formatting 36

6.2 Pseudocode Example 39

6.3 Flow chart example 40

6.4 Hierarchy of strengths from weakest to strongest of coverage criterion 41

6.5 Test case design, unit test writing and use of a code coverage tool. 43

6.6 Unit testing following Structured testing technique and mapping to the steps of the task "Design or update unit tests cases and apply them". 46

6.7 Defect correction 50

6.8 Test runners examples 52

6.9 C++ Example of an assertion macro 53

6.10 Unit tests life cycle 53

7. Checklist 54

Task Checklists 54

7.1 Assign task to the members of the work team 54

7.2 Construct or update software components 54

7.3 Design or update unit test cases and apply them 54

7.4 Correct the defects 54
Support Checklists 55
7.5 Code review checklist........ 55
7.6 What the architect and designer should provide 56
7.7 Sub-task: Select the user interface standard 56
8. Tool 57
8.1 Traceability Matrix 57
8.2 Code coverage tools 58
8.3 Unit testing frameworks 59
9. Reference to Other Standards and Models 60
ISO 9001 Reference Matrix 60
ISO/IEC 12207 Reference Matrix 61
CMMI Reference Matrix 62
10. References 64
11. Evaluation Form 65

1. Technical Description

Purpose of this document

This Deployment Package (DP) supports the Basic Profile as defined in ISO/IEC 29110 Part 5-1-2: Management and Engineering Guide. The Basic Profile is one profile of the Generic profile group. The Generic profile group is composed of 4 profiles: Entry, Basic, Intermediate and Advanced. The Generic profile group is applicable to VSEs that do not develop critical software, commercial off the shelf software products. The Generic profile group does not imply any specific application domain.

A DP is a set of artefacts developed to facilitate the implementation of a set of practices in a Very Small Entity (VSE). A DP is not a process reference model (i.e. it is not prescriptive). The elements of a typical DP are: description of processes, activities, tasks, roles and products, template, checklist, example, reference and reference to standards and models, and tools.

The content of this document is entirely *informative*.

This document has been produced by Javier Flores (UNAM, México) and Ana Vazquez of 5th level (México) beyond her participation to ISO JTC1/SC7/WG24.

Why are Construction and Unit Testing Important?

All the stages in development software are important, but the central activity is software construction and here are some reasons:

- Real projects often skip requirements and design. But no matter how rushed a project is, ***you can't drop construction***. So, Improving construction is thus a way of improving any software-development effort, no matter how abbreviated it is.

- ***In many projects, the only documentation available to programmers is the code itself***. Requirements specifications and design documents can go out of date, but the source code should be always up to date.

- ***80% of development costs*** *are consumed by software programmers identifying and correcting defects*[1].

As soon as the programmer develops a unit of code the next step is evaluates it to see if it works properly. Unit testing is the solution for this job since it is the first level of testing, and as illustrated in Table 1 the latest you found defects the bigger the cost is.

[1] NIST, Department of Commerce's National Institute of Standards and Technology

Time introduced	*Time Detected*				
	Requirements	*Architecture*	*Construction*	*System Test*	*Post-Release*
Requirements	1	3	5-10	10	10-100
Architecture	---	1	10	15	25-100
Construction	---	---	1	10	10-25

Table 1. Proportional average cost of fixing defects based on when they're introduced and when they're detected. [Code Complete]

As shown in chart 1 more than 70% of defects are injected before construction plus another 14% injected in this stage[2], so if nothing is done at this point the cost in later stages will increase significantly.

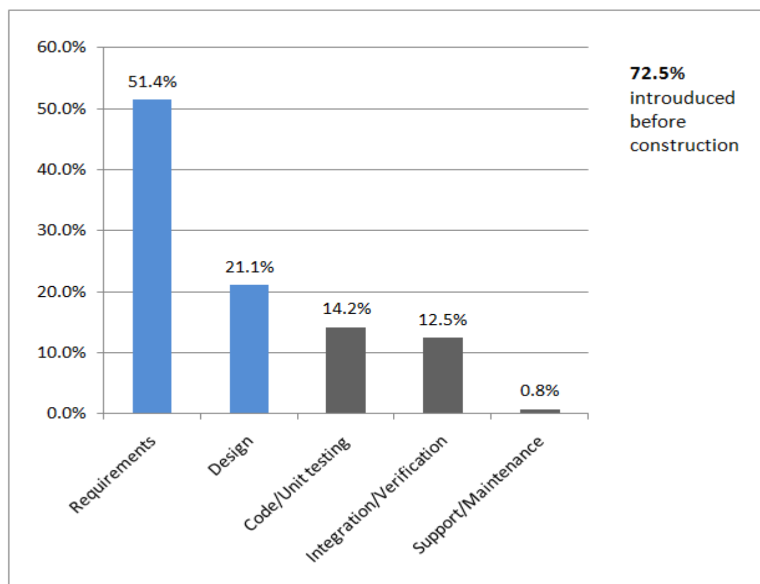


Chart 1. Origins of software defects (Selby 2007). Adequacy

[2] Selby, P., Selby, R.W., Measurement-Driven Systems Engineering Using Six Sigma Techniques to Improve Software Defect Detection, Proceedings of 17th International Symposium, INCOSE, June 2007, San Diego.

2. Definitions

In this section, the reader will find two sets of definitions. The first set defines the terms used in all Deployment Packages, i.e. generic terms. The second set of terms used in this Deployment package, i.e. specific terms.

Generic Terms

Process: set of interrelated or interacting activities which transform inputs into outputs [ISO/IEC 12207].

Activity: a set of cohesive tasks of a process [ISO/IEC 12207].

Task: required, recommended, or permissible action, intended to contribute to the achievement of one or more outcomes of a process [ISO/IEC 12207].

Sub-Task: When a task is complex, it is divided into sub-tasks.

Step: In a deployment package, a task is decomposed in a sequence of steps.

Role: a defined function to be performed by a project team member, such as testing, filing, inspecting, coding. [ISO/IEC 24765]

Product: piece of information or deliverable that can be produced (not mandatory) by one or several tasks. *(e. g. design document, source code)*.

Artefact: information, which is not listed in ISO/IEC 29110 Part 5, but can help a VSE during the execution of a project.

Specific Terms

Component: Set of functional services in the software, which, when implemented, represents a well-defined set of functions and is distinguishable by a unique name [ISO/IEC 29881:2008]

Defect: A problem which, if not corrected, could cause an application to either fail or to produce incorrect results [ISO/IEC 20926].

Traceability: Degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another [IEEE 1233-1998]

Unit test: Testing of individual routines and modules by the developer or an independent tester [ISO/IEC 24765]

Code coverage: Measure used in software testing. It describes the degree to which the source code of a program has been tested. [Practical Software Testing]

3. Relationships with ISO/IEC 29110

This deployment package covers the activities related to Construction and Unit Test of the ISO Technical Report ISO/IEC 29110 Part 5-1-2 for Very Small Entities (VSEs) – Basic Profile [ISO/IEC29110].

The construction activities should have been planned during the Project planning activity of the project. The construction activities should be described in the project plan. If this is not the case, the project manager should perform this activity before beginning construction. (see the Project Management Deployment Package)

In this section, the reader will find a list of Software Implementation (SI) process, activities, tasks and roles from Part 5 that are directly related to this topic. This topic is described in details in the next section.

- **Process:** SI - Software Implementation
  - **Activity:** SI.2[3] Software Requirement Analysis
    - **Tasks and Roles:**

Tasks	Roles[4]
SI.2.2 Document or update the *Requirements Specification.* Identify and consult information sources (customer, users, previous systems, documents, etc.) in order to get new requirements. Analyze the identified requirements to determinate the scope and feasibility. Generate or update the *Requirements Specification.*	AN, CUS

- **Process:** SI - Software Implementation
  - **Activity:** SI.4 Software Construction
    - **Tasks and Roles:**

Tasks	Roles
SI.4.1 Assign tasks to the Work Team members related to their role, according to the current *Project Plan*	TL, PR

[3] These numbers refer to processes, activities, tasks of ISO/IEC 29110 Part 5-1-2

[4] Roles are defined in a next section. Roles are also defined in ISO/IEC 29110 Part 5-1-2

SI.4.3 Construct or update *Software Components* based on the detailed part of the *Software Design*.	PR
SI.4.4 Design or update unit test cases and apply them to verify that the *Software Components* implements the detailed part of the *Software Design.*	PR
SI.4.5 Correct the defects found until successful unit test (reaching exit criteria) is achieved.	PR
SI.4.6 Update the Traceability Record incorporating Software Components constructed or modified.	PR

4. Description of Processes, Activities, Tasks, Steps, Roles and Products

- **Process:** SI - Software Implementation
  - **Activity:** SI.2 Software Requirements Analysis
    - **Tasks and roles**

Tasks	Roles[5]
SI.2.2 Document or update the *Requirements Specification.* Identify and consult information sources (customer, users, previous systems, documents, etc.) in order to get new requirements. Analyze the identified requirements to determinate the scope and feasibility. Generate or update the *Requirements Specification.*	AN , CUS

This task is related with the following sub-tasks:

- Define construction standards
- Report defect taxonomy of previous phases

Sub-task: Define construction standards

Objectives:	Provide guidance on the software coding to produce software easy to maintain inside or outside of the project.
Rationale:	Maintainability requirements are directly addressed by this sub-task. These requirements should have been agreed as part of the Requirements Specification, however most of the times they are not explicitly stated, but they are always expected. The lack of coding standards will be perceived by colleagues when trying to modify code, inside the team and out of it. Some Components could be replaced by new ones due the inability of understand them, if maintenance is performed by the development team, the cost of the project will increase, if it is performed by the client then they will absorb this cost. Coding Standards should be used in at least in those components

[5] Roles are defined in a next section. Roles are also defined in ISO/IEC 29110-5.1

	that perform key functionality. Coding Standards are not directly address by 29110-5, however investing some effort can help increment the productivity of the project and the quality of the product. *Note: A general construction standard is provided in the example section*
Roles:	Project Manager
	Technical Leader
	Programmer
Artefacts:	Construction Standards
Steps:	1. Plan the sub-task.
	2. Obtain available standards.
	3. Select the standards.
	4. Adopt the standards.
	5. Verify the adoption of the standards.
Step Description:	***Step 1. Plan the sub-task*** According to the project progress, the Project Manager includes in the plan these steps. Assigning effort is very important because defining standards may be endless, and if standards are not adopted, then also will be useless. Regarding the schedule, it is desirable to have the standards ready before start the construction however it is not always possible. ***Step 2. Obtain available standards.*** Find out if your customer has construction standards, if not look for them in the internet or any other available source. Avoid defining them from the scratch, in most of the projects it is outside of the scope, creating them can take a lot of effort and time. ***Step 3. Select the standards.*** If your costumer has not standards then ask the programmers to select one of those that were found or a combination of them. Regarding the documentation of the standard, the easiest way is to implement some components and use them as examples, if you have enough time, then create the construction standards.

	Note 1: An example of a general coding standard is provided in the example section. *Note 2: A Java construction template is provided in the template section.* ***Step 4. Adopt the standards.*** Ask the programmers to adopt the standards from now on, especially in those components that perform key functionality. ***Step 5. Verify the adoption of the standards.*** Make the components that perform key functionality to be verified regarding the adoption of construction standards by another programmer.

Sub-task: Report defect taxonomy of previous phases

Objectives:	Report defects of previous stages to the related area in charge.
Rationale:	As shown in chart 1 more than 70% of defects are injected before construction. So there is a big likelihood of finding defects of previous stages in the construction phase. So the best strategy is report defects to the area in charge before their corrections turns more expensive. This task is not directly address by 29110-5, however investing some effort can help increment the productivity of the project and the quality of the product.
Roles:	Technical Leader
	Programmer
Artefacts:	Defect Taxonomy [Updated]
Steps: ***(Programmer)***	1. Confirm the defect
	2. Write a brief description of the defect
	3. Write the defect area
	4. Write the possible cause
	5. Write the damage extend
Steps: ***(Technical Leader)***	1. Verify the defect report
	2. Write a mitigation strategy
	3. Report the defect to the area in charge
Step Description: ***(Programmer)***	***Step 1.* Confirm the defect** If you found inconsistencies in detail design, software architecture or a part of the requirements, explain the details to your technical leader.

<table>
<tr><td></td><td>
If your technical leader agrees with you that what you found is a defect. Continue with the next steps, otherwise skip the all sub-task. But remember that not reporting a defect at the right time means that after you code it, sometime you will have to change it anyway.

Step 2. Write a brief description of the defect

Write a brief description of the defect including how you found the defect and the conditions and circumstances where you found it.

Step 3. Write where the defect was injected

Write where the defect was discovered in the life cycle
• Requirements
• Software architecture
• Detail Design

Step 4. Write the possible cause

Writing the purpose of cause information is providing indications of the root cause of a defect:

• Systematic factor: guidelines and procedures; company culture; domain-specific information, such as documentation, -code, tools, etc.
• Human factors: an individual made a mistake for human reasons: omission, misapplication, didn´t look, didn´t find, incorrect solution etc.
• Unknown

Step 5. Write the damage extend

Write how widespread is its effect

• Function
• Object
• Process
• Compatibility
• Application
• Machine
• Server
• Clients
• Net
• Other
</td></tr>
<tr><td>Step Description:

(Technical Leader)</td><td>
Step 1. Verify the defect report

Verify the report made by the programmer. If there is some inconsistency ask the programmer to clarify the details.
</td></tr>
</table>

	Step 2. **Write a mitigation strategy** If you know a mitigation strategy for the defect found add it to the report. Some of the mitigations could be: **Mitigation type:** • New tool • Hardware • Training • Staff management • Communication (groups/individuals) • Knowledge access • Process change ***Step 3.*** **Report the defect to the area in charge** Once the report is complete send it to the area in charge and if possible, reassign the tasks for the programmer who found the defect while the related area responds the request.

- **Process:** SI - Software Implementation
  - **Activity:** SI.4 Software Construction
    - **Tasks and Roles:**

Tasks	Roles[6]
SI.4.1 Assign tasks to the Work Team members related to their role, according to the current *Project Plan*	TL, PR
SI.4.3 Construct or update *Software Components* based on the detailed part of the *Software Design*.	PR
SI.4.4 Design or update unit test cases and apply them to verify that the Software Components implements the detailed part of the *Software Design.*	PR
SI.4.5 Correct the defects found until successful unit test (reaching exit criteria) is achieved.	PR
SI.4.6 Update the Traceability Record incorporating Software Components constructed or modified.	PR

[6] Roles are defined in a next section. Roles are also defined in ISO/IEC 29110-5.1

Assign tasks to the members of the work team

Note: The tasks are related to their role, according to the Project Plan.

Objective:	Define the construction sequence and assign tasks to the members of the Work Team
Rationale:	Most of the times the first version of the Project Plan has identified the most of the components to be constructed, however at this stage of the project, when the Software Architectural and Detailed Design has been completed, the Project Plan should be updated to include construction and unit tests, in detail or in general manner, of all the components that have to be produced. The sequence of construction of the components should be coordinated with the integration sequence to have the components (unit tested) ready to be integrated at the right time. Defining the construction sequence is not included in 29110 part 5, however investing some effort to define the construction sequence can help optimize the construction calendar.
Roles:	Technical Leader
Products:	Project Plan
Steps:	1. Obtain software design documentation.
	2. Select the integration sequence strategy.
	3. Detail the project schedule.
	4. Allocate tasks to the members of the work team.
	5. Define exit criteria for unit testing
Step Description:	***Step 1. Obtain software design documentation*** • Obtain the software design documentation from the project repository; Detailed Low Level Software Design includes details of the software components to be constructed. • Obtain the Traceability Record from repository ***Step 2. Select the integration sequence strategy*** There are several strategies to determine the integration sequence, but they are more heuristics than algorithms. The most widespread integration approaches are: Big bang: integrate all parts at once, in which the entire software is assembled and tested in one step (this approach may be risky because of the level of entropy injected). - Bottom-up: Modules at the lowest levels are integrated at first, then by moving upward through the control structure. The process is

repeated until the component at the top of the hierarchy is integrated.

- Top-down: Modules are integrated by moving downward through the control structure. The process is repeated until the components at the bottom of the hierarchy are integrated

Note: Implementing pure Bottom-up or Top-down integration is sometimes radical, so a better choice is a hybrid approach instead

Some hybrids approaches are:

Risk oriented integration

Risk-oriented integration tends to integrate the components at the top (high-level business-object classes) and the bottom (device-interface and utility classes) first, leaving the middle-level components for last. You prioritize according to the level of risk associated with each component.

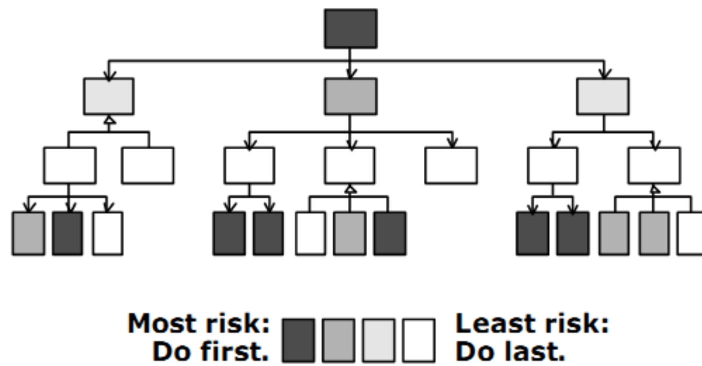


Figure 2 Risk-oriented integration strategy. [Code complete]

Feature-oriented integration

The idea is integrate components in groups that make up identifiable features. Each feature integrated brings an incremental addition in functionality. An advantage of this strategy is that it fits easily to with object oriented design.

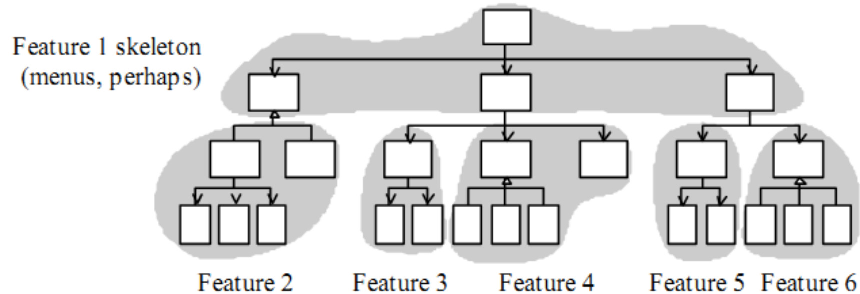


Figure 3 Feature oriented integration strategy. [Code complete]

At the end you decide whether using a single integration approach or a combination of two or more and tailoring to your specific project.

Step 3. Detail the project schedule

Detail the project schedule including the activities to develop or modify all the identified components according with the selected strategy.

Do your best to keep time and cost project's agreements, if it is necessary to modify them, then a change control request should be raised. For further details of project schedule go to the Project Management Deployment Package.

Step 4. Allocate tasks to the members of the work team

- Allocate tasks to members of work team
    - Allocate tasks to code software components
    - Allocate tasks to develop test cases and test software components
- Inform members of their tasks

To allocate tasks to the right people there are some aspects of programmers that you should consider. Always ask for the background of your team:

- Experience
- Skills
- Knowledge

This will help you to have the right person in the right activity. Also have in mind these considerations:

- Always try to assign the most critical components and tasks to the most experienced people.
- Leave the basic components to novice programmers; suddenly they will become expert programmers.
- Try to identify the skills of each programmer and take advantage of that.

5. Define exit criteria for unit testing

Testing all possible cases of a component is not feasible even for small projects. You should therefore decide an exit criterion which tells your programmers when they should stop testing or when there is enough testing for a kind of component.

A good start is defining a percentage of some code coverage criteria (see step 1 from task "Design or update unit test cases and apply them") depending on how risky the code under test is. It could be divided into three categories:

Type of component	Description
High risk	Code could cause severe damage (wipe out data, injure someone, give non-obviously wrong answers that might cost a user a lot of money), or has many users (so the cost of even minor bugs is multiplied), or seems likely to have many mistakes whose costs will add up (it was a tricky algorithm, or it talks to an ill-defined and poorly-understood interface, or you have already found an unusual number of problems).
Low risk	Code is unlikely to have bugs important enough to stop or delay a shipment, even when all the bugs are summed together. They would be annoyance bugs in inessential features, ones with simple and obvious workarounds.
Medium risk	Code is somewhere in between. Bugs here would not be individually critical, but having too many of them would cause a schedule slip. There's good reason to find and fix them as soon - and as cheaply - as possible. But there are diminishing returns here - time spent doing a more thorough job might better be spent on other tasks

Table 2 Categories of code under test. **[Misuse Code coverage]**

There is no algorithm that can tell you which category corresponds to a component. In reality, some medium risk code might be high risk with respect to certain types of failures and the same might happen with the low risk code You should evaluate the impact of each component and select properly.

Once you have defined the risk of the component it is suitable to use stronger criteria for testing higher risk components.

Type of component	Coverage criteria recommended
Low risk	Statement
Medium Risk	Branch/Decision
High Risk	Decision and condition

	Table 3 Example of coverage criteria for type of component A common amount selected for some respectable companies is 85% of code coverage (Marick 1999). However this is not always the right amount since each project has different constraints. The chosen coverage percentage depends on several circumstances such as the importance of the project, the schedule and the available resources. Achieving 100% code coverage is generally very difficult and sometimes is not cost effective. However achieving a high percentage can give you certain confidence that your code works properly even if is not entirely tested.

Construct or update software components

Note: Components are based on the detailed part of the software design

Objective:	Produce the Software Components as designed.
Rationale:	The quality of the construction substantially affects the quality of the software and the best way to do it is by improving developer's practices and techniques. Programmers may feel confidence enough to produce components without a systematic approach; however it still being useful for constructing complex or critical components. There are several approaches to produce components, here we use the Pseudocode and Flow chart approaches which are two of the most widespread accepted. Steps were adapted from Code Complete (see reference). *Note: Sometimes there are inconsistencies from previous stages (Requirements, Detail design). If this is the case the best option is report them before constructing something inadequate. (See subtask "Report defect taxonomy of previous phases").*
Roles:	Programmer
Products:	Software Components
Steps:	1. Understand the detail design of the component and its contribution
	2. Search for available functionalities in the standard libraries.
	3. Define the logic of the component
	4. Code the component according to the construction standard
	5. Verify the component

Step Description:	***Step 1.*** **Understand the detail design of the component and its contribution** **a) Verify the contribution** Verify the contribution of the component and see if the best way to solve the problem is with the creation of a new component or the adaptation of a previous one. **b) Detailed Enough** If Software detail design is not the clear enough the programmer should complete it with the support of the Designer. Detailed Software Design should includes details of the Components to facilitate their construction and testing within the programming environment: - Provides detailed design (classes diagram, activity diagram, entity relationship diagram, etc.) - Provides format of input / output data - Provides specification of data storage needs - Defines the format of required data structures - Defines the data fields and purpose of each required data element *Note: If you found any inconsistence in detail design tell your technical leader. See sub-task "Report defect taxonomy of previous phases"* ***Step 2. Research functionality available in the standard libraries*** Find out whether some or all of the component's functionality might already be available in the library code of the language, platform, or tools you're using. Improve your productivity by reusing code. Many algorithms have already been created, tested, and improved, so don't invent the wheel, just use it. ***Step 3. Define the logic of the component.*** There are several approaches to produce components, here we use the Pseudocode and Work flow approaches which are two of the most widespread accepted. Steps were adapted from Code Complete (see reference). **a) Pseudocode programming process** One of the best ways to abstract and idea is making a simple draw of it at a very high level. Then refine the idea and start writing some lines of Pseudocode that describe a possible solution at medium level (In your natural language), then continue refining the Pseudocode until you reach a low level design (Very close to a source code). So that turn the Pseudocode into source code can be

	easy. Always try as many ideas as you can in Pseudocode before you start coding. Once you start coding, you get emotionally involved with your code and it becomes harder to throw away a bad design and start over. *Note: An example of a Pseudocode is provided in the example section.* **b) Flow chart** Another way to abstract an idea is making a graph or symbolic representation of the algorithm you have in mind. Each step in the algorithm is represented by a symbol and contains a short description of the process step. The flow chart symbols are linked together with arrows showing the algorithm flow direction. You can start by defining an initial state and then adding steps, branches or loops to the process until you reach the end of the algorithm. Then you can refine each step as much as needed until turn it into source code can be easy. *Note: An example of a flow chart is in the example section.* ***Step 4.Code the component according to the construction standard*** Coding should be a mechanic process once the algorithm is ready. Focusing on configuration and language issues more than in the logic of the component. Some of the previous work can be reusable, for instance if you are using Pseudocode approach you can use the *high level* Pseudocode as comments and use the *low level* Pseudocode for coding the logic of the component. For this kind of issues is important to follow the construction standard when coding, because not only makes your code clean but it becomes standardized and easy to maintain. *See the subtask "Define construction standards".* If your component needs an interface you should code it according to an Interface standard. If you don't have one asks for it to your technical leader. *See "Select the user interface standard" in the checklist section.* **Check whether code should be further factored** You can consider create a subroutine if: • Some of the steps in the process are very repetitive and instead of copy-paste lots of code you just call the properly subroutine that makes the job.

	• Some steps in the algorithm are part of an independent sub-process that could be useful for another component. Creating a subroutine for it does not change the logic but make the code clear and more maintainable. ***Step 5. Verify the component*** Code review is a systematic examination in which software developers review their code, identifying bugs and keeping code more maintainable. It is intended to find and fix defects overlooked in the initial development phase, improving both quality of source code and the developers' skills. *Note: An example of a checklist for code review is provided in the checklist section.* **Compile the code**: Let the computer check for undeclared variables, naming conflicts, and so on.

Design or update unit test cases and apply them

Objective:	Find and correct defects injected when coding through the design and application of unit test cases
Rationale:	There are two fundamental approaches for testing, namely black box and white box. Black box testing tests the functional requirements, without knowing the component internal structure. White box testing tests the internal structure and the logic of the component. These components or units are the smallest building blocks of software and since it is easier and less costly to look for a defect in a small unit than in a larger part of the system, they should be performed as soon as units are ready. Even if is not always possible it is highly recommended that programmers perform white box testing since they know the components they code. To be practical in this package that last suggestion is assumed and the approach selected is white box testing. A major benefit of unit testing is that it lets you achieve parallelism, by enabling testing and debugging simultaneously by many programmers. With the help of tools and frameworks, testing efficiency can be improved. *Note: Unit testing is useful for all types of software. However, software implemented using an object-oriented language*

	introduces additional considerations that must be taken into account when unit testing such software. The interested reader can consult [Introduction to Software Testing] for a discussion on the issues associated with testing object-oriented implementations.
Roles:	Programmer
	Project Manager
Products:	Software Components [unit tested]
Artefacts:	Test Cases, Set of unit tests
Steps:	1. Obtain the exit criteria
	2. Design the test cases
	3. Code the unit tests
	4. Execute the unit tests
	5. Analyze the results
Step Description	***Step 1. Obtain the exit criteria*** The exit criteria are composed of one or more code coverage[7] criterion and a percentage to be achieved. It should have been previously defined by the Project Management. **Coverage criteria** Coverage criteria (adequacy criteria for some authors) are a white box framework that works as a stopping rule to determine whether or not sufficient test cases have been created. Common coverage criteria include statement, branch/decision and decision and condition coverage. Statement coverage is considered the weakest of these criteria, in the sense that it is the one requiring the less testing effort to reach, but it is also the one that is less efficient to reveal faults. Statement — Branch/Decision — Decision and Condition *Less* **Efficiency/Effort** *More* **a) Statement coverage** Satisfying this criterion ensures the execution of each line of code regardless of the path[8]. It is the weakest test coverage criteria because some defects are hidden in a particular path, which is not necessarily traversed when the only goal is to execute every line of code.

[7] Measure used in software testing. It describes the degree to which the source code of a program has been tested

[8] Path: A sequence of statement execution that begins at an entry and ends at an exit. (Lee Copeland, 2004)

	Note: This is the minimum level of testing that should be covered. **b) Branch/Decision coverage** This criterion requires that each possible outcome of each decision as a whole be exercised at least once (not for each individual condition contained in a compound predicate e.g. *if(a&b), a&b* as a whole instead of satisfying condition *a* and *b* separately). *Note: This could be a good goal to cover for components of medium complexity.* **c) Decision and Condition coverage:** Satisfying this criterion not only covers branch/decision coverage but ensures that every condition takes each possible outcome at least once and the decision as a whole takes each possible outcomes at least once (for each individual condition contained in a compound predicate e.g. *if(a&b),* satisfying condition *a* and *b* separated). Note: *It is recommended for complex and critical units.* There are stronger criteria like: Multiple conditions, MDC, etc. But they are beyond the scope of this package. **Percentage for the coverage criteria selected** Depending on the difficulty or importance of the component, the Project Management should choose a suitable coverage criterion. i.e. a component of medium complexity: Branch/decision coverage criterion of 85%. This means that once test cases are designed and that when they are collectively executed, at least 85% of the branches are covered, testing is "complete" for that goal. ***Step 2. Design the test cases*** A test case is a specific set of inputs values with which the programmer will determine whether the component is working properly, based on whether the outputs obtained are the expected ones or not. Each test case should have: ***Test case ID:*** To have a control of the number of test cases created. ***Description***: Describes the purpose of the test. The description might have additional information like the true or false outcome of certain decisions or conditions or perhaps the path covered if you

are using a control flow graph (CFG). It depends on the strategy you are using for obtaining the test cases.

Inputs: the input values configuration that forms the test case.

Expected outputs: The expected results for the input configuration.

Note 1: An extra field to indicate if the test case passes or fails is needed, but remains empty until the execution step, where this information is obtained.

Pass/Fail: Use P or F to indicate if the test case passes or fails.

Structured Testing Technique

Structured Testing (also known as basis path testing) is a technique that optimizes the number of tests cases needed to reach 100% branch/decision coverage.

Note: A complete example of Structure Testing is provided in the example section.

Step 3. Code the Unit tests

A unit test is a test script (typically a function or method, depending on the nature of the implementation) which tests one or more test cases of a specific component.

To code the unit tests, the best option is to use a unit testing framework for the language you are using in the implementation. If there is no existing framework for your language, another option is to build your own testing framework.

***Without* an external unit testing framework**

If there is no unit testing framework for the language you are using, you can write your own routines for testing. To do this, you need code that checks whether a call to the component under certain conditions returns true (if everything is working as expected) or false (if the call results in an unexpected output or error).

These kinds of routines are called assertions and they usually take two arguments: a Boolean expression that describes the assumption that's expected to be true, and a message to display if it isn't. The idea is that your test scripts use these assertions to verify if the test case can detect a defect. If your language doesn't directly support assertion routines, they are relatively easy to write.

Note: An example of an assertion macro is provided in the example

section.

With an external unit testing framework

These tools are language-dependent and let you write and run unit tests to see if they enable finding defects in the unit under test. Almost all of them have their own suite of assertions that help performing the tests.

The general idea is to write scripts that contain one or more assertions which receive an expected output and the unit under test with fixed inputs, so if some of the assertions fail, you know which one and you can compare the expected output against the output obtained.

The goal of this kind of tool is that once you have written a set of unit tests, if there are changes in the implementation, by running the unit tests again, you know that you have not "broken" functionality that was previously working.

An important advantage of these tools is that there is no need to remove anything, you keep your implementation and test sources separated from the start.

Note: In the tool section, links to the most popular unit testing frameworks for several languages are provided.

Step 4. Execute the unit tests

Note: Isolating the unit under test as much as possible is important to avoid inadequate behavior due to dependencies among units.

Without **an external unit testing framework**

Once you have coded your unit tests, you can create a mechanism to run them. It could be a separate program whose only purpose is running the unit tests, or a function or method inside the implementation code with the same objective.

With **an external unit testing framework**

Unit testing frameworks usually provides a Test Runner that basically is some form of command-line or graphical application that can be used to run the automated tests and report on the result.

a) Command-line Test Runner

Command-Line Test Runners are designed to be used from an OS command line or from batch files or shell scripts. They are very useful when working remotely via remote shells or when running

	the tests from a build script such as "make" or "ant". If the test suite is too large you can let it run overnight and see the results in a log file the next day. *Note: An example is provided in the example section.* **b) Graphical Test Runner** A Graphical Test Runner is typically a desktop application or part of an IDE[9] (either built-in or a plug-in) for running tests that makes their usage easier and more convenient. The most common feature of these applications is some sort of real-time progress indicator. Some of them also include a running count of test failures and a colored progress bar which starts off green and turns red as soon as an error or failure is encountered. *Note: An example is provided in the example section.* ***Step 5. Analyze the results*** Depending on how you execute the unit tests, you can either analyze the results stored in the log file or in the output window if you run them via a command line or simply look at the results graphically if you are using a GUI implementation. In any case the next step is to determine if any test fails. If this is the case, you should make the relevant corrections either in the implementation code or in the test code itself, and run the tests again. Once all your unit tests pass, you should look for the exit criteria defined by the Project Management and see if you need to create more test cases or if you have enough. It is highly recommended to use a *Code coverage* tool to see how much code your test cases cover, and what percentage you achieved in the selected criteria. Some popular code coverage tools are listed in the tool section and all of them consider at least statement and branch/decision coverage criteria.

[9] IDE - from Integrated Development Environment. Software application that provides facilities to programmers for software development, like: source code editor, compiler or interpreter, build automation, debuggers, etc.

Correct the defects

Note: Defects are corrected until successful unit test (e.g. reaching exit criteria) is achieved.

Objective:	Correct the defects found by the Unit Tests
Rationale:	Correct the defects just found by the Unit Test by the person in charge of the Component construction, is the fastest and cheapest way of correcting the defect.
Roles:	Programmer
Products:	Software Components [corrected]
Steps:	1. Confirm the defect
	2. Determine the nature and location of the defect
	3. Correct the defect
	4. Verify the corrections

Step Description:	***Step 1. Confirm the defect***
	
	 Verify that what you found is a defect in the implementation code (the purpose of the test cases) and not a defect in the test code itself.
	
	 Step 2. Determine the location of the defect
	
	 Error locating practices:
	 • ***Use code review checklist:*** If you don´t find the error you can support on a checklist for code review (An example is provided in the example section). It could give you an idea of where to look.
	 • ***Step through the code in the debugger:***
	 Once the routine compiles, put it into the debugger and step through each line of code. Make sure each line executes as you expect it to.
	 • ***Find the source of an error.*** Trying to reproduce it by several different ways to determine its exact cause. Some advices could be:
	     • Narrow the suspicious region of the code
	     • Use different data
	     • Refine the test cases

	Program / First test / Defect / Program / Defect / Second test / Program / Defect / Third test / Program / Defect / Subsequent tests **Figure 4. Reproduce an error several ways to determine its exact cause. [Code complete]** • ***Use experimentation as a last resource:*** The most common mistake some programmers make is trying to solve a problem by making experimental changes to the program. But usually it injects more defects. ***Step 3. Correct the defect.*** Once you have found the source of the defect, save the original source code and then make the corresponding changes. *Note: Another common failing is repairing the symptoms of the error, or just one instance of the error, rather than the error itself. So be aware of this.* ***Step 4. Verify the corrections*** Run the unit tests again to verify that the corrections you made works properly. Repeat from step 1 each a unit test finds a defect. If the unit tests do not find defects anymore you should look for achieving the exit criterion selected for your component.

Update the Traceability Record

Note: Incorporating *Software Components* constructed or modified.

Objective:	Ensure that all software components can be traced to a design element and that all design elements have been constructed.

Rationale:	The traceability record should have been developed in the previous phases of the project to ensure traceability from requirements to design elements. This task is devoted to ensure traceability between design and construction elements.
Roles:	Programmer
Products:	Traceability Record [updated]
Steps:	1. Update the Traceability Record
	Step 1. Update the Traceability Record This record should be updated with the following information: - Identification of software components - Identification of test cases (optional) - Verification date (i.e. date the software component had been tested and no defect is left)

Role Description

This is an alphabetical list of the roles, abbreviations and list of competencies as defined in ISO 29110 Part 5-1-2.

	Role	*Abbreviation*	*Competency*
1.	Analyst	AN	Knowledge and experience eliciting, specifying and analyzing the requirements. Knowledge in designing user interfaces and ergonomic criteria. Knowledge of the revision techniques. Knowledge of the editing techniques. Experience on the software development and maintenance.
2.	Customer	CUS	Knowledge of the Customer processes and ability to explain the Customer requirements. The Customer (representative) must have the authority to approve the requirements and their changes. The Customer includes user representatives in order to ensure that the operational environment is addressed. Knowledge and experience in the application domain.
3.	Programmer	PR	Knowledge and/or experience in programming, integration and unit tests. Knowledge of the revision techniques. Knowledge of the editing techniques. Experience on the software development and

			maintenance.
4.	Technical Leader	TL	Knowledge and experience in the software process domain.
5.	Project Manager	PM	Leadership capability with experience making decisions, planning, personnel management, delegation and supervision, finances and software development.

Product Description

This is an alphabetical list of the input, output and internal process products, its descriptions, possible states and the source of the product.

	Name	Description	Source

1.	Project Plan	Presents how the project processes and activities will be executed to assure the project's successful completion, and the quality of the deliverable products. It Includes the following elements which may have the characteristics as follows:
		- *Product Description*
		  o Purpose
		  o General Customer requirements
		- *Scope* description of what is included and what is not
		- *Objectives* of the project
		- *Deliverables* - list of products to be delivered to Customer
		- *Tasks, including* verification, validation and reviews with Customer and Work Team, to assure the quality of work products. Tasks may be represented as a Work Breakdown Structure (WBS).
		- *Relationship and Dependence of the Tasks*
		- *Estimated Duration* of tasks
		- *Resources* (humans, materials, equipment and tools) including the required training, and the schedule when the resources are needed.
		- *Composition of Work Team*
		- *Schedule of the Project Tasks,* the expected start and completion date, for each task.
		- *Estimated Effort and Cost*
		- *Identification of Project Risks*
		- *Version Control Strategy*
		  - Product repository tools or mechanism identified
		  - Location and access mechanisms for the repository specified
		  - Version identification and control defined
		  - Backup and recovery mechanisms defined
		  - Storage, handling and delivery (including archival and retrieval) mechanisms specified
			Project Management

		- *Delivery Instructions*
		  - Elements required for product release identified (i.e., hardware, software, documentation etc.)
		  - Delivery requirements
		  - Sequential ordering of tasks to be performed
		  - Applicable releases identified
		  - Identifies all delivered software components with version information
		  - Identifies any necessary backup and recovery procedures
		
		The applicable statuses are: verified, validated, changed and reviewed.	

2.	*Software Component*	A set of related code units. The applicable statuses are: unit tested, corrected and baselined.	Software Implementation

3.	Software Design	This document includes textual and graphical information on the software structure. This structure may includes the following parts:
		Architectural High Level Software Design – Describes the overall *Software* structure:
		- Identifies the required software *Components*
		- Identifies the relationship between software *Components*
		- Consideration is given to any required:
		  - software performance characteristics
		  - hardware, software and human interfaces
		  - security characteristics
		  - database design requirements
		  - error handling and recovery attributes
		
		Detailed Low Level Software Design – includes details of the Software Components to facilitate its construction and test within the programming environment;
		- Provides detailed design (could be represented as a prototype, flow chart, entity relationship diagram, pseudo code, etc.)
		- Provides format of input / output data
		- Provides specification of data storage needs
		- Establishes required data naming conventions
		- Defines the format of required data structures
		- Defines the data fields and purpose of each required data element
		- Provides the specifications of the program structure
		
		The applicable statuses are: verified and baselined.
			Software Implementation

4.	Traceability Record	Documents the relationship among the requirements included in the *Requirements Specification, Software Design* elements*, Software Components, Test Cases and Test Procedures. It may include:* - Identifies requirements of *Requirements Specification* to be traced - Provides forward and backwards mapping of requirements to *Software Design* elements, *Software Components*, *Test Cases and Test Procedures*. The applicable statuses are: verified, baselined and updated	Software Implementation

Artefact Description

This is an alphabetical list of the artefacts that could be produced to facilitate the documentation of a project. The artefacts are not required by Part 5, they are optional.

	Name	Description
1.	Construction Standards	Provides conventions, rules, idioms, and styles for: - Source code organization (into statements, routines, classes, packages, or other structures) - Code documentation - Modules and files - Variables and constants - Expressions - Control structures - Functions - Handling of error conditions—both planned errors and exceptions (input of bad data, for example) among others.
2.	Test Cases	A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. [IEEE 1012-2004]
3.	Set of unit tests	Set of routines written in a specific language designed for testing the test cases.
4.	Defect taxonomy	A method to reduce the number of product defects by learning about the kinds of errors being made in the product development process so that they can be analyzed to improve the process to reduce or eliminate the likelihood of the same kind of error being made in the future. The applicable statuses are: Verified, Updated

5. Template

5.1 Java construction template

```
/*****************************************************************************
* Copyright (c) 20XX [Organization Name], Inc. All rights reserved.
******************************************************************************
    Version Number - $Revision$
    Last Updated   - $Date$
    Updated By     - $Author$

    Summary of module purpose

    Low-level design, physical design discussions, build dependencies,
    assumptions, implementation issues, notes, etc.

/*****************************************************************************
    Package and import statements
*/

package com.mycompany.mypackage;
import com.somepackage;

/*****************************************************************************
    Use these dividers to split the module into logical groups.
*/
/*****************************************************************************
    Class Definitions
*/
/*============================================================================
    Summary of class goal
 ............................................................................
    Description of class
============================================================================*/
public class JavaClass extends SuperClass
    implements SomeInterface
    {
    /*--------------------------------------------------------------------
        Constructors
     ..................................................................
       Description of constructors
    --------------------------------------------------------------------*/
    // Constructor 1 summary
    public JavaClass( )
        {
```

```
        }
    /*==================================================================
        Public Member Functions
    */
    /*------------------------------------------------------------------
        Summary of member function
     ...................................................................
        Description of function
        Describe function's exceptions and return values
    ------------------------------------------------------------------*/
    public RETURN_TYPE
    SomeMethod(
        PARAM1_TYPE param1,     // Description of param1
        PARAM2_TYPE param2      // Description of param2
        )
        throws SomeException
        {

        }
    /*==================================================================
        Private Member Functions
    */
    /*==================================================================
        Protected Data Members
    */
    protected boolean aBooleanValue;

    /*==================================================================
        Private Data Members
    */
    private int anIntValue;
    }
/***************************************************************************/
```

6. Example

6.1 Example of a general construction standard

Formatting

1	All source files should be based on the appropriate construction template.
2	If **no discernable style** is present in existing code or style differs from construction standard, **reformat the file** according to it.

3	Avoid create lines greater than 79 characters in length. But if you need them, indent the next line one level. If the new line also exceeds 79 characters, insert another line break after the last operator within the 79-character limit, but do not indent the next line further. Continue until all lines of the statement are less than 79 characters in length.
	`If (PreliminarySize = 0 And CalculatedMeanSizeInLoc(Module) <> 0) Or`
	`    (PreliminarySize = CalculatedMeanSizeInLoc(Module)) Then`
	`   Call CalibrationData.GetProjectPhaseFromId(`
	`       ProjectCharacteristics.CurrentProjectPhase, ProjectPhase)`
	`End If`

4	Use **blank lines** liberally to separate and organize code, 1 inside function or comment, 2 between functions, 3 or 4 for sections.

5	**Place spaces** to visually separate the variables or functions.
	`If ( 3 == x)`
	`func1 ( param1, param2 );`
	`int array [ MAX_NUM_THINGS ];`

Comments

1	Use comments to describe the intentions of the programmer describing **'why' rather** than **'how' or 'what'**.
2	Only for **critical pieces of code** you should explain in detail the action or how it works. But in that case add an **special mark**.
3	The best way to comment is doing in **natural language** (English, Spanish), not more code or pseudo-code.
4	Others developers should understand the comment (information **useful**).
5	Try to not use redundant or unnecessary comments.
6	Comment each **major block of code** with a brief description of its intention.
7	Indent comments at the **same indentation level** as the code they're commenting, because they are not more neither less important than code itself.
8	Where possible, use a **machine-parsable comment** format such as **JavaDoc** to

	allow examination of code-level documentation separately from the code itself.
9	**Range of numeric** data should be commented.
10	Limitations on input or output data should be commented.

Modules and files

1	When **naming** a file **capitalize** the first letter and the **first letter** of any words or **acronyms** within the filename; use lowercase for all other alphabetic characters (use only **alphanumeric** characters).
	`FileName.java`
	`FileName2.cpp`

2	Files with similar **purposes** should be **named** similarly.
3	**Begin** all source files with the standard comment **header** describing the **purpose** of the file and giving a **brief synopsis** of its contents.
4	Placing more than one module or component into a single source file **is not recommended.**
5	Try to **use relative path** to prevent include the file in another environment.

Variables

1	Use each variable for exactly one purpose.

2	**Capitalize** the **first character** of words and acronyms within a variable name; use l**owercase for all other alphabetic characters.** First character of a variable name should be a lowercase alphabetic character.
	`balance // One-word variable name`
	`monthlyTotal // Two-word variable name`
	`exportDoc // Variable name with acronym`
	Precede a variable declaration with a **brief comment** describing the variable's **purpose.** Because Even the most descriptive variable names are terse.

3	**Try not to use Hungarian style** notation to tie variable type to variable name because it can distract from the main purpose of the variable name, which is to describe what the data contained in the variable means. Hungarian notation: `float fbalance`
4	**Initialize** a variable when declared, because ensures a known value for the variable.

Constants

1	Use **upper-case** alphanumeric characters in constant names. Separate words within a constant name by **underscores.** `THIS_IS_A_CONSTANT`
2	Try not to use a bare numeric constant (**magic number**), even if it is only used once.

3	Text strings that will be displayed to the user should always be pulled out into **string constants** or string tables. This allows them to be translated or updated as needed.
	`OPTION1 = "Select a tool"`
	`OPTION1 = "Selecciona una herramienta"`

Expression and statements

1	In languages where the assignment and comparison operators differ, be care with ambiguity when performing assignment inside conditional expressions. Cause of ambiguity: `If ( value = 3 + i )`
2	Try to have an **order** for comparison because is easier to understand. For example use < and <=, instead of > or >=.

Control structures

1	**Indent** blocks using the 'begin-end block boundaries' style and indent them at the **same level** as the code contained in the block.
	`if ( true == success )`
	`{`
	`// Do something.`
	`}`

2	Use **descriptive names** for loop indices, such as "row" and "col". If the loop is simple and small, traditional loop indices such as 'i', 'j', and 'k' are acceptable.
3	Using a **break** or **return** statement to exit a loop is **not recommended**.
4	**Prefer** `for` **loops** when they're appropriate, because makes the loop more readable and maintainable and prevent falling in infinite loops.
5	If you're including other **statements** in the `for` loop **header**, you should probably be using a `while` loop.

Functions

1	Follow the **same** form as **variable names** when naming functions, with the distinction that the **first** character of a function name should be **capitalized**

2	For a function whose main **purpose** is to **perform an action**, use a **verb-object** function name. If the method belongs to an object just use a **verb**.
	`PrintDocument( document );     //verb-object function name`
	`document.Print();              //verb function name`

3	For a function whose main purpose is to return a value, use a name that describes the value returned. `if ( FILE_OPEN == file.State() )`
4	Precede each function declaration with a comment describing the **intent of the function**, the **purpose of each argument**, and the function's **return value.**

	Note: Javadoc has its own tags to perform this action.
5	Using **multiple return statements** to exit a function is **not recommended.**

6.2 Pseudocode Example

Specification

A procedure must compute the insurance premium for a car driver, based on the following information:

- The person's age (integer).
- The person's sex ("M" for male, "F" for female).
- The person's marital status (married or not).

The business rules to compute the premium are the following:

- A basic premium of $500 applies for everyone.
- If the person is an unmarried male less than 25 years old, an extra cost of $1500 is added to the basic premium.
- If the person is married or is female, a rebate of $200 is applied on the basic premium.
- If the person is 45 years old or more and less than 65 years old, a rebate of $100 is applied on the basic premium.

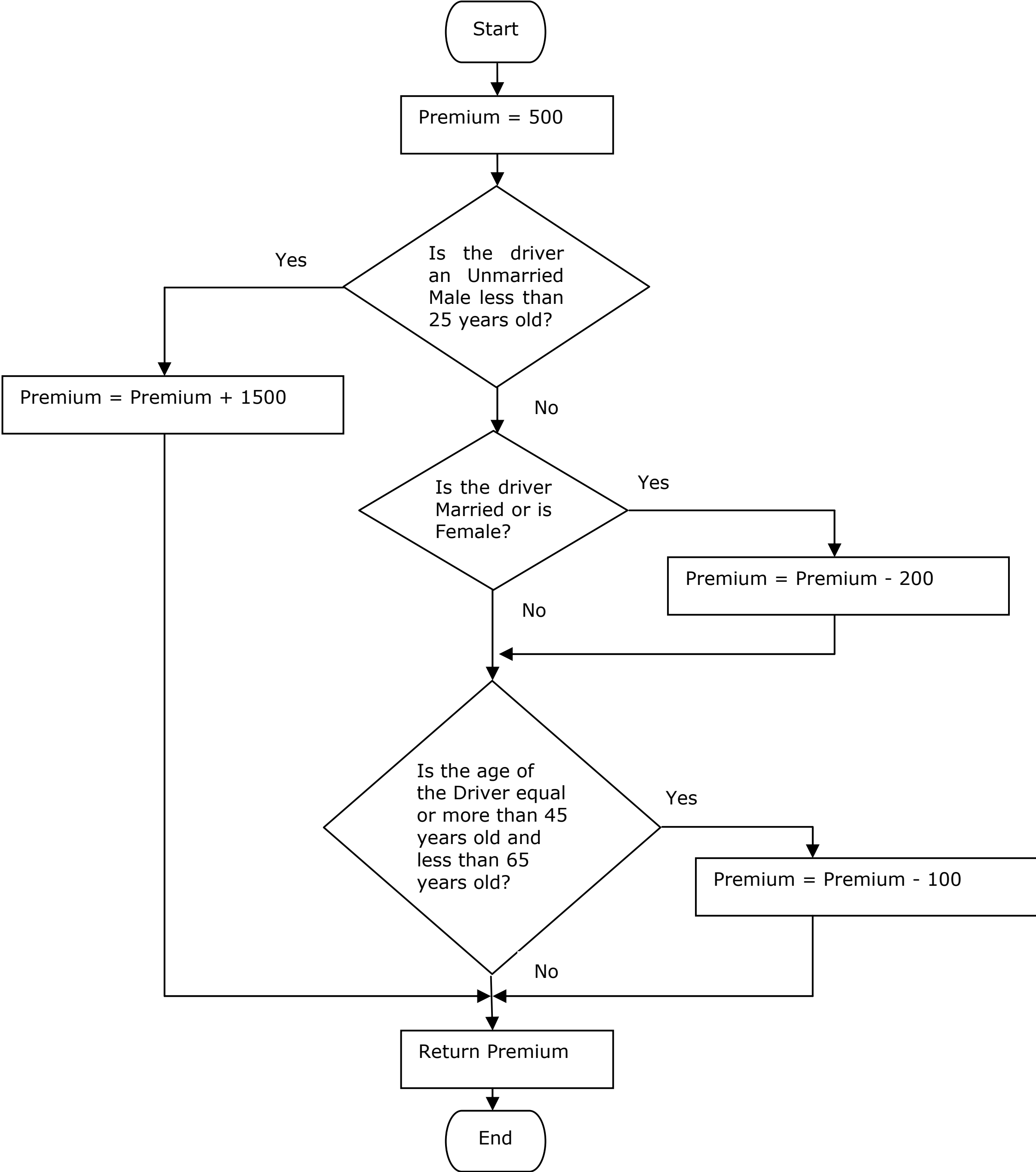
The procedure must return the related premium

Pseudocode based on the above specification

```
Set the cost of the basic premium to $500
If the car driver is an unmarried man less than 25 years old
        A cost of $1500 is added to the basic premium
Else
        If the driver is married or is female
                A rebate of $200 is applied on the basic premium
        If the driver is 45 years old or more and less than 65 years old
                A rebate of $100 is applied on the basic premium
Return premium
```

6.3 Flow chart example

This flow chart corresponds to the specification provided in example 6.2



6.4 Hierarchy of strengths from weakest to strongest of coverage criterion

Routine

```
x = 0
if(age < 65 && married == true)
{
	x = 2
	y = y + age
}
return (age/x)
```

The stronger the criterion, the more defects will be revealed by the tests.

Test cases for Statement coverage: All statements have to be executed at least once; only one test case is required to test this code with 100% statement coverage.

Test case ID	Description	Input		Expected output	Pass/Fail
		age	married		
TC1	A married person of 30 years old	30	True	15	P

Note: With this test case you execute all lines of code, but the case where the decision is false is never tested, so it doesn't detect the arithmetic defect age/x when x = 0.

Test cases for Branch/Decision coverage: Each decision must take each possible outcome at least once.

Decision: `age < 65 && married == true`

Test case ID	Description	Input		Decision outcome *(Compound predicate as a whole)*	Expected output	Pass /Fail
		age	Married			
TC1	A married person of 30 years old	30	True	True	15	P
TC2	A married person of 75 years old	75	True	False	0	F

Notes:

- *The test case TC2 detects a defect that statement coverage didn't.*
- *The value for married = False is never considered and even so 100% decision coverage is achieved. It happens because all possible values of each condition were not analyzed. So if a defect was hidden in condition 2 (married==true) it may not be detected.*

Test cases for Decision and Condition coverage

Each decision must take each possible outcome at least once, and each condition must take each possible outcome at least once.

Decision: `age < 65 && married == true`
Condition 1: `age < 65`
Condition 2: `married == true`

Test case ID	Description	Input		Condition 1 outcome	Condition 2 outcome	Decision outcome	Expected output	Pass /Fail
		Age	married					
TC1	A married person of 30 years old	30	True	True	True	True	15	P
TC2	A married person of 75 years old	75	True	False	True	False	0	F
TC3	An unmarried person of 30 years old	30	False	True	False	False	0	F

Notes: If condition 2 had a logical defect in its false outcome, then test case TC3 will detect it. Anyway this is not the case.

The criteria described above do not require the coverage of all the possible combinations of conditions. However, since the compilers usually stop evaluating an "*and*" expression as soon as it determines that one operand is *false or an* "or" expression as soon as it determines that one operand is *true*. Some of the conditions will not be evaluated regardless of their value.

To solve this problem it is recommended to test the "*and*" and "*or*" operators with the following order:

```
if( a && b && c && d )
```

Inputs	Values (T=true, F=false)
a,b,c,d	T T T T
a,b,c,d	F T T T
a,b,c,d	T F T T
a,b,c,d	T T F T
a,b,c,d	T T T F

```
if( a || b || c || d )
```

Inputs	Values (T=true, F=false)
a,b,c,d	F F F F
a,b,c,d	T F F F
a,b,c,d	F T F F
a,b,c,d	F F T F
a,b,c,d	F F F T

This way you ensure each condition takes its true and false outcome inside the execution of the program.

For satisfying the decision part of *decision and condition* coverage:

In the case of the "*and*" operators, the first set (T T T T) will work to evaluate the true outcome of the decision and any of the other cases will work to evaluate the false outcome.

In the case of the "*or*" operators, the set (F F F F) will work to evaluate the false outcome of the decision and any of the other cases will work to evaluate the true outcome.

Note: If the decision is compound of a mix between "and" and "or" operators you can use brackets "(,)" to determine the hierarchy of operators that bests suits you.

6.5 Test case design, unit test writing and use of a code coverage tool.

Routine

```
public int getPremium() {

		int premium = 500;

		if ((age < 25) && !married && sex.equals("M")) {
			premium += 1500;
		} else {
			if (married || sex.equals("F")) {
				premium -= 200;
			}
			if ((age >= 45) && (age < 65)) {
				premium -= 100;
			}
		}
		return premium;
}
```

Note: The unit tests were written using assertions with the input values of each test case and the expected output. For convenience the language chosen is Java, the Unit testing framework is Junit and the code coverage tool is Codecover.

Statement Coverage

To test all the statements is necessary to test the true and false outcomes of the first decision (first if)

Test Case ID	Description	Inputs	Expected outputs
TC1	An unmarried male of 20 years old	Age = 20, Unmarried, Male	2000
TC2	An married male of 50 years old	Age= 50, Married, Male	200

Table 4 Test cases required to satisfy 100% of statement coverage

Unit tests

```
public void testStatement(){
	// TC1
	pal.setAge(20);
	pal.setMarried(false);
	pal.setSex("M");
	assertEquals(pal.getPremium(), 2000);

	// TC2
	pal.setAge(50);
	pal.setMarried(true);
	pal.setSex("M");
	assertEquals(pal.getPremium(), 200);

}
```

Code coverage achieved

```
	int premium = 500;

	if ((age < 25) && !married && sex.equals("M")) {
		premium += 1500;
	} else {
		if (married || sex.equals("F")) {
			premium -= 200;
		}
		if ((age >= 45) && (age < 65)) {
			premium -= 100;
		}
	}
	return premium;
}
```

lems | @ Javadoc | Declaration | Boolean Analyzer | Coverage

v methods with Statement Coverage <= 90.5 %

Statement	Branch
100.0 %	66.7 %

Table 5 the unit tests satisfy 100% of Statement coverage. The color code has the following meaning: Green-Completely covered, Yellow-Partially covered, Red-Never reached (Codecover plugin for the Eclipse Java IDE)

As you see in the figure above, all statements were executed but not every outcome of each decision or condition has been exercised.

Branch/Decision Coverage

The decisions of the routine are:

D1	`if ((age < 25) && !married && sex.equals("M"))`
D2	`if (married || sex.equals("F"))`
D3	`if ((age >= 45) && (age < 65))`

Decision	True Outcome	False Outcome
D1	An unmarried male less than 25 years old	A 25 year old or more, or a married, or a female driver
D2	A married or female driver	An unmarried male driver
D3	A driver between 45 and 64 years old	A driver less than 45 years old or 65 years old or more

Table 6 Input situations to exercise all decision outcomes

Test Case ID	Description *(Testing the outcomes)*	Inputs	Expected outputs
TC1	D1T	Age = 20, Unmarried, male	2000
TC2	D1F,D2T,D3T	Age= 45, Married, male (irrelevant)	200
TC3	D1F,D2F,D3F	Age = 35, Unmarried, male	500

Table 7 Test cases obtained. Note: D#T means testing the true outcome of the related decision. D#F means testing the false outcome of the related decision.

Unit tests	Code coverage achieved

Unit tests:

```
public void testDecision(){
        // TC1
        pal.setAge(20);
        pal.setMarie(false);
        pal.setSexe("M");
        assertEquals(pal.getPrime(), 2000);

        // TC2
        pal.setAge(45);
        pal.setMarie(true);
        pal.setSexe("M");
        assertEquals(pal.getPrime(), 200);

        // TC3
        pal.setAge(35);
        pal.setMarie(false);
        pal.setSexe("M");
        assertEquals(pal.getPrime(), 500);
}
```

Code coverage achieved:

```
    int premium = 500;

    if ((age < 25) && !married && sex.equals("M")) {
        premium += 1500;
    } else {
        if (married || sex.equals("F")) {
            premium -= 200;
        }
        if ((age >= 45) && (age < 65)) {
            premium -= 100;
        }
    }
    return premium;
}
```

ms | @ Javadoc | Declaration | Boolean Analyzer | Coverage

nethods with Statement Coverage <= 90.5 %

Statement	Branch
100.0 %	100.0 %

Table 8 the unit tests satisfy 100% of Branch/Decision coverage. The color code has the following meaning: Green-Completely covered, Yellow-Partially covered (Codecover plugin for Eclipse Java IDE)

As you see in the figure above, the conditions of the decisions are partially covered (yellow colored code) since all their possible outcomes were not tested.

Decision and Condition Coverage

D1	`if ((age < 25) && !married && sex.equals("M"))`
D2	`if (married || sex.equals("F"))`
D3	`if ((age >= 45) && (age < 65))`

The conditions of the decisions are:

D1	C1	`age < 25`
D1	C2	`!married`
D1	C3	`sex.equals("M")`
D2	C4	`married`
D2	C5	`sex.equals("F")`
D3	C6	`age >= 45`
D3	C7	`age < 65`

Condition	**True Outcome**	**False Outcome**
C1	A driver less than 25 years old	A driver of 25 years old or older
C2	An unmarried driver	A married driver
C3	A male driver	A female driver
C4	A married driver	An unmarried driver
C5	A female driver	A male driver
C6	A driver of 45 years old or older	A driver less than 45 years old
C7	A driver less than 65 years old	A driver of 65 years old or older

Table 9 Input situations to evoke all condition outcomes

Test Case ID	**Description** *(Testing the outcomes)*		**Input**	**Expected output**
	Decisions	**Conditions**		
TC1	D1T	C1T,C2T,C3T	Age= 20, Unmarried, Male	2000
TC2	D1F	C1F,C2T,C3T	Age= 50, Unmarried, Male	400
TC3	D1F	C1T,C2F,C3T	Age= 20, Married, Male	300
TC4	D1F	C1T,C2T,C3F	Age= 20, Unmarried, Female	300
TC2	D2F	C4F,C5F	Age= 50, Unmarried, Male	400
TC3	D2T	C4T,C5F	Age= 20, Married, Male	200
TC4	D2T	C4F,C5T	Age= 20, Unmarried, Female	200
TC2	D3T	C6T,C7T	Age= 50, Unmarried, Male	400
TC5	D3F	C6F,C7T	Age= 30, Unmarried, Male	500
TC6	D3F	C6T,C7F	Age= 70, Unmarried, Male	500

Table 10 Test cases obtained. Note: C#T means testing the true outcome of the related condition. C#F means testing the false outcome of the related decision

Note 1: The test cases were obtained following the advice given in the example 6.4 for this coverage criterion.

Note 2: Remember that sometimes the same input configuration can test more than one test case. So always try to optimize the number of test cases.

Unit tests

```
public void testDecisionCondition(){

        // TC1
        pa1.setAge(20);
        pa1.setMarried(false);
        pa1.setSex("M");
        assertEquals(pa1.getPremium(), 2000);

        // TC2
        pa1.setAge(50);
        pa1.setMarried(false);
        pa1.setSex("M");
        assertEquals(pa1.getPremium(), 400);

        // TC3
        pa1.setAge(20);
        pa1.setMarried(true);
        pa1.setSex("M");
        assertEquals(pa1.getPremium(), 300);

        // TC4
        pa1.setAge(20);
        pa1.setMarried(false);
        pa1.setSex("F");
        assertEquals(pa1.getPremium(), 300);

        // TC5
        pa1.setAge(30);
        pa1.setMarried(false);
        pa1.setSex("M");
        assertEquals(pa1.getPremium(), 500);

        // TC6
        pa1.setAge(70);
        pa1.setMarried(false);
        pa1.setSex("M");
        assertEquals(pa1.getPremium(), 500);

}
```

Table 11 the unit tests satisfy 100% of decision and condition coverage. But the coverage tool used does not measure this criterion. However there are other tools that can measure this criterion.

With this set of unit tests each decision and condition takes each possible outcome at least once.

As you see to satisfy stronger criteria are needed typically more test cases. But the components become better tested and with less holes where defects can hide.

6.6 Unit testing following Structured testing technique and mapping to the steps of the task "Design or update unit tests cases and apply them".

The purpose of the structured testing is automatically guarantee both branch/decision and statement coverage. The structured testing process consists of the following steps:

1. Derive the control flow graph from the software module

All structured programs can be built from three basic structures: sequential (e.g., assignment statements), decision (e.g., if/then/else statements), and iterative (e.g., while, for loops).

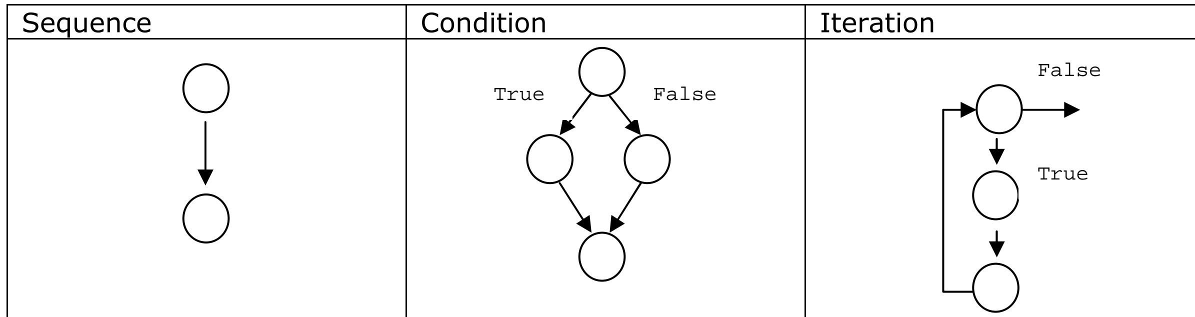
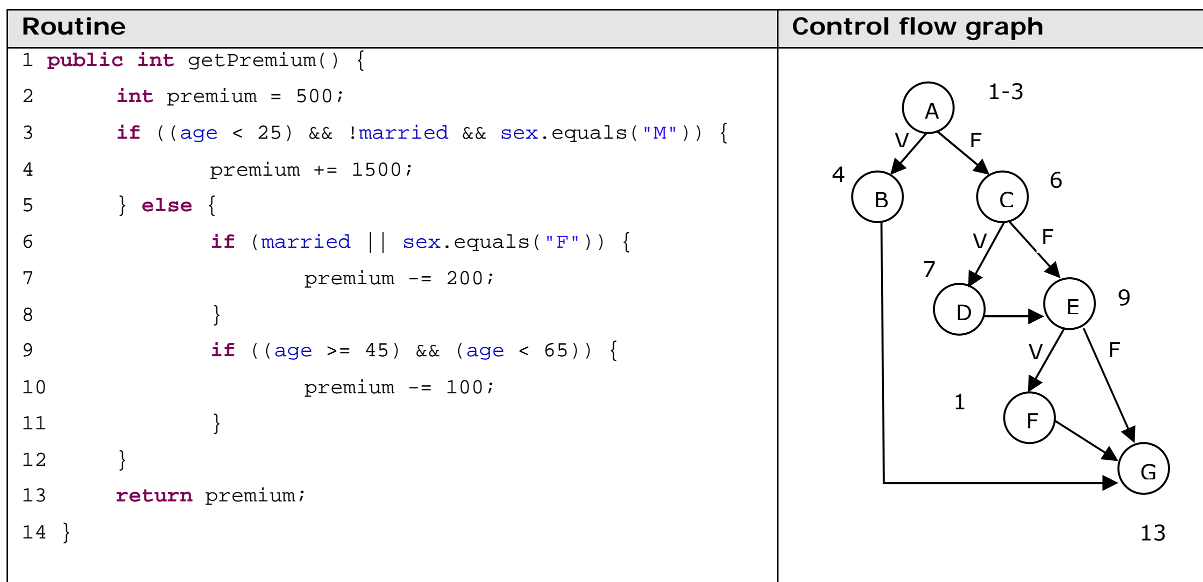


Figure 5 Graphical representations for the three basic structures

Using these structures and a combination of them a control flow graph for the software under test can be developed.



Routine	Control flow graph
(code below)	(graph)

```
1 public int getPremium() {
2       int premium = 500;
3       if ((age < 25) && !married && sex.equals("M")) {
4               premium += 1500;
5       } else {
6               if (married || sex.equals("F")) {
7                       premium -= 200;
8               }
9               if ((age >= 45) && (age < 65)) {
10                      premium -= 100;
11              }
12      }
13      return premium;
14 }
```


Note: There are commercial tools that will generate control flow graphs from code and in some cases from pseudo code.

2. Compute the graph's cyclomatic complexity (C)

Once you have the control flow graph of the unit you want to test, you determine the minimum number of test cases needed to cover all the statements and decisions. This number is called cyclomatic complexity[10] and is calculated by the following equation (assuming a single entry node and a single output node):

$$C = edges - nodes + 2$$

---

[10] McCabe (December 1976). "A Complexity Measure"

In our example, the cyclomatic complexity is: C = 9 – 7 + 2 = 4. You therefore need at least four test cases to satisfy this criterion.

Note: The step 1 "Obtain the exit criteria" of the task "Design or update unit tests cases and apply them" is set by default to satisfy Branch/Decision coverage, since it is the objective of structure testing.

3. Select a set of basis paths

To determine the set of basis paths[11] you should follow this process:

- Pick a "baseline" path. This path should be a reasonably "typical" path of execution rather than an exception processing path.
- To choose the next path, change the outcome of the first decision along the baseline path while keeping the maximum number of other decisions the same as the baseline path.
- To generate the third path, begin again with the baseline but vary the outcome of the second decision rather than the first.
- To generate the fourth path, begin again with the baseline but vary the outcome of the third decision rather than the second. Continue varying each decision, one by one, until the bottom of the graph is reached.
- This pattern is continued until the basis path set is complete.

Following the steps from above and choosing as baseline A-B-G the resulting basis paths are:

- A-B-G
- A-C-D-E-F-G
- A-C-E-F-G
- A-C-D-E-G

4. Create a test case for each basis path

The next step is to create a test case for each path. This set of test cases will guarantee both statement and branch coverage. Note that multiple sets of basis paths can be created and are not necessarily unique. Each set, however, has the property that a set of test cases based on it will execute every statement and every branch.

Test case ID	**Description** (Path covered)	**Input**			**Expected output**
		Age	**Married**	**Sex**	
TC1	A-B-G	20	False	M	2000
TC2	A-C-D-E-F-G	45	True	M	200
TC3	A-C-E-F-G	50	False	M	400
TC4	A-C-D-E-G	30	True	M	300

[11] A basis path set is the minimum number of independent, non-looping paths that can, in linear combination, generate all possible paths through the module. In terms of a control flow graph, each basis path traverses at least one edge that no other path does.

Note: The step 2 "Design the test cases" of the task "Design or update unit tests cases and apply them" is covered with this step.

Note: This set of test cases never tests the input Sex with the value = "False" and even so it covers the whole graph. To be stricter a Decision and Condition criterion could cover this issue

5. Execute the tests

This step can be divided into two sub-steps:

5.1.Code the unit tests

The unit tests for the test cases are

Unit tests

```
public void testBasisPath(){
        // TC1 Path:A-B-G
        pa1.setAge(20);
        pa1.setMarried(false);
        pa1.setSex("M");
        assertEquals(pa1.getPremium(), 2000);

        // TC2 Path:A-C-D-E-F-G
        pa1.setAge(45);
        pa1.setMarried(true);
        pa1.setSex("M");
        assertEquals(pa1.getPremium(), 200);

        // TC3 Path:A-C-E-F-G
        pa1.setAge(50);
        pa1.setMarried(false);
        pa1.setSex("M");
        assertEquals(pa1.getPremium(), 400);

        // TC4 Path:A-C-D-E-G
        pa1.setAge(30);
        pa1.setMarried(true);
        pa1.setSex("M");
        assertEquals(pa1.getPremium(), 300);

}
```

Note: The step 3 "Code the unit tests" of the task "Design or update unit tests cases and apply them" is covered with this sub-step

5.2.Execute the unit tests

For the unit tests execution you can use a command line or a graphical test runner. At the end if some of them fail the framework will show you which one. So in that case you need to make the pertinent corrections. (*See the example 6.7 for defect correction)*

For this example all the unit tests passed.

Test case ID	Description: Path covered	Input			Expected output	Pass/Fail
		Age	Married	Sex		
1	A-B-G	20	False	M	2000	P

2	A-C-D-E-F-G	45	True	M	200	P
3	A-C-E-F-G	50	False	M	400	P
4	A-C-D-E-G	30	True	M	300	P

Note: The step 4 "Execute the unit tests" of the task "Design or update unit tests cases and apply them" is covered with this sub-step

Note: The step 5 "Analyze the results" of the task "Design or update unit tests cases and apply them" could be covered after this step. The structured testing process does not mention it but it would be implicit.

6.7 Defect correction

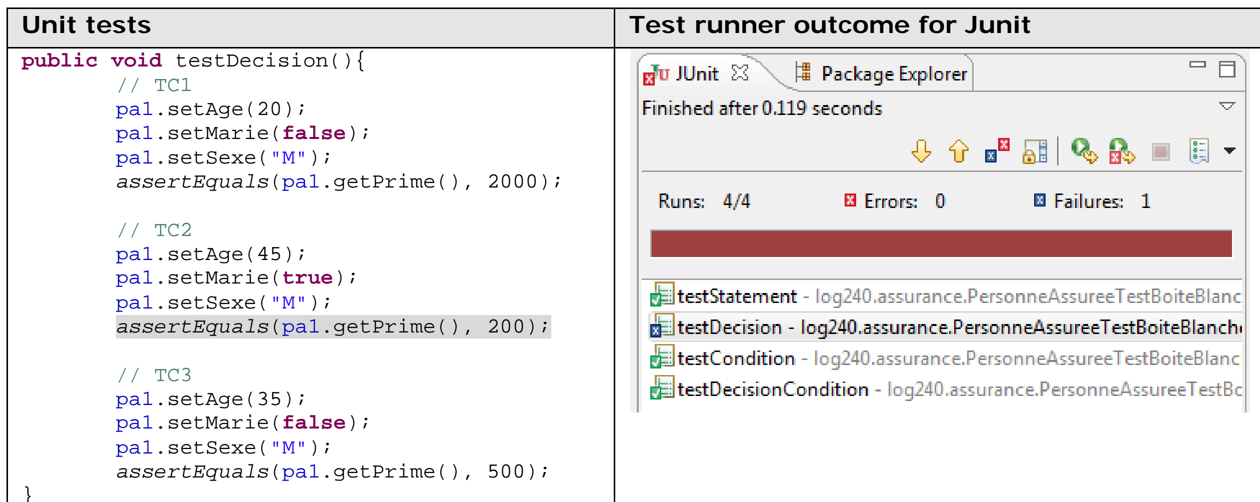
The following change to the source code of the example 6.5 injects a defect:

Routine

```
1 public int getPremium() {
2       int premium = 500;
3       if ((age < 25) && !married && sex.equals("M")) {
4               premium += 1500;
5       } else {
6               if (married || sex.equals("F")) {
7                       premium -= 200;
8               }
9               if ((age > 45) && (age < 65)) {
10                      premium -= 100;
11              }
12      }
13      return premium;
14 }
```

When the unit tests designed for Decision coverage are executed an error is revealed and one of the tests failed.

Test Case ID	Description	Inputs	Expected outputs	Pass/ Fail
TC1	An unmarried male driver of 20 years old	Age= 20, Unmarried, Male	2000	P
TC2	A married driver of 45 years old	**Age= 45**, Married, Male	200	**F**
TC3	An unmarried male driver of 35 years old	Age= 35, Unmarried, Male	500	P



Unit tests

```
public void testDecision(){
        // TC1
        pal.setAge(20);
        pal.setMarie(false);
        pal.setSexe("M");
        assertEquals(pal.getPrime(), 2000);

        // TC2
        pal.setAge(45);
        pal.setMarie(true);
        pal.setSexe("M");
        assertEquals(pal.getPrime(), 200);

        // TC3
        pal.setAge(35);
        pal.setMarie(false);
        pal.setSexe("M");
        assertEquals(pal.getPrime(), 500);
}
```

Test runner outcome for Junit

Table 12 when executing the unit tests the Test runner shows a defect. Expected output: 200, Output obtained: 300

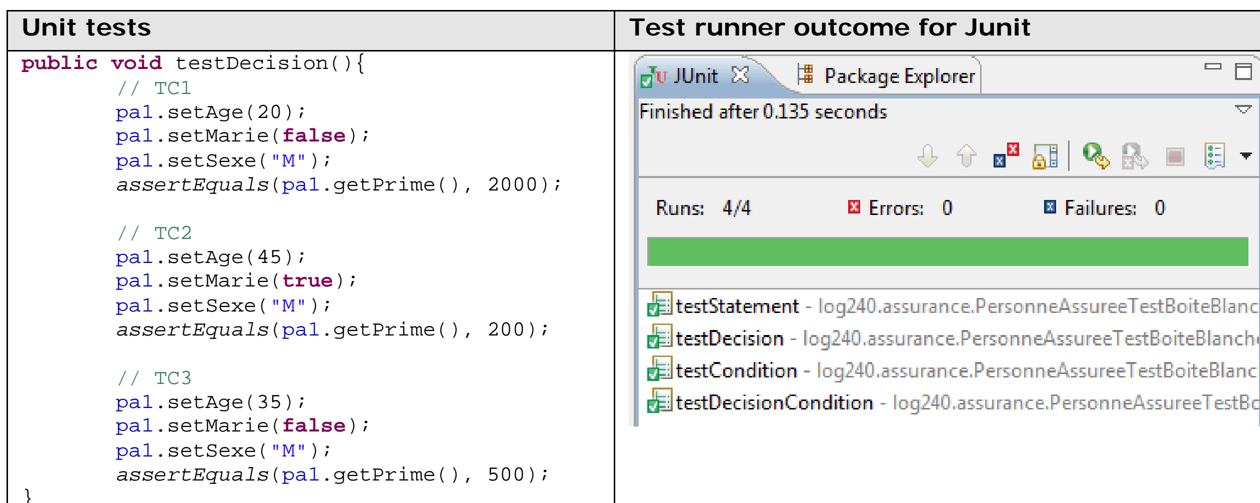
Since the error is not in the test method but in the implementation code. The next step is finding the location of the defect.

```
Line 9          if ((age > 45) && (age < 65)) {
```

Once you have found the defect, you should see if there is no other defect around the code. If it is not the case you should perform the fix and then run the set of unit test again to see if have not "broken" functionality that was previously working.

```
Line 9          if ((age >= 45) && (age < 65)) {
```

Test Case ID	Description	Inputs	Expected outputs	Pass/Fail
TC1	An unmarried male driver of 20 years old	Age= 20, Unmarried, Male	2000	P
TC2	A married driver of 45 years old	Age= 45, Married, Male	200	P
TC3	An unmarried male driver of 35 years old	Age= 35, Unmarried, Male	500	P



Unit tests

```
public void testDecision(){
        // TC1
        pal.setAge(20);
        pal.setMarie(false);
        pal.setSexe("M");
        assertEquals(pal.getPrime(), 2000);

        // TC2
        pal.setAge(45);
        pal.setMarie(true);
        pal.setSexe("M");
        assertEquals(pal.getPrime(), 200);

        // TC3
        pal.setAge(35);
        pal.setMarie(false);
        pal.setSexe("M");
        assertEquals(pal.getPrime(), 500);
}
```

Test runner outcome for Junit

Table 13 once the defect is corrected the unit tests are executed again and they passed successfully.

Now that your component passed all your tests you should decide if you stop testing or continue creating more test cases. This depends on the exit criteria that your project manager asked you.

6.8 Test runners examples

Command line Test runner

The following is an example of running a unit test in rubyUnit from the command line:

```
>ruby testrunner.rb c:/examples/tests/MySuiteTest.rb
Loaded suite MySuiteTest
Started
....Finished in 0.014 seconds.4 tests, 5 assertions, 0 failures, 0 errors
>Exit code: 0
```

The first line is the invocation at the command prompt. In this example it is running the tests defined in MySuiteTest. The next two lines are the initial feedback as it starts up. The series of dots indicate progress, one per test completed. The last two lines are the summary statistics that provide an overview of what happened. Typically, the exit code is set to the total number of failed tests.

Graphical Test runner

The following are screenshots of two graphical test runners.

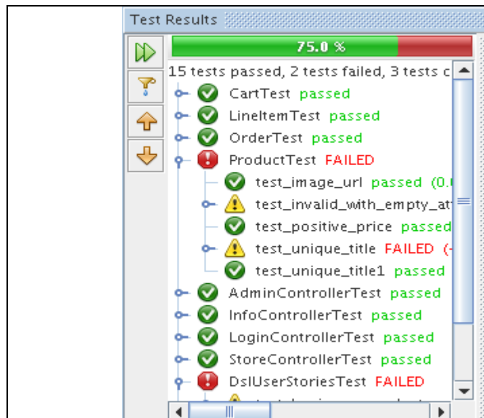


Figure 6. Test runner for Ruby on Rails

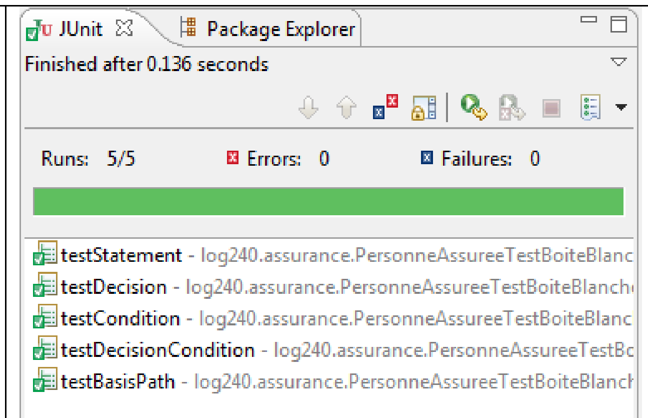
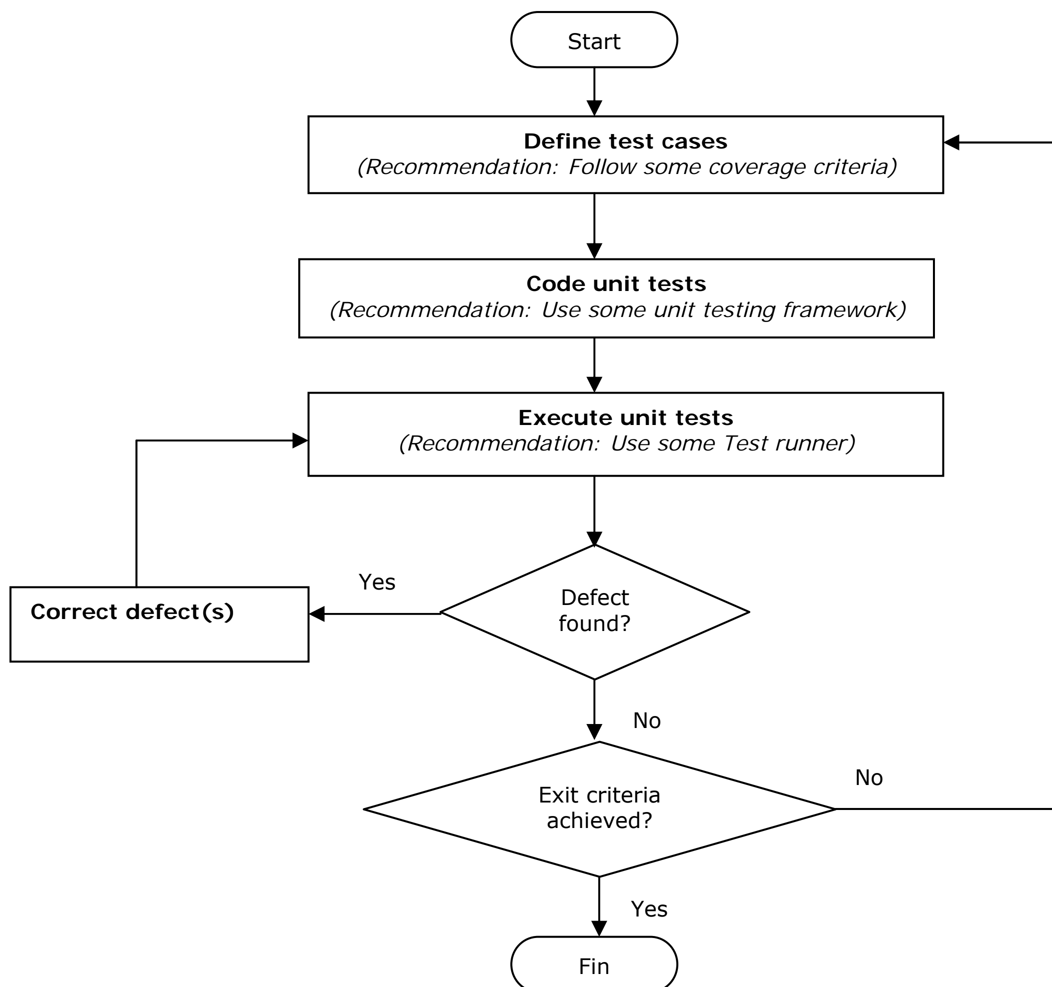


Figure 7. Test runner for Junit4 in Eclipse Java EE IDE

6.9 C++ Example of an assertion macro

```
#define ASSERT ( condition, message ){
      if( !(condition) ){
            fprintf (stderr, "Assertion %s failed: %s\n",
                     #condition, message);
            exit( EXIT_FAILURE );
        }
}
```

6.10 Unit tests life cycle



7. Checklist

Task Checklists

7.1 Assign task to the members of the work team

☐	Have you obtained the software design documentation?
☐	Have you selected a integration sequence strategy?
☐	Have you detailed the project schedule?
☐	Have you allocated tasks to members of the work team?
☐	Have you defined exit criteria for unit testing?

7.2 Construct or update software components

☐	Have you understood the detail design and component contribution?
☐	Have you looked for functionalities available in the standard libraries?
☐	Have you defined the logic of the component?
☐	Have you followed the construction standard while coding?
☐	Have you verified the component?

7.3 Design or update unit test cases and apply them

☐	Have you obtained the exit criteria?
☐	Have you designed the test cases?
☐	Have you coded the unit tests?
☐	Have you executed the unit tests?
☐	Have you analyzed the results?

7.4 Correct the defects

☐	Have you confirmed the defect?
☐	Have you determined the location of the suspected error?
☐	Have you looked for similar errors around the code you just found?
☐	Are you correcting the error from the source and not just patched?
☐	Have you verified the corrections?
☐	Do your fix changes totally the design of the component? (If it does, say it to your technical leader).

Support Checklists

7.5 Code review checklist

Note: This checklist can be adapted to apply to the language you are programming in.

A tag is added in front of each item of the checklist to help accelerate the recording anomalies

TAG - Subject		Description
☐	**CR1 Complete**	- Verify that all functions in the design are coded and that all necessary functions and procedures have been implemented.
☐	**CR2 Logic**	- Verify that the program flow and all procedure and function logic is consistent with the detailed design.
☐	**CR3 Loops**	- Ensure that every loop is properly initiated and terminated. - Check that every loop is executed the correct number of times.
☐	**CR4 Calls**	- Check every function and procedure call to insure that it exactly matches the definition for formats and types.
☐	**CR5 Declarations**	Verify that each variable and parameter: - has exactly one declaration - is only used within its declared scope - is spelled correctly wherever used
☐	**CR6 Initialization**	- Check that every variable is initialized.
☐	**CR7 Limits**	- Check all variables, arrays, and indexes to ensure that their use does not exceed declared limits.
☐	**CR8 Begin-end**	- Check all begin-end pairs or equivalents, including cases where nested ifs could be misinterpreted.
☐	**CR9 Boolean**	- Check Boolean conditions
☐	**CR10 Format**	- Check every program line for instruction format, spelling, and punctuation.
☐	**CR11 Pointers**	- Check that all pointers are properly used.
☐	**CR12 Input-output**	- Check all input-output formats.
☐	**CR13 Spelling**	- Check that every variable, parameter, and key work is properly spelled.
☐	**CR14 Comments**	- Ensure that all commenting is accurate and according to standard.
☐	**CR15 Calculation**	- Check that all arithmetic operations and equations express what the algorithm indicate.
☐	**CR16 Data Base**	- Ensure that all connections are closed - Verify that the driver connection is written properly

7.6 What the architect and designer should provide

Requirements

☐	Should any requirement be specified in more detail before construction?
☐	Are the inputs specified, including their source, accuracy, range of values, and frequency?
☐	Are the outputs specified, including their destination, accuracy, range of values, frequency, and format (including web pages, reports, and so on)?
☐	Are all the external hardware and software interfaces specified?
☐	Is maximum memory/storage specified?
☐	Are timing considerations specified, such as expected response time, processing time, data- transfer rate, and system throughput?
☐	Is the level of security specified?

Architecture and Design

☐	Are the most critical components or data design described and justified?
☐	Is the database organization and content specified?
☐	Is a strategy for the user interface design described?
☐	Is a strategy for handling I/O described and justified?
☐	Are the architecture's security requirements described?
☐	Is a coherent error-handling strategy provided?

7.7 Sub-task: Select the user interface standard

☐	Have you planned the sub-task?
☐	Have you obtained available interface standards?
☐	Have you selected the interface standard?
☐	Have you obtained approval from customer?
☐	Have your programmers adopted the standard?
☐	Have you verified the adoption of the standard?

8. Tool

8.1 Traceability Matrix

- Objectives:
  - To maintain the linkage from the source of each requirement through its decomposition to implementation and test (verification).
  - To ensure that all requirements are addressed and that only what is required is developed.
  - Useful when conducting impact assessments of requirements, design or other configured item changes.

Note: A Traceability Matrix is a widespread used tool to implement the Traceability Record.

Traceability Matrix

Date (yy-mm-dd): _______________

Title of project:___________________________________

	Name (Print)	**Signature**	**Date (yy-mm-dd)**
Verified by:	___________________________________	________________________________	___________________
Approved by:	___________________________________	________________________________	__________________

Identification Number	Text of the need	Text of the requirement	Verification method	Title or ID of Use Case	Title or ID of Code Module	Title or ID of test Procedure	Verification Date	Name of person who performed the verification	Result of verification

Instructions	
The above table should be created in a spreadsheet or database such that it may be easily sorted by each column to achieve bi-directional traceability between columns. The unique identifiers for items should be assigned in a hierarchical outline form such that the lower level (i.e. more detailed) items can be traced to higher items.	
Unique Requirement Identification (ID)	The Unique Requirement ID / System Requirement Statement where the requirement is referenced, and/or the unique identification (ID) for decomposed requirements
Requirement Description	Enter the description of the requirement (e.g., Change Request description).
Design Reference	Enter the paragraph number where the CR is referenced in the design documentation
Module / Configured Item Reference	Enter the unique identifier of the software module or configured item where the design is realized.
Release Reference	Enter the release/build version number where the requirement is fulfilled

Test Script Name/Step Number Reference	Enter the test script name/step number where the requirement is referenced (e.g., Step 1)

Guideline

Requirements traceability should:

- Ensure traceability for each level of decomposition performed on the project. In particular:
  - Ensure that every lower level requirement can be traced to a higher level requirement or original source
  - Ensure that every design, implementation, and test element can be traced to a requirement
  - Ensure that every requirement is represented in design and implementation
  - Ensure that every requirement is represented in testing/verification
- Ensure that traceability is used in conducting impact assessments of requirements changes on project plans, activities and work products
- Be maintained and updated as changes occur.
- Be consulted during the preparation of Impact Assessments for every proposed change to the project
- Be planned for, since maintaining the links/references is a labor intensive process that should be tracked/monitored and should be assigned to a project team member
- Be maintained as an electronic document

8.2 Code coverage tools

Links to the most popular coverage tools

Tool	**Language**	**Source**
CodeCover	Java	http://codecover.org/
Cobertura	Java	http://cobertura.sourceforge.net/
Bullseye	C/C++	http://www.bullseye.com/
NCover	C# .NET	http://www.ncover.com/
PHPUnit - Xdebug	PHP	http://xdebug.org/

8.3 Unit testing frameworks

Links to the most popular Unit testing frameworks

Tool	Language	Source
JUnit	Java	http://www.junit.org/
NUnit	C# .NET	http://www.nunit.org/
CppUnit 2	C++	https://launchpad.net/cppunit2
PHPUnit	PHP	http://phpunit.sourceforge.net/
DUnit	Delphi	http://dunit.sourceforge.net/
PyUnit	Python	http://pyunit.sourceforge.net/

9. Reference to Other Standards and Models

This section provides references of this deployment package to selected ISO and ISO/IEC Standards and to the Capability Maturity Model Integration[SM] version 1.2 of the Software Engineering Institute (CMMI[®12]).

Notes:

- This section is provided for information purpose only.
- Only tasks covered by this Deployment Package are listed in each table.
- The tables use the following convention:
  - Full Coverage = F
  - Partial Coverage = P
  - No Coverage = N

ISO 9001 Reference Matrix

Title of the Task and Step	Coverage F/P/N	Clause of ISO 9001	Comments
SI.2.2 Document or update Requirements Specification.	P	7.3.2 Design and development inputs d) other requirements essential for design and development.	
Sub-task: Define Construction Standards.			Most of the cases they are not specified by customers, but are essential for some components.
SI.4.1 Assign tasks to the Work Team members related to their role, according to the current *Project Plan.*	P	7.3.1 Design and development planning	It only includes communicating tasks.
SI.4.3 Construct or update *Software Components* based on the detailed part of the *Software Design*.	P	7.3.3 Design and development outputs. a) meet the input requirements for design and development.	

[SM] CMM Integration is a service mark of Carnegie Mellon University.

[®] Capability Maturity Model, CMMI are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

SI.4.4 Design or update unit test cases and apply them to verify that the Software Components implements the detailed part of the *Software Design.*	P	7.3.4 Design and development review a) to evaluate the ability of the results of design and development to meet requirements, and b) to identify any problems and propose necessary actions.	
SI.4.5 Correct the defects found until successful unit test (reaching exit criteria) is achieved.	P	7.3.4 Design and development review a) to evaluate the ability of the results of design and development to meet requirements, and b) to identify any problems and propose necessary actions.	
SI.4.6 Update the Traceability Record incorporating Software Components constructed or modified.	P	7.3.7 Manage design and development changes.	

ISO/IEC 12207 Reference Matrix

Title of the Task and Step	Coverage F/P/N	Clause of ISO/IEC 12207	Comments
SI.2.2 Document or update Requirements Specification.	P	7.1.2.3.1 Software requirements analysis. 7.1.2.3.1.1 The implementer shall establish and document software requirements (including the quality characteristics specifications) described below. b) Interfaces external to the software item. k) User maintenance requirements.	a) Only covers requirements that affect the Construction phase. b) Is related with UI, and k) with Construction Standards.
Sub-task: Select the User Interface standard.		7.1.4.3.1 Software detailed design. 7.1.4.3.1.2 The implementer shall develop and document a detailed design for the interfaces external to the software item, between the software components, and between the software units. The detailed design of the interfaces shall permit coding without the need for further information.	The UI standard will be useful when performing 7.1.4.3.1.2

Sub-task: Define Construction Standards.		7.1.5.3.1 Software construction. 7.1.5.3.1.5 The implementer shall evaluate software code and test results considering the criteria listed below. The results of the evaluations shall be documented. e) Appropriateness of coding methods and standards used.	Construction Standards will be useful when performing 7.1.5.3.1.5
SI.4.1 Assign tasks to the Work Team members related to their role, according to the current *Project Plan.*	P	6.3.1.3.3 Project activation. 6.3.1.3.3.3 The manager shall initiate the implementation and criteria set, exercising control over the project.	This task only activates the Construction phase.
SI.4.3 Construct or update *Software Components* based on the detailed part of the *Software Design*.	P	7.1.5.3.1 Software construction. 7.1.5.3.1.1 The implementer shall develop and document the following: a) Each software unit and database.	
SI.4.4 Design or update unit test cases and apply them to verify that the Software Components implements the detailed part of the *Software Design.*	P	7.1.5.3.1 Software construction. 7.1.5.3.1.1 The implementer shall develop and document the following: b) Test procedures and data for testing each software unit and database.	
SI.4.5 Correct the defects found until successful unit test (reaching exit criteria) is achieved.	P	7.1.5.3.1 Software construction. 7.1.5.3.1.2 The implementer shall test each software unit and database ensuring that it satisfies its requirements. The test results shall be documented.	
SI.4.6 Update the Traceability Record incorporating Software Components constructed or modified.	P	7.1.5.3.1 Software construction. 7.1.5.3.1.5 The implementer shall evaluate software code and test results considering the criteria listed below. The results of the evaluations shall be documented. a) Traceability to the requirements and design of the software item.	

CMMI Reference Matrix

Title of the Task and Step	Coverage F/P/N	PA/Objective/ Practice of CMMI V1.2	Comments

SI.2.2 Document or update Requirements Specification.	P	Requirements Development (RD) SG 1 Develop Customer Requirements SP 1.2 Develop the Customer Requirements	Constraints for verification and validation are not covered.
Sub-task: Select the User Interface standard.			Stakeholders of the Construction phase may provide them
Sub-task: Define Construction Standards.			Stakeholders of the Construction phase may provide them
SI.4.1 Assign tasks to the Work Team members related to their role, according to the current *Project Plan.*	P	Technical Solution(TS) SG 3 Implement the Product Design GP 2.8 Monitor and Control the Process	It only activates the plan execution.
SI.4.3 Construct or update *Software Components* based on the detailed part of the *Software Design*.	P	Technical Solution(TS) SG 3 Implement the Product Design SP 3.1 Implement the Design	Covers subpractices 1. Use effective methods to implement the product components. and 2. Adhere to applicable standards and criteria.
SI.4.4 Design or update unit test cases and apply them to verify that the Software Components implements the detailed part of the *Software Design.*	P	Technical Solution(TS) SG 3 Implement the Product Design SP 3.1 Implement the Design.	Covers subpractice 4. Perform unit testing of the product component as appropriate.
SI.4.5 Correct the defects found until successful unit test (reaching exit criteria) is achieved.	P	Technical Solution(TS) SG 3 Implement the Product Design SP 3.1 Implement the Design.	Covers subpractice 4. Perform unit testing of the product component as appropriate.
SI.4.6 Update the Traceability Record incorporating Software Components constructed or modified.	P	Requirements Management (REQM) SG 1 Manage Requirements SP 1.4 Maintain Bidirectional Traceability of Requirements	It only covers the traceability of Components constructed.

10. References

Key	Reference
[Code Complete]	Steve McConnell, Code Complete, Second Edition, Redmond, Washington, Microsoft Press, 2004.
[Art of Software testing]	Glenford J. Myers, The Art of Software Testing, Second Edition, 2004.
[Practitioner's Guide]	Lee Copeland, A Practitioner's Guide to Software Test Design, 2004
[Defect Prevention]	Marc McDonald, The Practical Guide To Defect Prevention, 2008
[Introduction to Software Testing]	Paul Ammann & Jeff Offutt, Introduction to Software testing, 2008
[Testing Computer Software]	Cem Kaner, Testing Computer Software
[Pratical Software Testing]	Ilene Burnstein, Practical Software Testing, 2002
[SE Support Activities for VSE]	Vicent Ribaud, Software Engineering Support Activities for Very Small Entities, 2010
[Application of ISES in VSE]	Claude Y. Laporte, The application of International Software Engineering Standards in Very Small Enterprises, 2008
[A SE Lifecycle Standard for VSEs]	Claude Y. Laporte, A Software Engineering Lifecycle Standard for Very Small Enterprises, 2008
[Misuse Code Coverage]	Brian Marick, How to Misuse Code Coverage, 1999
[IEEE 1012-2004]	IEEE 1012-2004 IEEE Standard for Software Verification and Validation, IEEE Computer Society
[ISO/IEC 12207]	ISO/IEC 12207:2008 Systems and software engineering – Software life cycle processes.
[ISO/IEC 29110-5-1-2]	ISO/IEC TR 29110-5-1-2:2011, Software Engineering—Lifecycle Profiles for Very Small Entities (VSEs) – Part 5-1-2: Management and Engineering Guide – Basic VSE Profile
[ISO/IEC 24765]	ISO/IEC 24765:2011 Systems and software engineering vocabulary
[ISO/IEC 20926]	ISO/IEC 20926:2003 Software engineering -- IFPUG 4.1 Unadjusted functional size measurement method -- Counting practices manual
[ISO/IEC 29881:2008]	ISO/IEC 29881:2008 Information technology--Software and systems engineering--FiSMA 1.1 functional size measurement method,
[IEEE 1233-1998]	IEEE Guide for Developing System Requirements Specifications

11. Evaluation Form

Deployment Package Construction and Unit Test Version 0.4 Your feedback will allow us to improve this Deployment Package, your comments and suggestions are welcomed.
1. How satisfied are you with the CONTENT of this deployment package? *Very Satisfied* *Satisfied* *Neither Satisfied nor Dissatisfied* *Dissatisfied* *Very Dissatisfied*
2. The sequence in which the topics are discussed, are logical and easy to follow? *Very Satisfied* *Satisfied* *Neither Satisfied nor Dissatisfied* *Dissatisfied* *Very Dissatisfied*
3. How satisfied were you with the APPEARANCE/FORMAT of this deployment package? *Very Satisfied* *Satisfied* *Neither Satisfied nor Dissatisfied* *Dissatisfied* *Very Dissatisfied*
4. Have any unnecessary topics been included? (please describe)
5. What missing topic would you like to see in this package? (please describe) • Proposed topic: • Rationale for new topic
6. Any error in this deployment package? • Please indicate: • Description of error : • Location of error (section #, figure #, table #) :
7. Other feedback or comments:
8. Would you recommend this Deployment package to a colleague from another VSE? *Definitely* *Probably* *Not Sure* *Probably Not* *Definitely Not*

Optional

- Name: ____
- e-mail address : ____________________________________

Email this form to: claude.y.laporte@etsmtl.ca or Avumex2003@yahoo.com.mx or thexaviermail@gmail.com