[Plataforma UdeG]

Modelo de Diseño Construcción de Software

Versión [1.0]

[Plataforma simple creada a partir de html,css,jquery y sql para consulta de información académica ]

Historia de revisiones

|  |  |  |  |
| --- | --- | --- | --- |
| Fecha | Versión | Descripción | Autor |
| [20/11/2019] | [1.0] | [Posee datos del usuario] | [Juan Pablo Padilla] |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Contenido

[Plataforma UdeG] 1

Modelo de Diseño Construcción de Software 1

Versión [1.0] 1

Historia de revisiones 1

Contenido 2

1. Introducción 3

1.1. Propósito 3

1.2. Alcance 3

1.3. Definiciones, siglas y abreviaturas. 3

1.4. Referencias 3

1.5. Visión general 3

2. Diseño de Casos de Uso 3

2.1. Diseño del Caso de Uso [nombre del caso de uso 1] 3

2.1.1. Diagrama de paquetes 3

2.1.2. Diagrama de Interacción 3

2.1.3. Diseño de Flujo de eventos 3

2.1.4. Requerimientos especiales o de implementación 4

2.2. Diseño del Caso de Uso [nombre del caso de uso 2] 4

3. Diseño de Objetos 4

3.1. [Objeto 1] 4

3.2. [Objeto 2] 4

4. Diseño de Subsistemas 4

4.1. Subsistemas Específicos 4

4.1.1. [Nombre del Subsistema Específico 1] 4

4.1.2. [Nombre del Subsistema Específico 2] 5

4.2. Subsistemas de Soporte 5

4.2.1. [Nombre del Subsistema de soporte 1] 5

4.2.2. [Subsistema de soporte 2] 5

5. Revisión de la Interfaz de Usuario 5

5.1. Formatos individuales de interfaz de pantalla 6

5.2. Controles y elementos de diseño de interfaz de pantalla 6

5.3. Forma de navegación de interfaz de pantalla 6

5.4. Formatos de impresión 6

6. Diseño de Datos 6

6.1. Modelo de Datos del Núcleo 6

6.2. Modelo de Datos General 6

6.3. Especificación de la Distribución de Datos 7

1. Introducción

[Construcción de software sencillo, pero útil para alumnado]

* 1. Propósito

[Facilitar la información por medio de un sitio]

* 1. Alcance

[Nivel: Académico tanto alumnado como docentes]

* 1. Definiciones, siglas y abreviaturas.

[Simple user UdeG]

* 1. Referencias

[siiau   
http://siiauescolar.siiau.udg.mx/wus/gupprincipal.inicio]

* 1. Visión general

[Replantear el manejo de la información académica para optimizarla como herramienta]

1. Diseño de Casos de Uso

[Sistema de ingreso de datos  
Login->Registro->Consulta->Resultado y datos.]

* 1. Diseño del Caso de Uso [nombre del caso de uso 1]
     1. Diagrama de paquetes

[cache,buffer y cookies del sitio ]

* + 1. Diagrama de Interacción

[-]

* + 1. Diseño de Flujo de eventos

[-]

* + 1. Requerimientos especiales o de implementación

[Subir el sitio y mantenerlo funcional en un servidor]

* 1. Diseño del Caso de Uso [nombre del caso de uso 2]

1. Diseño de Objetos
   1. [Objeto 1]

[Maquetar el proyecto en html y añadir estetica con css.]

* 1. [Objeto 2]

...

1. Diseño de Subsistemas
   1. Subsistemas Específicos
      1. [sistema UdeG]

**Propósito**

[Facilitar la consulta de información a docente y académicos ]

**Función**

[Brindar información rápidamente por medio de consultas]

**Subordinados**

[-]

**Dependencias**

[Depende de un servidor para mantener el sitio y un administrador de base de datos que brinde soporte]

|  |  |  |
| --- | --- | --- |
| Subsistema del que depende | Naturaleza de interacción | Características |
| [Servidor] | [Usuario.->Webservice] | [El usuario realiza consultas sobre su situación académica ] |

**Recursos**

[Computadora ( no importan los requisitos, solo se necesita para editar código)  
Servidor  
Personal de mantenimiento ]

**Interfases**

[Interfaz disponible en todo tipo de dispositivo (computadoras,consolas,celulares,pantallas smart]

|  |  |  |
| --- | --- | --- |
| Interacción | Método de interacción | Reglas de la interacción |
| [Usuario->Servidor] | Consulta de datos | El usuario se logeara a la plataforma y consumira datos |

* + 1. [-]

...

* 1. Subsistemas de Soporte
     1. [-]

**Propósito**

**Función**

**Subordinados**

**Dependencias**

**Recursos**

**Interfases**

* + 1. [Subsistema de soporte 2]

...

1. Revisión de la Interfaz de Usuario

[El usuario será enviado a distintas paginas previamente maquetadas dependiendo de la consulta o acción que realice ]

* 1. Formatos individuales de interfaz de pantalla

[Menú responsivo disponible adaptable para todo dispositivo(hecho en mediaquery adaptado al % de cada pantalla]

* 1. Controles y elementos de diseño de interfaz de pantalla

[dispositivos de entrada como el teclado, touchscreen o algún mando remoto]

* 1. Forma de navegación de interfaz de pantalla

[Entre distintas páginas maquetadas]

* 1. Formatos de impresión

[Formatos png,jpg,word o pdf]

1. Diseño de Datos

[Estructura funcional gracias a scripts]

* 1. Modelo de Datos del Núcleo

[-]

* 1. Modelo de Datos General

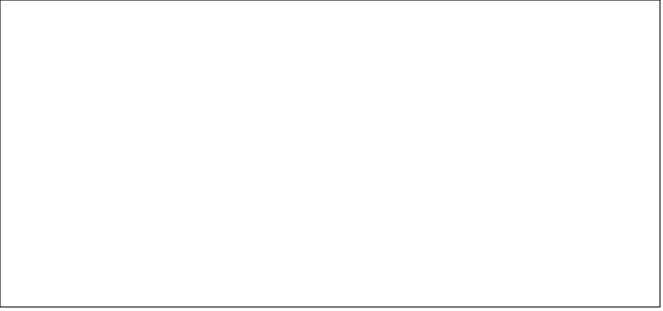
[-]

* 1. Especificación de la Distribución de Datos

[La información se encuentra almacenada en una base de datos en tablas relacionales]

**Paquete de Despliegue**

**Construcción y Pruebas Unitarias**



**Notas:**

Este documento es propiedad intelectual de la organización del autor. De todas formas, la información contenida en el documento es de uso libre.

El uso comercial de este documento está estrictamente prohibido. Este documento es distribuido para mejorar el intercambio de información técnica y científica.

Este material está proporcionado en el estado en que se encuentra. El autor no garantiza ningún tipo, explícito o implícito, de cualquier asunto, sin estar limitado a, garantía o aptitud para propósito o comercialización, exclusividad, o resultados obtenidos del uso del material.

Los procesos descritos en este Paquete de Despliegue no intentan excluir o desalentar el uso de procesos adicionales que las Pequeñas Empresas puedan encontrar útiles.

|  |  |
| --- | --- |
| **Autor Documento Original** | Juan Pablo Padilla Martin |
| **Autor Versión Español** | Juan Pablo Padilla Martin |
| **Editores** | - |
| **Fecha de creación** | 20/11/19 |
| **Fecha de última actualización** | - |
| **Estado** | Versión Final |
| **Versión** | 1.0 |

**Historial de Versiones**

|  |  |  |  |
| --- | --- | --- | --- |
| **Fecha** | **Versión** | **Descripción** | **Autor** |
| 1/11/2019 | 0 | Inicio del proyecto,login maquetado | Juan Pablo Padilla |
| 20/11/2019 | 1 | Sitio listo para su uso | Juan Pablo Padilla |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Abreviaciones/Acrónimos**

|  |  |
| --- | --- |
| **Abre./Acro.** | **Definiciones** |
| PD | - |
| PO | - |
| POs | - |
| LT | Juan Pablo Padilla Martin |
| AN | // // // // |
| DIS | // // // // |
| PR | // // // // |
| GP | // // // // |

**Tabla de Contenidos**

[***1. Descripción Técnica5***](bookmark://_Toc372239427)

[*Propósito de este documento5*](bookmark://_Toc372239428)

[*¿Por qué es importante la Construcción y las Pruebas Unitarias?5*](bookmark://_Toc372239429)

[***2. Definiciones7***](bookmark://_Toc372239430)

[*Términos Genéricos7*](bookmark://_Toc372239431)

[*Términos Específicos7*](bookmark://_Toc372239432)

[***3. Relación con la norma ISO/IEC 291108***](bookmark://_Toc372239433)

[***4. Descripción de Procesos, Actividades, Tareas, Pasos, Roles y Productos10***](bookmark://_Toc372239434)

[Sub-tarea: Definir estándares de construcción10](bookmark://_Toc372239435)

[Sub-tarea: Reportar la taxonomía de defectos de fases previas12](bookmark://_Toc372239436)

[*Asignar tareas a los miembros del Equipo de Trabajo15*](bookmark://_Toc372239437)

[*Construir o actualizar componentes de software20*](bookmark://_Toc372239438)

[*Diseñar o actualizar los casos de prueba y aplicarlos23*](bookmark://_Toc372239439)

[*Corregir los defectos29*](bookmark://_Toc372239440)

[*Actualizar el Registro de Trazabilidad31*](bookmark://_Toc372239441)

[*Descripción de Roles31*](bookmark://_Toc372239442)

[*Descripción de Productos32*](bookmark://_Toc372239443)

[*Descripción de Artefactos35*](bookmark://_Toc372239444)

[***5. Plantillas37***](bookmark://_Toc372239445)

[*5.1 Plantilla de construcción para Java(en su caso)37*](bookmark://_Toc372239446)

[***6. Ejemplo39***](bookmark://_Toc372239447)

[*6.1 Ejemplo de un Estándar de Construcción General39*](bookmark://_Toc372239448)

[*Formato39*](bookmark://_Toc372239449)

[*6.2 Ejemplo de pseudocódigo42*](bookmark://_Toc372239450)

[Especificación42](bookmark://_Toc372239451)

[El Pseudocódigo basado en la especificación previa43](bookmark://_Toc372239452)

[*6.3 Ejemplo de diagrama de flujo44*](bookmark://_Toc372239453)

[*6.4 Jerarquía de fuerzas de criterios de cobertura desde la más débil hacia la más fuerte45*](bookmark://_Toc372239454)

[Casos de prueba para cobertura de Decisión y Condición46](bookmark://_Toc372239455)

[*6.5 Diseño de caso de prueba, escritura de prueba unitaria y uso de herramienta de cobertura de código.47*](bookmark://_Toc372239456)

[Cobertura de Sentencias47](bookmark://_Toc372239457)

[Cobertura Rama/Decisión48](bookmark://_Toc372239458)

[Cobertura de Decisión y Condición49](bookmark://_Toc372239459)

[*6.6 Pruebas unitarias siguiendo Técnicas de Pruebas Estructuradas y mapeando los pasos de la tarea "Diseñar o actualizar casos de pruebas unitarias y aplicarlos".51*](bookmark://_Toc372239460)

[*6.7 Corrección de defectos55*](bookmark://_Toc372239461)

[*6.8 Ejemplos de ejecutores de pruebas56*](bookmark://_Toc372239462)

[*6.9 Ejemplo de macro de aserción en C++57*](bookmark://_Toc372239463)

[*6.10 Ciclo de vida de las pruebas unitarias58*](bookmark://_Toc372239464)

[***7. Listas de Verificación59***](bookmark://_Toc372239465)

[*Listas de Verificación de Tareas59*](bookmark://_Toc372239466)

[*7.1 Asignar tareas a los miembros del Equipo de Trabajo59*](bookmark://_Toc372239467)

[*7.2 Construir o actualizar los Componentes de Software59*](bookmark://_Toc372239468)

[*7.3 Diseñar o actualizar los casos de pruebas unitarias y aplicarlas59*](bookmark://_Toc372239469)

[*7.4 Corregir los defectos59*](bookmark://_Toc372239470)

[*Listas de Verificación de Soporte60*](bookmark://_Toc372239471)

[*7.5 Lista de verificación de revisión de código60*](bookmark://_Toc372239472)

[*7.6 Lo que deberían proveer el arquitecto y el diseñador61*](bookmark://_Toc372239473)

[Requerimientos61](bookmark://_Toc372239474)

[Arquitectura y Diseño61](bookmark://_Toc372239475)

[*7.7 Sub-tarea: Seleccionar el estándar de interfaz de usuario61*](bookmark://_Toc372239476)

[***8. Herramientas63***](bookmark://_Toc372239477)

[*8.1 Matriz de Trazabilidad63*](bookmark://_Toc372239478)

[*8.2 Herramientas de Cobertura de Código64*](bookmark://_Toc372239479)

[*8.3 Frameworks de Pruebas Unitarias65*](bookmark://_Toc372239480)

[***9. Referencias a Otros Estándares y Modelos66***](bookmark://_Toc372239481)

[*Matriz de Referencia ISO 900166*](bookmark://_Toc372239482)

[*Matriz de Referencia ISO/IEC 1220768*](bookmark://_Toc372239483)

[*Matriz de Referencia CMMI69*](bookmark://_Toc372239484)

[***10. Referencias72***](bookmark://_Toc372239485)

[***11. Formulario de Evaluación74***](bookmark://_Toc372239486)

￼

# **1. Descripción Técnica**

## **Propósito de este documento**

*[Es un proyecto que mejora una herramienta que ya posee el alumnado optimizándola para ser mejor.]*

*“El propósito de este documento, es permitir definir los lineamientos a seguir para realizar la planeación de la etapa de integración del software y de pruebas sobre el proyecto “nombre del proyecto”, planteando una estrategia que conduzca al objetivo enfocado en el aseguramiento de calidad del software.”*

Un Paquete de Despliegue (PD) es un conjunto de artefactos desarrollados para facilitar la implementación de un conjunto de prácticas en una Pequeña Organización (PO). Los elementos de un PD típico son: descripción de procesos, actividades, tareas, roles y productos, plantillas, lista de verificación, ejemplo y referencia a estándares y modelos, y herramientas.

## **¿Por qué es importante la Construcción y las Pruebas Unitarias?**

*[Se pone a prueba el algoritmo introduciendo datos tanto correctos como erroneos para evaluar el comportamiento del software y corregir errores]*

“A continuación se listan los elementos (artefactos, entregables, documentos, etc.) que serán objeto de prueba dentro del esfuerzo de pruebas:”

* Funcionalidad especificada en el documento de especificación de requisitos.
* Manejo de los datos y transacciones involucradas en las funcionalidades del punto anterior.
* Rendimiento al ejecutar las funcionalidades del primer punto.

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

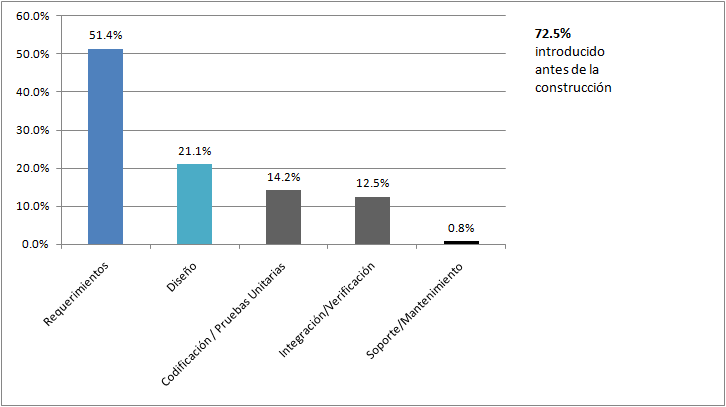
* Proyectos reales usualmente omiten los requerimientos y el diseño. Pero sin importar la prisa del proyecto, **no se puede evitar la construcción**. Así, mejorar la construcción es una manera de mejorar cualquier esfuerzo de desarrollo de software, sin importar lo breve que sea.
* **En muchos proyectos, la única documentación disponible a los programadores es el código mismo**. Las especificaciones de los requerimientos y los documentos de diseño pueden estar fuera de fecha, pero el código fuente siempre estará actualizado a la fecha.
* ***80% del costo de desarrollo*** *es consumido por programadores de software identificando y corrigiendo defectos****1***.

Tan pronto como el programador desarrolla una unidad de código, el siguiente paso es evaluar si funciona correctamente. Las pruebas unitarias son la solución para este trabajo debido a que es el primer nivel de pruebas, y tal como se muestra en la Tabla 1 mientras más tarde se encuentren los defectos, se tiene un mayor costo.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ***Tiempo Introducido*** | ***Tiempo Detectado*** |  |  |  |  |
|  | ***Requerimientos*** | ***Arquitectura*** | ***Construcción*** | ***Pruebas del Sistema*** | ***Post-Entrega*** |
| ***Requerimientos*** | 1 | 3 | 5-10 | 10 | 10-100 |
| ***Arquitectura*** | --- | 1 | 10 | 15 | 25-100 |
| ***Construcción*** | --- | --- | 1 | 10 | 10-25 |

**Tabla 1. Costo promedio proporcional de arreglar defectos basado en el momento en que fueron introducidos y detectados. [Código Completo]**

Como se muestra en el gráfico 1, más del 70% de los defectos se introducen antes de la construcción, y otro 14% se introducen en esta etapa2, así que si no se toman acciones en este punto, el costo en etapas posteriores incrementará significativamente.



**Gráfico 1. Orígenes de los defectos de software (Selby 2007). Lo adecuado**

# **2. Definiciones**

*[-]*

En esta sección, el lector encontrará dos conjuntos de definiciones. El primer conjunto define los términos utilizados en todos los Paquetes de Despliegue, Patrones de integración, Integración de Datos y Pruebas, esto es, términos genéricos. El segundo conjunto de términos utilizados en este Paquete de Despliegue, es decir, los términos específicos.

## **Términos Genéricos**

***Proceso:*** conjunto de actividades interrelacionadas o que interactúan entre ellas para transformar 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 uno o más resultados de un proceso.[ISO/IEC 12207]

***Sub-Tarea:*** cuando una tarea es compleja, se divide en sub-tareas.

***Paso:*** en un paquete de despliegue, una tarea es descompuesta en una serie de pasos.

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

***Producto:*** pieza de información o entregable que puede ser producida (no obligatoriamente) por una o muchas tareas (por ejemplo, un documento de diseño, código fuente).

***Artefacto:*** información, que puede no estar listada en la norma ISO/IEC 29110 Parte 5, pero que puede ayudar a una PO durante la ejecución del proyecto.

## **Términos Específicos**

**Componente:** Conjunto de servicios funcionales en el software que, cuando se implementa, representa un conjunto de funciones bien definido y es distinguible por un nombre único[ISO/IEC 29881:2008].

**Defecto:** Un problema que, si no es corregido, puede causar fallos o producir resultados incorrectos en una aplicación [ISO/IEC 20926].

**Trazabilidad:** Grado que existe en la relación que se puede establecer entre dos o más productos del proceso de desarrollo, especialmente productos que poseen una relación predecesor - sucesor o maestro - subordinado entre ellos. [IEEE 1233-1998]

**Prueba unitaria:** Prueba de rutinas y módulos individuales por el desarrollador o un tester independiente. [ISO/IEC 24765]

**Cobertura de código:** Medida utilizada en las pruebas de software. Describe el grado en que se ha probado el código fuente de un programa. [Practical Software Testing]

# **3. Relación con la norma ISO/IEC 29110**

Este paquete de despliegue cubre las actividades relacionadas a la Construcción y Pruebas Unitarias del Reporte Técnico ISO/IEC 29110 Parte 5-1-2 para Pequeñas Organizaciones (POs) – Perfil Básico [ISO/IEC29110].

Las actividades de construcción deberán haber sido planeadas durante la actividad de Planificación de Proyecto. Las actividades de construcción deberán ser descritas en el Plan de Proyecto. Si este no es el caso, el Gestor de Proyecto deberá ejecutar esta actividad antes de iniciar la construcción (ver el Paquete de Despliegue de Gestión de Proyecto).

En esta sección, el lector encontrará una lista de procesos, actividades, tareas y roles de Implementación de Software (IS) de la Parte 5 que están directamente relacionados a este tema. Este tema está descrito en detalles en la siguiente sección.

* **Proceso:** IS - Implementación de Software
  + **Actividad:** IS.23 Análisis de Requerimientos de Software
    - Tareas y Roles:

|  |  |
| --- | --- |
| **Tareas** | **Roles**4 |
| IS.2.2 Documentar o actualizar la *Especificación de Requerimientos*.  Identificar y consultar fuentes de información (*Cliente*, usuarios, sistemas previos, documentos previos, etc.) con el propósito de obtener nuevos requerimientos.  Analizar los requerimientos identificados para determinar el alcance y la viabilidad.  Generar o actualizar la *Especificación de Requerimientos*. | AN,  CLI |

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

|  |  |
| --- | --- |
| **Tareas** | **Roles** |
| IS.4.1 Asignar tareas a los miembros del Equipo de Trabajo en relación a su rol, de acuerdo al *Plan de Proyecto* actual | LT,  PR |
| IS.4.3 Construir o actualizar los *Componentes de Software* basados en la parte detallada del *Diseño de Software* | PR |
| IS.4.4 Diseñar o actualizar los *Casos de Pruebas* unitarias y aplicarlos para verificar que los *Componentes de Software* implementan la parte detallada de *Diseño de Software* | PR |
| IS.4.5 Corregir los defectos encontrados hasta lograr la prueba unitaria exitosa (logrado el criterio de salida) | PR |
| IS.4.6 Actualizar el *Registro de Trazabilidad* incorporando Componentes de Software construidos o modificados | PR |

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

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

|  |  |
| --- | --- |
| **Tareas** | **Roles5** |
| IS.2.2 Documentar o actualizar la *Especificación de Requerimientos*.  Identificar y consultar fuentes de información (*Cliente*, usuarios, sistemas previos, documentos previos, etc.) con el propósito de obtener nuevos requerimientos.  Analizar los requerimientos identificados para determinar el alcance y la viabilidad.  Generar o actualizar la *Especificación de Requerimientos*. | AN ,  CLI |

Esta tarea está relacionada con las siguientes sub-tareas:

* Definir estándares de construcción
* Reportar la taxonomía de defectos de fases previas

### **Sub-tarea: Definir estándares de construcción**

|  |  |
| --- | --- |
|  |  |
| ***Objetivos:*** | Proveer una guía en la codificación de software para producir software de fácil mantenimiento dentro o fuera del proyecto. |
| ***Razón Fundamental:*** | Los requerimientos de mantenibilidad son directamente atendidos por esta sub-tarea. Estos requerimientos deberán haber sido aceptados como parte de la Especificación de Requerimientos, sin embargo la mayoría de las veces no están mencionados explícitamente, pero siempre son esperados.  La carencia de estándares de codificación será percibida por los colegas cuando intenten modificar el código, dentro del equipo y fuera de él. Algunos Componentes podrían ser reemplazados por nuevos debido a la incapacidad de su entendimiento, si el mantenimiento se realiza por el equipo de desarrollo, el costo del proyecto incrementará, si se realiza por el cliente entonces ellos absorberán este costo.  Los Estándares de Codificación deberán ser utilizados al menos en aquellos componentes que ejecutan las funcionalidades claves.  Los Estándares de Codificación no son directamente atendidos por 29110-5, sin embargo la inversión de esfuerzo puede ayudar a incrementar la productividad del proyecto y la calidad del producto.  *Nota: Se provee un estándar general de construcción en la sección de ejemplos* |
| ***Roles:*** | Gestor de Proyecto |
|  | Líder Técnico |
|  | Programador |
| ***Artefactos:*** | Estándares de Construcción |
| ***Pasos:*** | 1. Planificar la sub-tarea. |
|  | 1. Obtener estándares disponibles. |
|  | 1. Seleccionar los estándares. |
|  | 1. Adoptar los estándares. |
|  | 1. Verificar la adopción de los estándares. |
| ***Descripción de Pasos:*** | ***Paso 1. Planificar la sub-tarea***  De acuerdo al progreso del proyecto, el Gestor de Proyecto incluye estos pasos en el Plan de Proyecto.  Asignar el esfuerzo es muy importante porque la definición de estándares puede ser interminable, y si los estándares no son adoptados, entonces serán inútiles.  Respecto al cronograma, es deseable tener los estándares listos antes de empezar la construcción. Sin embargo esto no siempre es posible.  ***Paso 2. Obtener estándares disponibles***  Averiguar si su cliente tiene estándares de construcción, de lo contrario buscarlos en internet o en otra fuente disponible.  Evitar definirlos desde cero, en la mayoría de proyectos está fuera del alcance, crearlos puede tomar mucho esfuerzo y tiempo.  ***Paso 3. Seleccionar los estándares***  Si su cliente no posee estándares entonces pregunte a los programadores para elegir uno de aquellos que fueron encontrados o la combinación de ellos.  Respecto a la documentación del estándar, la manera más fácil es implementar algunos componentes y utilizarlos como ejemplos, si posee tiempo suficiente, entonces cree los estándares de construcción.  *Nota 1: Se provee un ejemplo de estándar general de codificación en la sección de ejemplos.*  *Nota 2: Se provee una plantilla de construcción en Java en la sección de plantillas.*  ***Paso 4. Adoptar los estándares***  Pedir a los programadores para adoptar los estándares desde este momento, especialmente en aquellos componentes que realizan funcionalidades claves.  ***Paso 5. Verificar la adopción de los estándares***  Crear los componentes que realizan funcionalidades claves para ser verificados con respecto a la adopción de estándares de construcción por otro programador. |

### **Sub-tarea: Reportar la taxonomía de defectos de fases previas**

|  |  |
| --- | --- |
|  |  |
| ***Objetivos:*** | Reportar los defectos de las etapas previas al área encargada. |
| ***Razón Fundamental:*** | Tal como se muestra en el gráfico 1, más del 70% de los defectos son introducidos antes de la construcción. Existe una gran probabilidad de encontrar defectos en las etapas previas a la fase de construcción. La mejor estrategia es reportar los defectos al área encargada antes de que sus correcciones se vuelvan más costosas.  Esta tarea no está dirigida por la ISO/IEC 29110-5, sin embargo la inversión de un esfuerzo puede ayudar a incrementar la productividad del proyecto y la calidad del producto. |
| ***Roles:*** | Líder Técnico |
|  | Programador |
| ***Artefactos:*** | Taxonomía de Defectos [Actualizado] |
| ***Pasos:***  ***(Programador)*** | 1. Confirmar el defecto |
|  | 1. Escribir una breve descripción del defecto |
|  | 1. Escribir el área donde fue encontrado el defecto |
|  | 1. Escribir las causas posibles |
|  | 1. Escribir la extensión de los daños |
| ***Pasos:***  ***(Líder Técnico)*** | 1. Verificar el reporte de defectos |
|  | 1. Escribir una estrategia de mitigación |
|  | 1. Reportar el defecto al área encargada |
| ***Descripción de Pasos:***  ***(Programador)*** | ***Paso 1.* Confirmar el defecto**  Si se encuentran inconsistencias en el Diseño Detallado, la Arquitectura de Software o una parte de los Requerimientos, se debe explicar los detalles al líder técnico.  Si el líder técnico concuerda con que se ha encontrado un defecto, continuar con los pasos siguientes. De lo contrario, omitir todas las sub-tareas. Pero, recordar que el no reportar un defecto en el momento adecuado significa que luego de codificarlo, se deberá cambiar en algún momento.  ***Paso 2.* Escribir una breve descripción del defecto**  Escribir una breve descripción del defecto, incluyendo cómo se encontró y las condiciones y circunstancias donde se encontró.  ***Paso 3.* Escribir el área donde fue encontrado el defecto**  Escribir la etapa del ciclo de vida donde se encontró el defecto   * Análisis de Requerimientos * Arquitectura de Software * Diseño Detallado   ***Paso 4.* Escribir las causas posibles**  Escribir el propósito sirve para proveer indicaciones de la causa raíz de un defecto:   * **Factores sistemáticos:** guías y procedimientos; cultura de la compañía; información específica del dominio, como documentación, código, herramientas, etc. * **Factores humanos:** una persona cometió un error por razones humanas: omisión, mala aplicación, no miró, no encontró, solución incorrecta, etc. * **Factores desconocidos**   ***Paso 5.* Escribir la extensión de los daños**  Escribir cómo se extienden los efectos del defecto   * Función * Objeto * Proceso * Compatibilidad * Aplicación * Máquina * Servidor * Clientes * Red * Otros |
| ***Descripción de Pasos:***  ***(Líder Técnico)*** | ***Paso 1.* Verificar el reporte de defectos**  Verificar el reporte hecho por el programador. Si existen inconsistencias, preguntar al programador para clarificar los detalles.  ***Paso 2.* Escribir una estrategia de mitigación**  Si se conoce una estrategia de mitigación para el defecto encontrado, agregarla al reporte. Algunas estrategias de mitigación pueden ser:  **Tipo de Mitigación:**   * Nueva herramienta * Hardware * Entrenamiento * Gestión de personal * Comunicación (grupos/individuos) * Acceso al conocimiento * Cambio de proceso   ***Paso 3.* Reportar el defecto al área encargada**  Una vez que el proceso esté completo, enviarlo al área encargada y si es posible, reasignar las tareas para el programador que encontró el defecto mientras que el área relacionada responde la solicitud. |

* **Proceso:** IS -Implementación de Software
  + **Actividad:** IS.4 Construcción de Software
    - Tareas and Roles:

|  |  |
| --- | --- |
| **Tareas** | **Roles6** |
| IS.4.1 Asignar *Tareas* a los miembros del Equipo de Trabajo en relación a su rol, de acuerdo al *Plan de Proyecto* actual | LT,  PR |
| IS.4.3 Construir o actualizar los *Componentes de Software* basados en la parte detallada del *Diseño de Software* | PR |
| IS.4.4 Diseñar o actualizar los *Casos de Pruebas* unitarias y aplicarlos para verificar que los *Componentes de Software* implementan la parte detallada de *Diseño de Software* | PR |
| IS.4.5 Corregir los defectos encontrados hasta lograr la prueba unitaria exitosa (logrado el criterio de salida) | PR |
| IS.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 asociadas a su rol, de acuerdo al Plan de Proyecto.

|  |  |
| --- | --- |
|  |  |
| ***Objetivos:*** | Definir la secuencia de construcción y asignar tareas a los miembros del Equipo de Trabajo |
| ***Razón Fundamental:*** | La mayoría de veces, la primera versión del Plan de Proyecto ha identificado la mayoría de componentes que serán construidos. Sin embargo, en esta etapa del proyecto, cuando la Arquitectura de Software y el Diseño Detallado han sido completados, el Plan de Proyecto debe actualizarse para incluir la construcción y pruebas unitarias, en detalle o en rasgos generales, de todos los componentes a producir.  La secuencia de construcción de componentes deberá ser coordinada con la secuencia de integración para tener los componentes (probados unitariamente) listos para ser integrados en el momento adecuado.  Definir la secuencia de construcción no está incluido en la ISO/IEC 29110 Parte 5, sin embargo invertir un esfuerzo para definir la secuencia de construcción puede ayudar a optimizar el cronograma de construcción. |
| ***Roles:*** | Líder Técnico |
| ***Productos:*** | Plan de Proyecto |
| ***Pasos:*** | 1. Obtener la documentación del diseño de software. |
|  | 2. Seleccionar la estrategia de secuencia de integración. |
|  | 3. Detallar el cronograma del proyecto. |
|  | 4. Designar tareas a los miembros del Equipo de Trabajo. |
|  | 5. Definir criterios 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 de Proyecto. El Diseño Detallado de Software a Bajo Nivel incluye los detalles de los componentes de software por construir. * Obtener el Registro de Trazabilidad del Repositorio de Proyecto.   ***Paso 2. Seleccionar la estrategia de secuencia de integración***  Existen diversas estrategias para determinar la secuencia de integración, pero son más heurísticas que algoritmos.  Los enfoques de integración más comunes son:   * Big bang: integrar todas las partes inmediatamente, donde el software completo se ensambla y se prueba en un solo paso (este enfoque puede ser riesgoso debido al alto nivel de entropía introducido). * Bottom-up: Los módulos en los niveles más bajos se integran primero, luego se escala hacia la siguiente estructura de control. El proceso se repite hasta que se integre el componente en la cima de la jerarquía. * Top-down: Los módulos de más alto nivel se integran primero a través de la estructura de control. El proceso se repite hasta que se integran los componentes de más bajo nivel en la jerarquía.   *Nota: Implementar puramente la integración Bottom-up o Top-down es radical algunas veces, así que una mejor elección es utilizar un enfoque híbrido.*  Algunos enfoques híbridos son:  **Integración orientada a riesgos**  La integración orientada a riesgos tiende a integrar los componentes que se encuentran en la cima (clases de objetos de negocio de alto nivel) y de menor nivel (clases utilitarias e interfaces a dispositivos) primero, dejando los componentes intermedios para el final. Se prioriza de acuerdo al nivel de riesgo asociado con cada componente.    **Figura 1 Estrategia de integración orientada a riesgos. [Código completo]**  **Integración orientada a características**  La idea es integrar los componentes en grupos que los identifique por sus características. Cada característica integrada brinda un incremento de funcionalidad. Una ventaja de esta estrategia es que fácilmente encaja con el diseño orientado a objetos.    **Figura 2 Estrategia de integración orientada a características [Código completo]**  Se puede utilizar un solo enfoque de integración o una combinación de dos o más enfoques y evaluarlo contra el proyecto específico.  ***Paso 3. Detallar el cronograma del proyecto***  Detallar el cronograma de proyecto incluyendo las actividades para desarrollar o modificar todos los componentes identificados de acuerdo con la estrategia seleccionada.  Hacer lo mejor para mantener el tiempo y costo del proyecto acordados. Si es necesario modificarlos, entonces se debe iniciar una Solicitud de Cambio. Para más detalles del cronograma de proyecto, dirigirse al Paquete de Despliegue de Gestión de Proyectos.  ***Paso 4. Designar tareas a los miembros del Equipo de Trabajo***   * Designar las tareas a los miembros del Equipo de Trabajo   + Designar tareas para codificar componentes de software   + Designar tareas para desarrollar casos de prueba y pruebas de los componentes de software * Informar a los miembros acerca de sus tareas   Para designar tareas al personal adecuado, existen algunos aspectos de los programadores que deberán considerarse. Siempre preguntar por los antecedentes del equipo:   * Experiencia * Habilidades * Conocimiento   Esto ayudará a tener al personal adecuado en la actividad correcta. También se debe tener en cuenta estas consideraciones:   * Siempre intentar asignar las tareas y los componentes más críticos al personal más experimentado. * Dejar los componentes básicos a los programadores principiantes; luego ellos se convertirán en programadores expertos. * Intentar identificar las habilidades de cada programador y tomar ventajas de ello.   ***Paso 5. Definir criterios de salida para las pruebas unitarias***  Probar todos los casos posibles de un componente no es factible incluso para proyectos pequeños. Por lo tanto, se deberá decidir un criterio de éxito que indique a los programadores cuando ellos deberán detener las pruebas o cuando existan las pruebas suficientes para un tipo de componente.  Un buen inicio es definir el criterio de porcentaje de cobertura de código (vea el Paso 1 de la tarea “Designar o actualizar casos de prueba y aplicarlos”) dependiendo en cuán riesgoso sea el código a probar. Se puede dividir en tres categorías:   |  |  | | --- | --- | | **Tipo de Componente** | **Descripción** | | Alto riesgo | El código que podría causar daños severos (eliminar datos, perjudicar a alguien, dar respuestas equivocadas no muy obvias que pueden costar mucho dinero al usuario), o que tiene muchos usuarios (así el costo de una falla pequeña se multiplica), o parece que tendrá muchos errores cuyos costos se acumularán (un algoritmo difícil, o se comunica con una interfaz mal definida y pobremente entendida, o haber encontrado un número inusual de problemas). | | Bajo riesgo | El código es improbable de tener fallas lo suficientemente importantes como para retrasar o detener una entrega, inclusive cuando las fallas son sumarizadas. Estas serían fallas molestas en características de baja importancia, algunas con solución simple y obvia. | | Medio riesgo | El código se encuentra entre ambas categorías. Las fallas no serían críticas individualmente, pero tener muchas de ellas causaría un retraso en el cronograma. Existe una buena razón para encontrarlas y arreglarlas a la brevedad - y al menor costo - posible. Pero existen retornos menores - tiempo utilizado para un trabajo exhaustivo que podría ser empleado en otras tareas. |   **Tabla 2 Categorías de código bajo pruebas. [Mal uso de la cobertura de código]**  No existe un algoritmo que pueda decir a qué categoría corresponde un componente. En realidad, un código de riesgo medio puede ser de alto riesgo respecto a ciertos tipos de fallas y lo mismo podría pasar con código de bajo riesgo. Se deberá evaluar el impacto de cada componente y seleccionarlo apropiadamente.  Una vez que se haya definido el riesgo de cada componente, es adecuado utilizar criterios más exhaustivos para probar los componentes de mayor riesgo.   |  |  | | --- | --- | | **Tipo de componente** | **Criterio recomendado de cobertura** | | Bajo riesgo | Sentencia | | Medio riesgo | Rama/Decisión | | Alto riesgo | Decisión y condición |   **Tabla 3 Ejemplo de criterio de cobertura por tipo de componente**  Un porcentaje común seleccionado por algunas compañías respetables es 85% de cobertura de código (Marick 1999). Sin embargo, este no es el porcentaje correcto puesto que cada proyecto posee diferentes restricciones. El porcentaje de cobertura elegido depende de muchas circunstancias como la importancia del proyecto, el cronograma y los recursos disponibles.  Alcanzar el 100% de cobertura de código es generalmente muy difícil y algunas veces no es rentable. Sin embargo, alcanzar un alto porcentaje puede dar cierta confianza en que el código trabaja apropiadamente incluso si no ha sido probado completamente. |

## **Construir o actualizar componentes de software**

#### *Nota: Los componentes se basan en la parte detallada del diseño de software*

|  |  |
| --- | --- |
|  |  |
| ***Objetivos:*** | Producir los Componentes de Software tal como fueron diseñados. |
| ***Razón Fundamental:*** | La calidad de la construcción afecta sustancialmente la calidad del software y la mejor manera de hacerlo es mejorando las prácticas y técnicas del desarrollador.  Los programadores pueden sentir suficiente confianza para producir componentes sin un enfoque sistemático; sin embargo, aún resulta útil para construir componentes complejos o críticos.  Existen diversos enfoques para producir componentes, aquí se usan los enfoques de Pseudocódigo y el Diagrama de Flujo que son dos de los más comunes aceptados. Los pasos fueron adaptados de Code Complete (ver referencia).  *Nota: Algunas veces existen inconsistencias respecto a las etapas anteriores (Requerimientos, Diseño Detallado). Si este es el caso, la mejor opción es reportarlas antes de construir algo inadecuado. (Ver subtarea “Reportar la taxonomía de defectos de fases previas”).* |
| ***Roles:*** | Programador |
| ***Productos:*** | Componentes de Software |
| ***Pasos:*** | 1. Entender el diseño detallado de los componentes y su distribución. |
|  | 1. Buscar funcionalidades disponibles en las librerías estándares. |
|  | 1. Definir la lógica del componente |
|  | 1. Codificar el componente de acuerdo al estándar de construcción. |
|  | 1. Verificar el componente |
| ***Descripción de Pasos:*** | ***Paso 1. Entender el diseño detallado de los componentes y su distribución***   1. **Verificar la contribución**   Verificar la contribución del componente y ver si la mejor manera de resolver el problema es la creación de un nuevo componente o la adaptación de un componente previo.   1. **Detalle Suficiente**   Si el Diseño Detallado de Software no es suficientemente claro para el programador, deberá completarse con el apoyo del Diseñador.  El Diseño Detallado de Software deberá incluir detalles de los Componentes para facilitar su construcción y pruebas dentro del ambiente de programación:   * Proveer diseño detallado (diagrama de clases, diagrama de actividades, diagrama de entidad relación, etc.) * Proveer formatos de entrada / salida de datos * Proveer especificaciones de las necesidades de almacenamiento de datos * Definir el formato de las estructuras de datos requeridas * Definir los campos de datos y el propósito de cada elemento de dato   *Nota: Si se encuentra alguna inconsistencia en el Diseño Detallado, comunicarlo al Líder Técnico. Ver “Reportar la taxonomía de defectos de fases previas”.*  ***Paso 2. Buscar funcionalidades disponibles en las librerías estándares***  Buscar si la funcionalidad de algunos o todos los componentes podría estar disponible en una librería del lenguaje, plataforma o herramientas que se utilizan. Mejorar la productividad reutilizando código. Muchos algoritmos ya han sido creados, probados y mejorados, así que no reinventar la rueda, solo usarlos.  ***Paso 3. Definir la lógica del componente***  Existen diversos enfoques para producir componentes, acá se utilizan Pseudocódigo y Diagrama de Flujo, que son dos de los ampliamente más aceptados. Los pasos fueron adaptados de Code Complete (ver referencia).   1. **Proceso de programación por pseudocódigo**   Una de las mejores formas para abstraer una idea es hacer un dibujo simple de ella a alto nivel. Luego, refinar la idea y empezar escribiendo algunas líneas de Pseudocódigo que describan una posible solución a nivel intermedio (en su lenguaje natural), luego continuar refinando el Pseudocódigo hasta que se alcance un diseño de bajo nivel (muy cercado al código fuente). Así, se facilita la conversión Peudocódigo a código fuente.  Siempre intentar tantas ideas como se puedan en Pseudocódigo antes de empezar a codificar. Una vez que se empiece a codificar, se involucran sentimientos con el código y se vuelve más difícil de eliminar un mal diseño y empezar nuevamente.  *Nota: Se provee un ejemplo de Pseudocódigo en la sección de ejemplos.*   1. **Diagrama de Flujo**   Otra forma de abstraer una idea es haciendo un gráfico o una representación simbólica del algoritmo que se tenga en mente. Cada paso en el algoritmo se representa por un símbolo y contiene una descripción corta de cada paso del proceso. Los símbolos de diagrama de flujo están vinculados por flechas que muestran la dirección del flujo del algoritmo.  Se puede iniciar por definir un estado inicial y luego agregar pasos, ramas o bucles para procesar hasta que se alcance el final del algoritmo. Entonces se puede refinar cada paso tal como sea necesario hasta convertirlo en código fuente que pueda ser fácil.  *Nota: Se provee un ejemplo de diagrama de flujo en esta sección.*  ***Paso 4. Codificar el componente de acuerdo al estándar de construcción***  La codificación deberá ser un proceso mecánico una vez que el algoritmo esté listo. Se debe enfocar en los problemas de configuración y del lenguaje más que en la lógica del componente.  Algunos de los trabajos previos puede ser reusable. Por ejemplo, si se está usando el enfoque de Pseudocódigo, se puede usar el Pseudocódigo de *alto nivel* como comentario y usar el Pseudocódigo de *bajo nivel* para codificar la lógica del componente.  Para este tipo de problemas, es importante seguir un estándar de construcción mientras se codifica, de manera que el código no solo resulte limpio, debe estar estandarizado y ser fácil de mantener. Ver subtarea "Definir estándares de construcción".  Si el componente necesita una interfaz, se deberá codificar de acuerdo al estándar de interfaces Si no existe, solicitarlo al Líder Técnico. Ver "Seleccionar el estándar de interfaces de usuario" en la sección de Listas de Verificación.  **Verificar si el código deberá ser factorizado**  Se puede considerar crear una subrutina si:   * Alguno de los pasos en el proceso es muy repetitivo y en lugar de copiar y pegar mucho código solo llamar a la subrutina adecuada que realiza el trabajo. * Algunos de los pasos en el algoritmo son parte de un sub-proceso independiente que pueden ser utilizados por otros componentes. Crear una subrutina que no cambie la lógica pero haga el código claro y más mantenible.   ***Paso 5. Verificar el componente***  La revisión de código es una examinación sistemática en el que los desarrolladores de software revisan el código, identifican fallas y conservan el código más mantenible. Se pretende encontrar y corregir defectos pasados por alto en la fase inicial del desarrollo, mejorando ambos la calidad del código cuente y las habilidades del desarrollador.  *Nota: Se provee un ejemplo de una lista de verificación para la revisión de código en la sección de Lista de Verificación.*  **Compilar el código:**  Dejar a la computadora revisar variables no declaradas, conflictos de nombres y otros. |

## **Diseñar o actualizar los casos de prueba y aplicarlos**

|  |  |
| --- | --- |
|  |  |
| ***Objetivos:*** | Encontrar y corregir los defectos introducidos cuando se codifica a través del diseño y aplicación de los casos de prueba unitarios. |
| ***Razón Fundamental:*** | Existen dos enfoques fundamentales para pruebas, llamados Caja Negra y Caja Blanca. Las pruebas de Caja Negra prueban los requerimientos funcionales, sin conocer la estructura interna de los componentes. Las pruebas de Caja Blanca prueban la estructura interna y la lógica del componente.  Estos componentes o unidades son los bloques de construcción más pequeños de software y debido a que es más fácil y menos costoso buscar los defectos en una pequeña unidad que en una parte más larga del sistema, ellos deberán ser realizados tan pronto como las unidades de prueba estén listas.  Incluso si no siempre es posible, se recomienda altamente que los programadores realicen pruebas de Caja Blanca puesto que ellos conocen mejor el código de sus componentes. Siendo prácticos en este paquete, se asume la última sugerencia y el enfoque elegido es pruebas de Caja Blanca.  Un beneficio importante de las pruebas unitarias es que se logra paralelismo, habilitando las pruebas y la depuración en simultáneo por muchos programadores. Con la ayuda de herramientas y frameworks, se puede mejorar la eficiencia de las pruebas.  *Nota: Las pruebas unitarias son útiles para todo tipo de software. Sin embargo, el software implementado utilizando un lenguaje orientado a objetos introduce consideraciones adicionales que deben ser tomadas en cuenta al probar dicho software. Los lectores interesados pueden consultar [Introduction to Software Testing] para una discusión de los problemas asociados con las pruebas unitarias de implementaciones orientadas a objetos.* |
| ***Roles:*** | Programador |
|  | Gestor de Proyecto |
| ***Productos:*** | Componentes de Software [a ser probados unitariamente] |
|  | Casos de Prueba, Conjunto de pruebas unitarias |
| ***Pasos:*** | 1. Obtener los criterios de salida |
|  | 2. Diseñar los casos de prueba |
|  | 3. Codificar las pruebas unitarias |
|  | 4. Ejecutar las pruebas unitarias |
|  | 5. Analizar los resultados |
| ***Descripción de Pasos:*** | ***Paso 1. Obtener los criterios de salida***  Los criterios de salida están compuestos por uno más criterios de cobertura de código7 y un porcentaje a ser alcanzado. Los criterios deberían haber sido definidos previamente por el Gestor de Proyectos.  **Criterios de cobertura**  Los criterios de cobertura (criterios de adecuación para algunos autores) son un framework de Caja Blanca que trabajan como una regla de detención para determinar si se han creado los suficientes casos de prueba.  El criterio común de cobertura incluye cobertura de sentencias, rama/decisión y decisión y condición. La cobertura de sentencias se considera como el más débil de estos criterios, en el sentido que es el que requiere menos esfuerzos de prueba para ser alcanzado, pero también es el menos eficiente para mostrar fallas.     1. **Cobertura de sentencias**   Satisfacer este criterio asegura la ejecución de cada línea de código sin importar el camino8. Es el criterio de prueba de cobertura más débil porque los defectos pueden estar escondidos en un camino particular, que no es necesariamente atravesado cuando la única meta es ejecutar cada línea de código.  *Nota: Este es el nivel mínimo de prueba que debe ser cubierto.*   1. **Cobertura de rama/decisión**   Este criterio requiere que cada posible salida de cada decisión sea ejercitada como un todo al menos una vez (no para cada condición individual en un predicado compuesto, es decir, *si(a y b)*, *a y b* como un todo en lugar de satisfacer las condiciones *a* y *b* por separado).  *Nota: Esta puede ser una buena meta a cubrir por los componentes de complejidad media.*   1. **Cobertura de decisión y condición**   Satisfacer este criterio no solo cubre la cobertura de rama/decisión sino asegura que cada condición tome cada posible resultado al menos una vez y la decisión como un todo toma cada posible salida al menos una vez (para cada condición individual contenida como un predicado compuesto, por ejemplo, *si(a y b)*, satisfaciendo las condiciones *a* y *b* por separado).  Nota: Se recomienda para unidades complejas y críticas*.*  Existen criterios más fuertes como: Condiciones múltiples, DCM (Decisión y Condición Modificada), etc. pero ellos están fuera del alcance de este paquete.  **Porcentaje para el criterio de cobertura seleccionado**  Dependiendo de la dificultad o importante del componente, el Gestor de Proyecto deberá escoger el criterio de cobertura más adecuado.  Es decir, un componente de complejidad media: Criterio de cobertura rama/decisión de 85%. Esto significa que una vez que los casos de prueba son diseñados y ejecutados colectivamente, cuando al menos 85% de las ramas están cubiertas, las pruebas estarán "completas" para esta meta.  ***Paso 2. Diseñar los casos de prueba***  Un caso de prueba es un conjunto específico de valores de entrada en los que el programador determinará si el componente está trabajando propiamente, basado en si los resultados obtenidos son los esperados o no.  Cada caso de prueba deberá tener:  ***ID de caso de prueba:*** Para tener control del número de casos de prueba creados.  ***Descripción*:** Describe el propósito de la prueba. La descripción podrá tener información adicional como el resultado verdadero o falso para ciertas decisiones o condiciones o quizás la ruta cubierta si se utiliza un grafo de flujo de control. Esto depende de la estrategia que se utiliza para obtener los casos de prueba.  ***Entradas:***la configuración de los valores de entrada que forman el caso de prueba.  ***Salidas esperadas:***Los resultados esperados para la configuración de entrada.  *Nota: Se necesita un campo extra para indicar si el caso de prueba aprueba o falla, pero se mantiene vacío hasta el paso de ejecución, donde se obtiene esta información.*  ***Aprueba/Falla:*** Usar P o F para indicar si el caso de prueba aprueba o falla.  **Técnica de Pruebas Estructuradas**  Las Pruebas Estructuradas (también conocido como pruebas de ruta base) es una técnica que optimiza el número de casos de prueba necesarios para alcanzar el 100% de cobertura rama/decisión.  *Nota: Se provee un ejemplo completo de Pruebas Estructuradas en la sección de ejemplo.*  ***Paso 3. Codificar las pruebas unitarias***  Una prueba unitaria es un script de prueba (usualmente 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 utilizar un framework de pruebas unitarias para el lenguaje que se utiliza en la implementación. Si no existe un framework para el lenguaje, otra opción es construir un framework de pruebas propio.  ***Sin* un framework externo de pruebas unitarias**  Si no existe un framework de pruebas unitarias para el lenguaje utilizado, se puede escribir las rutinas propias para las pruebas. Para hacer esto, se necesita código que revise si una llamada al componente bajo ciertas condiciones retorna verdadero (si todo trabajo según lo esperado) o falso (si la llamada resulta en una salida inesperada o error).  Estos tipos de rutinas son llamadas aserciones y usualmente toman dos argumentos: una expresión booleana que describe la suposición esperada a ser verdadera, y un mensaje a mostrar si no lo es. La idea es que los scripts de prueba usen estas aserciones para verificar si el caso de prueba puede detectar un defecto. Si el lenguaje no soporta rutinas de aserción directamente, ellas son relativamente fáciles de escribir.  *Nota: Se provee un ejemplo de una macro de aserción en la sección de ejemplo.*  ***Con* un framework externo de pruebas unitarias**  Estas herramientas son dependientes del lenguaje y permiten escribir y ejecutar pruebas unitarias para ver si habilitan la búsqueda de defectos en la unidad bajo pruebas. Casi todas las herramientas tienen su propia suite de aserciones que ayudar a ejecutar las pruebas.  La idea general es escribir scripts que contengan una o más aserciones que reciben una salida esperada y la unidad bajo pruebas con las entradas arregladas, así si cada una de las aserciones falla, se sabe cuál es y se puede comparar el resultado esperado contra el resultado obtenido.  La meta de este tipo de herramienta es que una vez que se hayan escrito un conjunto de unidades de prueba, si existieran cambios en la implementación, al ejecutar nuevamente las pruebas unitarias, se sabe que no existe funcionalidad "corrompida" que se encontraba trabajando previamente.  Una ventaja importante de estas herramientas es que no hay necesidad de remover nada, se mantiene la implementación y las fuentes de pruebas separadas desde el inicio.  *Nota: En la sección de Herramientas, se proveen vínculos a los frameworks de pruebas unitarias más populares para diversos lenguajes.*  ***Paso 4. Ejecutar las pruebas unitarias***  *Nota: Aislar la unidad bajo pruebas tanto como sea posible es importante para evitar comportamiento inadecuado debido a las dependencias entre unidades.*  ***Sin* un framework externo de pruebas unitarias**  Una vez que se hayan codificado las pruebas unitarias, se puede crear un mecanismo para ejecutarlas. Podría ser un programa separado cuyo único propósito es ejecutar las pruebas unitarias, o una función o método dentro de la implementación de código con el mismo objetivo.  ***Con* un framework externo de pruebas unitarias**  Los frameworks de pruebas unitarias usualmente proveen un Ejecutor de Pruebas que básicamente es una forma de aplicación de línea de comandos o una aplicación gráfica que puede ser utilizada para ejecutar las pruebas automatizadas y reportar los resultados.   1. **Ejecutor de Pruebas de Línea de comandos**   Los Ejecutores de Pruebas de Línea de comandos están diseñados para ser utilizados desde una línea de comandos del sistema operativo o desde archivos batch o scripts de shell. Son muy útiles cuando se trabaja de manera remota vía shells remotos o cuando se ejecutan las pruebas desde un script de construcción como "make" o "ant". Si la suite de pruebas es demasiado extensa, se puede dejar ejecutándose por las noches y verificar los resultados en un archivo log al día siguiente.  *Nota: se provee un ejemplo en la sección de ejemplo.*   1. **Ejecutor de Pruebas Gráfico**   Un Ejecutor de Pruebas Gráfico usualmente es una aplicación de escritorio o parte de un IDE9 (cualquiera incorporado o un componente adicional) para ejecutar pruebas que hace su uso fácil y más conveniente. La característica más común de estas aplicaciones es una especia de indicador de progreso en tiempo real. Algunas de ellas también incluyen un contador de ejecución de pruebas fallidas y una barra de progreso de colores que empieza en verde y termina en rojo tan pronto como se encuentre un error o falla.  *Nota: Se provee un ejemplo en la sección de ejemplos.*  ***Paso 5. Analizar los resultados***  Dependiendo cómo se hayan ejecutado las pruebas unitarias, se puede analizar los resultados almacenados en un archivo log o en una ventana de salida si se ejecutaban vía línea de comandos o por simplemente mirar los resultados gráficamente si se utiliza una implementación de la interfaz de usuario.  En cualquier caso, el siguiente paso es determinar si alguna prueba falla. Si este es el caso, se deberá hacer las correcciones relevantes ya sea en el código de implementación o en el código de la prueba mismo, y ejecutar las pruebas nuevamente.  Una vez que todas las pruebas unitarias aprueben, se deberá examinar por los criterios de salida definidos por el Gestor de Proyecto y verificar si se necesitan crear más casos de prueba o si se tienen suficientes.  Es altamente recomendado utilizar una herramienta de *Cobertura de Código* para ver cuánto código se cubre con los casos de prueba, y qué porcentaje se ha alcanzado en el criterio seleccionado. Se listan algunas herramientas populares para cobertura de código en la sección de Herramientas y todas ellas consideran al menos los criterios de cobertura de sentencias y de rama/decisión. |

# **4.0 Documentación base para la ejecución de las pruebas**

*[Se sube el proyecto a un servidor y se procede a probar con el.]*

**4.1 Entorno de la prueba**

* + 1. **Hardware**

[Cualquier dispositivo con acceso a internet puede entrar]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Equipo** | **Procesador** | **DD** | **RAM** | **Aplicación a instalar** |
| PC Windows (XP o superior) | Intel | 10GB | 2GB | Windows – Herramientas ofimáticas |

* + 1. **Software**

[No se instala ningún software, se consulta mediante internet]

### **Datos de prueba**

“Se desarrollarán y especificarán conjuntos de datos de prueba, tomando las muestras necesarias para la ejecución de las pruebas, de manera que se verifique que cumple con diversos tipos de datos.”

* 1. **Identificación de la prueba**
     1. **Caso de prueba**

[Cada usuario realiza la consulta de datos sobre su situacion academica]

*[Se puede tomar como ejemplo la siguiente plantilla para la elaboración de un caso de prueba]*

|  |  |
| --- | --- |
| Columna | Instrucciones |
| Id | [Sugeto1] |
| Caso de Prueba | [Primer consulta.] |
| Descripción | [El usuario digito caracteres incorrectos y el algoritmo no le permitio el ingreso .] |
| Fecha | [20/11/2019.] |
| Funcionalidad / Característica | [Se logea en una plataforma y se selecciona la informacion a consultar] |
| Datos / Acciones de Entrada | [Usuario y contraseña- Codigo y nip.] |
| Resultado Esperado | [el usuario logra logearse y ver su informacion ] |
| Procedimientos especiales requeridos | [intentar ingresar con cuentas no dadas de alta] |
| Dependencias con otros casos de Prueba | [-] |
| **Información para el seguimiento** |  |
| Resultado Obtenido | [El usuario ingreso a su plataforma y pudo consultar facil y rapido] |
| Estado | [Optimo] |
| Última Fecha de Estado | [20/11/2019.] |
| Observaciones | [Ninguna importante] |

## **Resultados de las Pruebas / Corregir los defectos**

*[Las pruebas arrojaron que el software es optimo y funcional.]*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba** |  |  |  |  |
| **Componente / Producto** | **Caso de prueba** | **Resultado** | **Seguimiento** | **Conclusión** |
| *[Sitio de consulta]* | *[consulta.]* | *[La consulta fue exitosa]* | *[el usuario ingreso y consulto sus datos.]* | *[El algoritmo funciona.]* |

*Nota: Los defectos son corregidos hasta que se alcancen pruebas unitarias satisfactorias (por ejemplo, alcanzar los criterios de salida).*

|  |  |
| --- | --- |
|  |  |
| ***Objetivo:*** | Corregir los defectos encontrados por las Pruebas Unitarias |
| ***Razón Fundamental:*** | Corregir los defectos encontrados solo por las Pruebas Unitarias por la persona a cargo de la construcción del Componente, es la manera más rápida y menos costosa de corregir los defectos. |
| ***Roles:*** | Programador |
| ***Productos:*** | Componentes de Software [corregidos] |
| ***Pasos:*** | 1. Confirmar el defecto |
|  | 1. Determinar la naturaleza y ubicación del defecto |
|  | 1. Corregir el defecto |
|  | 1. Verificar las correcciones |
| ***Descripción de Pasos:*** | ***Paso 1. Confirmar el defecto***  Verificar que lo que se ha encontrado es un defecto en la implementación del código (el propósito de los casos de prueba) y no en el código de prueba mismo.  ***Paso* 2. Determinar la naturaleza y ubicación del defecto**  *Prácticas para ubicar errores:*   * ***Usar una lista de verificación de código:*** Si no se encuentra el error, se puede apoyar en una lista de verificación para la revisión de código (se provee un ejemplo de la sección de ejemplo). Esto da una idea de dónde buscar. * ***Ir a través del código en el depurador:*** Una vez que la rutina compila, colocarla en el depurador y pasar a través de cada línea de código. Asegurarse de que cada línea se ejecuta según lo esperado. * ***Encontrar la fuente de un error:*** Intentar reproducirlo de muchas diferentes maneras para determinar si es la causa exacta. Algunos consejos podrían ser: * Reducir las regiones de código sospechoso * Usar datos diferentes * Refinar los casos de prueba     **Figura 1. Reproducir un error de muchas maneras para determinar su causa exacta. [Código completo]**   * ***Usar la experimentación como último recurso:*** El error más común que cometen algunos programadores es intentar resolver un problema haciendo cambios experimentales al programa. Pero usualmente esto introduce más defectos.   ***Paso 3. Corregir el defecto***  Una vez que se haya encontrado la fuente del defecto, grabar el código fuente original y realizar los cambios correspondientes.  *Nota: Otra falla común es reparar los síntomas del error, o solo una instancia del error, en lugar de reparar el error mismo. Tener cuidado de esto.*  ***Paso 4. Verificar las correcciones***  Ejecutar las pruebas unitarias nuevamente para verificar que las correcciones realizadas funcionan propiamente.  Repetir desde el paso 1 cada vez que una prueba unitaria encuentre un defecto. Si la prueba unitaria no encuentra más defectos, se deberá buscar alcanzar el criterio de éxito seleccionado para el componente. |

## **Actualizar el Registro de Trazabilidad**

Nota: Incorporar los *Componentes de* *Software* construidos o modificados.

|  |  |
| --- | --- |
|  |  |
| ***Objetivos:*** | Asegurar que todos los Componentes de Software pueden ser trazados a un elemento de diseño y que todos los elementos de diseño han sido construidos. |
| ***Razón Fundamental:*** | El *Registro de Trazabilidad* deberá haber sido desarrollado en las fases previas del proyecto para asegurar la trazabilidad desde los requerimientos hacia los elementos de diseño. Esta tarea está dedicada a asegurar la trazabilidad entre los elementos de diseño y construcción. |
| ***Roles:*** | Programador |
| ***Productos:*** | Registro de Trazabilidad [actualizado] |
| ***Pasos:*** | 1. Actualizar el *Registro de Trazabilidad* |
| ***Descripción de Pasos:*** | ***Paso 1.* Actualizar el *Registro de Trazabilidad***  Este Registro deberá ser actualizado con la siguiente información:  - Identificación de los *Componentes de Software*  - Identificación de casos de prueba (opcional)  - Fecha de verificación (es decir, fecha en que el *Componente de Software* ha sido probado y no se encontraron defectos) |

## **Descripción de Roles**

Esta es una lista alfabética de los roles, abreviaciones y lista de competencias definidas en ISO 29110 Parte 5-1-2.

|  |  |  |  |
| --- | --- | --- | --- |
|  | ***Rol*** | ***Abreviación*** | ***Competencias*** |
| 1. | Analista | AN | Conocimiento y experiencia que permita obtener, especificar y analizar los requerimientos.  Conocimiento en diseño de interfaces de usuario y criterios ergonómicos.  Conocimiento en técnicas de revisión.  Conocimiento en técnicas de edición.  Experiencia en desarrollo y mantenimiento de Software. |
| 2. | Cliente | CL | Conocimiento de los procesos del Cliente y habilidad para explicar los requerimientos del Cliente.  El Cliente (representante) debe tener la autoridad de aprobar los requerimientos y sus cambios.  El Cliente incluye usuarios representativos con la finalidad de asegurar que el entorno operacional sea dirigido de forma correcta.  Conocimiento y experiencia en el dominio de la aplicación. |
| 3. | Gestor de Proyecto | GP | Capacidad de liderazgo con experiencia para toma de decisiones, planificación, gestión de personal, delegación y supervisión, conocimiento de finanzas y desarrollo de software. |
| 4. | Líder Técnico | LT | Conocimiento y experiencia en el dominio del proceso de software. |
| 5. | Programador | PR | Conocimiento y/o experiencia en programación, integración y pruebas unitarias.  Conocimiento de técnicas de revisión.  Conocimiento de técnicas de edición.  Experiencia en desarrollo y mantenimiento de software. |

## **Descripción de Productos**

Esta es una lista en orden alfabético de los productos de entrada, salida y de uso interno del proceso, sus descripciones, posibles estados y el origen del producto.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Nombre** | **Descripción** | **Fuente** |
| 1. | *Componente de Software* | Un conjunto de unidades de código relacionadas.  Los estados aplicables son: unidad probada, corregida e incorporada en la línea base. | Implementación de Software |
| 2. | *Diseño de Software* | Información textual y gráfica de la estructura del Software. Esta estructura puede incluir las siguientes partes:  Diseño Arquitectónico de Software – Describe la estructura global del *Software*:   * Identifica los stakeholders del diseño arquitectónico y sus preocupaciones * Identifica los mecanismos arquitectónicos relevantes (patrones, tácticas, herísticas, buenas prácticas, …) * Identifica los tipos de vistas que son relevantes para transmitir la arquitectura del software, tomando en consideración las preocupaciones de los stakeholders y los requerimientos varios (funcionales y no funcionales) * Provee vistas arquitectónicas relevantes del software en varias formas (diagramas, modelos, tablas, texto plano, …) * Identifica y describe los elementos principales de la arquitectura de software (subsistemas, capas, módulos) y sus relaciones * Identifica y describe los *Componentes* de Software requeridos, sus interfaces y las relaciones entre ellos * Describe la razón fundamental, provee cualquier análisis utilizado para producir la solución, identifica riesgos conocidos e inconsistencias   Diseño Detallado de Software – incluye detalles de los *Componentes de Software* para facilitar su construcción y pruebas dentro del entorno de programación:   * Proporciona diseño detallado (puede ser representado como un prototipo, diagrama de flujo, diagrama entidad relación, pseudo código, etc.) * Proporciona el formato de entrada / salida de los datos * Proporciona especificaciones de las necesidades de almacenamiento de los datos * Establece convenciones de denominación de los datos requeridos * Define el formato de las estructuras de datos requeridas * Define los campos de datos y el propósito de cada elemento de datos requerido * Proporciona las especificaciones de la estructura del programa   Los estados aplicables son: verificado e incorporado en la línea base. | Implementación de Software |
| 3. | Plan de Proyecto | Presenta cómo serán ejecutados los procesos y actividades del proyecto para asegurar su conclusión exitosa, así como la calidad de los productos entregables. Puede incluir los siguientes elementos y características:   * *Descripción de producto*   + Propósito   + Requisitos generales del *Cliente* * *Alcance* descripción respecto de lo que está incluido y de lo que no está incluido * *Objetivos* del proyecto * *Entregables* - lista de productos a ser entregados al *Cliente* * *Tareas*, incluyendo verificación, validación y revisiones con el *Cliente* y *Equipo de Trabajo* que permitan asegurar la calidad de los productos de trabajo. Las *Tareas* pueden ser representadas como una Estructura de Descomposición de Trabajo (EDT) * *Relación y Dependencia de las Tareas* * *Duración Estimada de las Tareas* * *Recursos* (humanos, materiales, estándares, equipos y herramientas), incluyendo la capacitación necesaria. Incluye la identificación y programación de los *Recursos* * Composición del *Equipo de Trabajo* * Calendario de las *Tareas* del proyecto, indicando la fecha de inicio y fecha de finalización previstas para cada *Tarea* y las relaciones y dependencias entre ellas * Esfuerzo y costo estimado * Identificación de los riesgos del proyecto * Estrategia de Control de Versiones   + Herramientas de repositorio del producto o mecanismos identificados   + Localización y mecanismos de acceso para el repositorio especificado   + Identificación y control de versiones definidos   + Respaldo y mecanismos de recuperación definidos   + Mecanismos de almacenamiento, manipulación y entrega especificados (incluyendo archivo y recuperación) * Instrucciones de Entrega   + Elementos requeridos para la liberación del producto (por ejemplo, hardware, *Software*, documentación, etc.)   + *Requisitos* de entrega   + *Tareas* a realizar en orden secuencial   + Liberaciones aplicables identificadas   + Identifica todos los Componentes de *Software* entregados con información dela versión   + Identifica cualquier procedimiento de copia de respaldo y recuperación necesarios   Los estados aplicables son: verificado, aceptado, actualizado y revisado. | Gestión del Proyecto |
| 4. | *Registro de Trazabilidad* | Documenta la relación entre los requisitos incluidos en la *Especificación de Requerimientos*, los elementos del *Diseño de Software*, los *Componentes de Software*, los *Casos de Prueba y Procedimientos de Prueba*.   * Identifica los requerimientos de la *Especificación de Requerimientos* por rastrear. * Proporciona el mapeo (hacia adelante y hacia atrás) de los requisitos a los elementos del *Diseño de Software*, los *Componentes de Software*, los *Casos de Prueba y Procedimientos de Prueba*.   Los estados utilizados son: verificado, en línea base y actualizado. | Implementación de Software |

## **Descripción de Artefactos**

Esta es una lista en orden alfabético de los artefactos que podrían ser producidos para facilitar la documentación de un proyecto. Los artefactos no son requeridos por la Parte 5, son opcionales.

|  |  |  |
| --- | --- | --- |
|  | **Nombre** | **Descripción** |
| 1. | Casos de Prueba | Un conjunto de entradas, condiciones de ejecución y resultados esperados de prueba desarrollados para un objetivo en particular, como ejercitar un camino particular de un programa o para verificar la conformidad con un requerimiento específico. [IEEE 1012-2004] |
| 2*.* | Conjunto de Casos de Prueba | Conjunto de rutinas escritas en un lenguaje específico diseñado para probar los casos de prueba. |
| 3. | Estándares de Construcción | Provee convenciones, reglas, idiomas y estilos para:   * Organización de código fuente (dentro de las sentencias, rutinas, clases, paquetes u otras estructuras) * Documentación de código * Módulos y archivos * Variables y constantes * Expresiones * Estructuras de control * Funciones * Manejo de condiciones de error — ambos errores planificados y excepciones (por ejemplo, ingreso de datos erróneos)   Entre otros. |
| 4. | Taxonomía de Defectos | Un método para reducir el número de defectos del producto al aprender acerca de los tipos de errores hechos en el proceso de desarrollo del producto. Así, ellos pueden ser analizados para mejorar el proceso para reducir o eliminar la probabilidad de cometer el mismo error en el futuro.  Los estados aplicables son: Verificado, Actualizado. |

# **5. Plantillas**

## **5.1 Plantilla de construcción para Java**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Copyright © 20XX [Nombre de Organización], Inc. Todos los derechos reservados.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Número de Versión - $Revision$

Última Actualización - $Fecha$

Actualizado Por - $Autor$

Resumen del propósito del módulo

Diseño de bajo nivel, discusiones de diseño físico, dependencias de construcción,

suposiciones, problemas de implementación, notas, etc.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Sentencias package e import

\*/

package com.mycompany.mypackage;

import com.somepackage;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Usar estos divisores para dividir el modulo en grupos lógicos.

\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Definiciones de la Clase

\*/

/\*============================================================================

Resumen de la meta de la clase

............................................................................

Descripción de la clase

============================================================================\*/

public class JavaClass extends SuperClass

implements SomeInterface

{

/\*--------------------------------------------------------------------

Constructores

...................................................................

Descripción de constructores

--------------------------------------------------------------------\*/

// Resumen de Constructor 1

public JavaClass( )

{

}

/\*====================================================================

Métodos públicos

\*/

/\*--------------------------------------------------------------------

Resumen de los métodos

....................................................................

Descripción del método

Describir las excepciones del método y valores de retorno

--------------------------------------------------------------------\*/

public RETURN\_TYPE

SomeMethod(

PARAM1\_TYPE param1, // Descripcion de param1

PARAM2\_TYPE param2 // Descripcion de param2

)

throws SomeException

{

}

/\*====================================================================

Métodos privados

\*/

/\*====================================================================

Métodos protegidos

\*/

protected boolean aBooleanValue;

/\*====================================================================

Campos privados

\*/

private int anIntValue;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

# **6. Ejemplo**

## **6.1 Ejemplo de un Estándar de Construcción General**

## **Formato**

|  |  |
| --- | --- |
| 1 | Todos los archivos fuente deberán estar basados en una plantilla de construcción apropiada. |
| 2 | Si **no existe un estilo discernible** en el código existente o el estilo difiere del estándar de construcción, **reformatear el archivo** de acuerdo al estándar. |
| 3 | Evitar crear líneas con más de 79 caracteres de longitud. Si se necesitan, indentar la siguiente línea un nivel. Si la nueva línea también excede 79 caracteres, insertar otro salto de línea luego del último operador dentro del límite de 79 caracteres, pero no indentar más la siguiente línea. Continuar hasta que todas las líneas de la sentencia sean menores que 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 | Usar **líneas en blanco** liberalmente para separar y organizar el código, 1 dentro de la función o comentario, 2 entre funciones, 3 ó 4 entre secciones. |
| 5 | **Colocar espacios** para separar visualmente las variables o funciones.  If ( 3 == x)  func1 ( param1, param2 );  int array [ MAX\_NUM\_THINGS ]; |

**Comentarios**

|  |  |
| --- | --- |
| 1 | Usar comentarios para describir las intenciones del programador describiendo **"porqué"** en lugar de **"cómo"** o **"qué"**. |
| 2 | Solo para **piezas de código críticas**, se deberá explicar en detalle la acción o como trabaja. Pero en ese caso, agregar una **marca especial**. |
| 3 | La mejor manera de comentar es haciéndolo en el **lenguaje natural** (inglés, español), no más código o pseudo-código. |
| 4 | Otros desarrolladores deberán entender el comentario (información **útil**). |
| 5 | Intentar no utilizar comentarios redundantes o innecesarios. |
| 6 | Comentar cada **bloque de código importante** con una breve descripción de su intención. |
| 7 | Indentar comentarios al **mismo nivel de indentación** del código que se está comentando, porque no son menos importantes que el código mismo. |
| 8 | Donde sea posible, usar un **formato de comentario procesable por la máquina** como **JavaDoc** para permitir la examinación de documentación a nivel de código de forma separada al código mismo. |
| 9 | Los **rangos de datos numéricos** deberán ser comentados. |
| 10 | Las **limitaciones en datos de entrada o salida** deberán ser comentados. |

**Módulos y archivos**

|  |  |
| --- | --- |
| 1 | Cuando se **nombra** un archivo **capitalizar** la primera letra y la **primera letra** de cualquier palabra o **acrónimo** dentro del nombre del archivo; usar minúsculas para todos los otros caracteres alfabéticos (usar solo caracteres **alfanuméricos**).  NombreArchivo.java  NombreArchivo2.cpp |
| 2 | Archivos con **propósitos** similares deberían ser **nombrados** similarmente. |
| 3 | **Empezar** todos los archivos fuente con el comentario de **cabecera** estándar describiendo el **propósito** del archivo y dando una **breve sinopsis** de sus contenidos. |
| 4 | Colocar más de un módulo o componente dentro de un único archivo fuente **no es recomendado**. |
| 5 | Intentar **utilizar rutas relativas** para prevenir incluir el archivo en cualquier ambiente. |

**Variables**

|  |  |
| --- | --- |
| 1 | Usar cada variable exactamente para un propósito. |
| 2 | **Capitalizar** la **primera letra** de palabras y acrónimos dentro del nombre de una variable; utilizar **minúsculas para todos los otros caracteres alfabéticos**. El primer caracter del nombre de una variable debería ser un caracter alfabético en minúscula.  balance // Nombre de variable de una palabra  totalMensual // Nombre de variable de dos palabras  exportaDoc // Nombre de variable con acronimo  Anteceder la declaración de una variable con un **breve comentario** describiendo el **propósito** de la variable. Porque incluso los nombres de variables más descriptivos son breves. |
| 3 | **Intentar no utilizar la notación de estilo húngaro** para amarrar el tipo de la variable al nombre de la variable porque puede distraer del propósito principal del nombre de la variable, que es describir el significado de la data que almacena la variable.  Notación húngara: float fbalance |
| 4 | **Inicializar** una variable cuando sea declarada, porque asegura un valor conocido para la variable. |

**Constantes**

|  |  |
| --- | --- |
| 1 | Usar caracteres alfanuméricos en **mayúsculas** para los nombres de constantes. Separar las palabras de una constante con **sub guiones**.  ESTA\_ES\_UNA\_CONSTANTE |
| 2 | Intentar no utilizar una constante numérica desnuda (**número mágico**), inclusive si se utiliza solo una vez. |
| 3 | Las cadenas de texto que serán mostradas a los usuarios deberían ser siempre extraídas en **constantes de cadenas** o tablas de cadenas. Esto les permite ser traducidas o actualizadas tal como se necesite.  OPTION1 = “Select a tool”  OPTION1 = “Selecciona una herramienta” |

**Expresiones y sentencias**

|  |  |
| --- | --- |
| 1 | En lenguajes donde se distinguen los operadores de asignación y comparación, tener cuidado con la ambigüedad cuando se realicen asignaciones dentro de expresiones condicionales.  Causa de ambigüedad: If ( value = 3 + i ) |
| 2 | Intentar tener un **orden** para las comparaciones para facilitar su comprensión. Por ejemplo usar **<** y **<=** en lugar de **>** o **>=**. |

**Estructuras de control**

|  |  |
| --- | --- |
| 1 | **Indentar** bloques de código usando el estilo de 'límites de bloques de inicio y fin' e indentarlos al **mismo nivel** del código contenido en el bloque.  if ( true == success )  {  // Do something.  } |
| 2 | Usar **nombres descriptivos** para índices de bucles, tal como "fil" y "col". Si el bucle es simple y pequeño, los índices tradicionales como 'i', 'j' y 'k' son aceptables. |
| 3 | Usar una sentencia **break** o **return** para salir de un bucle **no es recomendado**. |
| 4 | **Preferir bucles** forcuando sean apropiados, porque hace los bucles más fáciles de leer y mantener y previene caer en bucles infinitos. |
| 5 | Si se incluyen otras **sentencias** en la **cabecera** del bucle for, probablemente se debería usar un bucle while. |

Funci**ones**

|  |  |
| --- | --- |
| 1 | Seguir la **misma** forma para los **nombres de variables** cuando se nombran funciones, con la diferencia que el **primer** caracter de una función debería estar **en mayúscula**. |
| 2 | Para una función cuyo **propósito** principal es **realizar una acción**, use un nombre de función **verbo - objeto**. Si el método pertenece solo a un objeto, usar solo un **verbo**.  ImprimirDocumento( documento ); //verbo-objeto como nombre de funcion  documento.Imprimir(); //verbo como nombre de funcion |
| 3 | Para una función cuyo propósito principal es retornar un valor, usar un nombre que describa el valor retornado.  if ( FILE\_OPEN == file.State() ) |
| 4 | Anteceder cada declaración de función con un comentario describiendo 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 **sentencias de retorno múltiple** para salir de una función **no es recomendado**. |

## **6.2 Ejemplo de pseudocódigo**

### **Especificación**

Un procedimiento debe computar la prima asegurada para un conductor de carro, basado en la siguiente información:

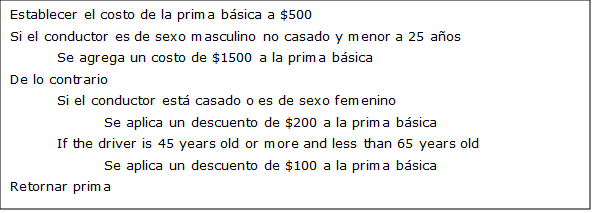
* La edad de la persona (entero).
* El sexo de la persona ("M" para masculino, "F" para femenino).
* El estado civil de la persona (casado o no).

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

* Una prima básica de $500 aplica para todos.
* Si la persona es de sexo masculino no casado y de 25 años, se agrega un costo de $1500 a la prima básica.
* Si la persona está casada o es de sexo femenino, se aplica un descuento de $200 en la prima básica.
* Si la persona tiene 45 años o más y menos de 65 años, se aplica un descuento de $100 en la prima básica.

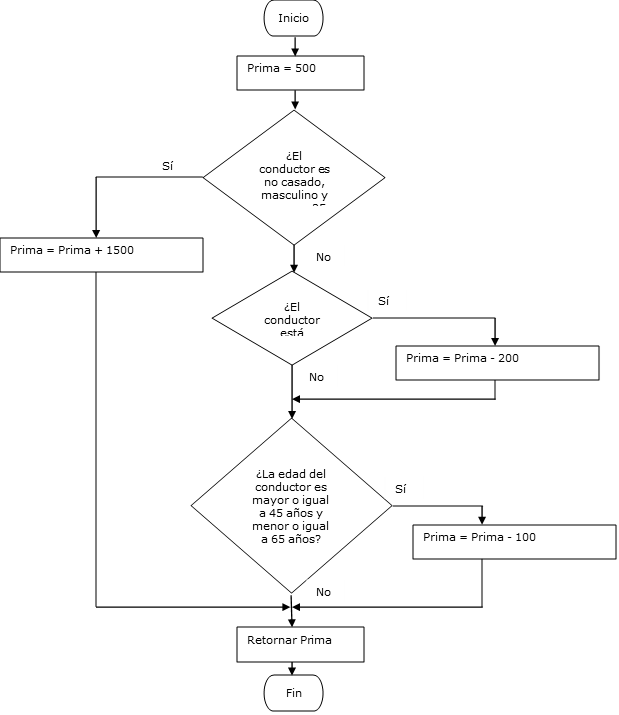
El procedimiento debe devolver la prima relacionada.

### **El Pseudocódigo basado en la especificación previa**



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 fuerzas de criterios de cobertura desde la más débil hacia la más fuerte**

|  |
| --- |
| **Rutina** |
| x = 0  **if**(age < 65 && married == **true**)  {  x = 2  y = y + age  }  **return** (age/x) |

A criterio más fuerte, se revelarán más defectos por las pruebas.

**Casos de prueba para cobertura de Sentencia:** Todas las sentencias tienen que ser ejecutadas al menos una vez; solo se requiere un caso de prueba para probar al 100% la cobertura de sentencia.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID de Caso de Prueba** | **Descripción** | **Entrada** |  | **Salida esperada** | **Aprueba / Falla** |
|  |  | **edad** | **casado** |  |  |
| CP1 | Una persona casada de 30 años | 30 | True | 15 | P |

*Nota: Con este caso de prueba se ejecutan todas las líneas de código, pero el caso donde la decisión es falso no es probado, así que no se puede detectar el defecto aritmético de la división cuando x = 0.*

**Casos de prueba para cobertura Rama/Decisión:** Cada decisión debe tomar cada salida posible al menos una vez.

#### *Decisión: edad < 65 y casado == true*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **ID de Caso de Prueba** | **Descripción** | **Entrada** |  | **Salida de decisión** *(Predicado compuesto como un todo)* | **Salida esperada** | **Aprueba / Falla** |
|  |  | **edad** | **casado** |  |  |  |
| CP1 | Una persona casada de 30 años | 30 | True | True | 15 | P |
| CP2 | Una persona casada de 75 años | 75 | True | False | 0 | F |

*Notas:*

* *El caso de prueba CP2 detecta un defecto que la cobertura de sentencia no pudo.*
* *El valor para casado = False no se considera y aún así se alcanza el 100% de la cobertura de decisión. Esto sucede porque todos los valores posibles de cada condición no fueron analizados. Así, si un defecto estaba escondido en la condición 2 (casado==true) puede que no sea detectado.*

### **Casos de prueba para cobertura de Decisión y Condición**

Cada decisión debe tomar cada salida posible al menos una vez, y cada condición debe tomar cada valor posible al menos una vez.

Decisión: edad < 65 y casado == true

Condición 1: edad < 65

Condición 2: casado == true

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ID de Caso de Prueba** | **Descripción** | **Entrada** |  | **Salida de Condición 1** | **Salida de Condición 2** | **Salida de decisión** | **Salida esperada** | **Aprueba / Falla** |
|  |  | **edad** | **casado** |  |  |  |  |  |
| CP1 | Una persona casada de 30 años | 30 | True | True | True | True | 15 | P |
| CP2 | Una persona casada de 75 años | 75 | True | False | True | False | 0 | F |
| CP3 | Una persona no casada de 30 años | 30 | False | True | False | False | 0 | F |

*Notas: Si la condición 2 tuvo un defecto de lógica en su salida falsa, entonces el caso de prueba CP3 lo detectará. Sin embargo, este no es el caso.*

El criterio descrito previamente no requiere la cobertura de todas las combinaciones posibles de las condiciones. Sin embargo, puesto que los compiladores usualmente se detienen de evaluar una expresión "*and*" tan pronto como determina que un operando es *falso* o una expresión "*or*" tan pronto como se determine que un operando es *verdadero*. Algunas de las condiciones no serán evaluadas sin importar su valor.

Para resolver este problema, se recomienda probar los operadores "*and*" y "*or*" en el siguiente orden:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| if( a && b && c && d )   |  |  | | --- | --- | | **Entradas** | **Valores (T=verdadero, F=falso)** | | 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 )   |  |  | | --- | --- | | **Entradas** | **Valores (T=verdadero, F=falso)** | | 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 | |

De esta manera, se asegura que cada condición tome las salidas verdadera y falsa dentro de la ejecución del programa.

Para satisfacer la parte de decisión de cobertura de *decisión y condición*:

En caso de los operadores "*and*", el primer conjunto (T T T T) trabajará para evaluar la salida verdadera de la decisión y cualquiera de los otros casos trabajará en evaluar la salida falsa.

En caso de los operadores "*or*", el conjunto (F F F F) trabajará para evaluar la salida falsa de la decisión y cualquiera de los otros casos trabajará para evaluar la salida verdadera.

*Nota: Si la decisión está compuesta por una mezcla de operadores "and" y "or", se pueden usar paréntesis "(,)" para determinar la jerarquía de los operadores que se ajuste mejor.*

## **6.5 Diseño de caso de prueba, escritura de prueba unitaria y uso de herramienta de cobertura de código.**

|  |
| --- |
| **Rutina** |
| **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;  } |

*Nota: Las pruebas unitarias fueron escritas utilizando aserciones con los valores de entrada de cada caso de prueba y con la salida esperada. Por conveniencia del lenguaje elegido es Java, el framework de Pruebas Unitarias es JUnit y la herramienta de cobertura de código es Codecover.*

### **Cobertura de Sentencias**

Para probar todas las sentencias, se necesita probar las salidas verdaderas y falsas de la primera decisión (primer if)

|  |  |  |  |
| --- | --- | --- | --- |
| **ID de Caso de Prueba** | **Descripción** | **Entradas** | **Salidas esperadas** |
| CP1 | Un conductor masculino no casado de 20 años | Edad = 20, No casado, Masculino | 2000 |
| CP2 | Un conductor masculino casado de 50 años | Edad = 50, Casado, Masculino | 200 |

**Tabla 4 Casos de prueba requeridos para satisfacer el 100% de la cobertura de sentencia**

|  |  |
| --- | --- |
| **Pruebas unitarias** | **Cobertura de código alcanzada** |
| **public** **void** testStatement(){  // TC1  pa1.setAge(20);  pa1.setMarried(**false**);  pa1.setSex(“M”);  *assertEquals*(pa1.getPremium(), 2000);  // TC2  pa1.setAge(50);  pa1.setMarried(**true**);  pa1.setSex(“M”);  *assertEquals*(pa1.getPremium(), 200);  } |  |

**Tabla 5 las pruebas unitarias satisfacen el 100% de cobertura de Sentencias. El color del código tiene el siguiente significado: Verde - cubierto completamente, Amarillo = cubierto parcialmente, Rojo - nunca alcanzado (plugin Codecover para Eclipse Java IDE)**

Como se puede ver en la figura anterior, se ejecutan todas las sentencias pero no se ha ejercido la salida de cada decisión o condición.

### **Cobertura Rama/Decisión**

Las decisiones de la rutina son:

|  |  |
| --- | --- |
| D1 | **if** ((age < 25) && !married && sex.equals(“M”)) |
| D2 | **if** (married || sex.equals(“F”)) |
| D3 | **if** ((age >= 45) && (age < 65)) |

|  |  |  |
| --- | --- | --- |
| **Decisión** | **Salida verdadera** | **Salida falsa** |
| D1 | Un conductor masculino no casado menor a 25 años | Un conductor de 25 años o más, o casado, o femenino |
| D2 | Un conductor casado o femenino | Un conductor no casado masculino |
| D3 | Un conductor entre 45 y 64 años | Un conductor menor a 45 años o mayor a 65 años |

**Tabla 6 Situaciones de Entrada para ejercer todas las salidas de decisión**

|  |  |  |  |
| --- | --- | --- | --- |
| **ID de Caso de Prueba** | **Descripción**  *(Probar las salidas)* | **Entradas** | **Salidas esperadas** |
| CP1 | D1T | Edad = 20, No casado, Masculino | 2000 |
| CP2 | D1F,D2T,D3T | Edad = 45, Casado, Masculino (irrelevante) | 200 |
| CP3 | D1F,D2F,D3F | Edad = 35, No casado, Masculino | 500 |

**Tabla 7 Casos de prueba obtenidos. Nota: D#T significa probar la salida verdadera de la decisión relacionada. D#F significa probar la salida falsa de las decisión relacionada.**

|  |  |
| --- | --- |
| **Pruebas Unitarias** | **Cobertura de código alcanzada** |
| **public** **void** testDecision(){  // TC1  pa1.setAge(20);  pa1.setMarie(**false**);  pa1.setSexe(« M »);  *assertEquals*(pa1.getPrime(), 2000);  // TC2  pa1.setAge(45);  pa1.setMarie(**true**);  pa1.setSexe(“M”);  *assertEquals*(pa1.getPrime(), 200);  // TC3  pa1.setAge(35);  pa1.setMarie(**false**);  pa1.setSexe(“M”);  *assertEquals*(pa1.getPrime(), 500);  } |  |

**Tabla 8 las pruebas unitarias satisfacen el 100% de cobertura Rama/Decisión. El color del código tiene el siguiente significado: Verde - cubierto completamente, Amarillo = cubierto parcialmente, Rojo - nunca alcanzado (plugin Codecover para Eclipse Java IDE)**

Como se puede ver en la figura anterior, las condiciones de las decisiones están cubiertas parcialmente (código en color amarillo) puesto que no se probaron todas las posibles salidas.

### **Cobertura de Decisión y Condición**

|  |  |
| --- | --- |
| D1 | **if** ((age < 25) && !married && sex.equals(“M”)) |
| D2 | **if** (married || sex.equals(“F”)) |
| D3 | **if** ((age >= 45) && (age < 65)) |

Las condiciones de las decisiones son:

|  |  |  |
| --- | --- | --- |
| 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 |

|  |  |  |
| --- | --- | --- |
| **Condición** | **Salida verdadera** | **Salida falsa** |
| C1 | Un conductor menor a 25 años | Un conductor de 25 años o más |
| C2 | Un conductor no casado | Un conductor casado |
| C3 | Un conductor masculino | Un conductor femenino |
| C4 | Un conductor casado | Un conductor no casado |
| C5 | Un conductor femenino | Un conductor masculino |
| C6 | Un conductor de 45 años o más | Un conductor menor a 45 años |
| C7 | Un conductor menor a 65 años | Un conductor de 65 años o más |

**Tabla 9 Situaciones de salida para evocar todas las condiciones de salida**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID de Caso de Prueba** | **Descripción**  *(Prueba las salidas)* |  | **Entrada** | **Salida Esperada** |
|  | **Decisiones** | **Condiciones** |  |  |
| TC1 | D1T | C1T,C2T,C3T | Edad = 20, No casado,Masculino | 2000 |
| TC2 | D1F | C1F,C2T,C3T | Edad = 50, No casado,Masculino | 400 |
| TC3 | D1F | C1T,C2F,C3T | Edad = 20, Casado,Masculino | 300 |
| TC4 | D1F | C1T,C2T,C3F | Edad = 20, No casado,Femenino | 300 |
| TC2 | D2F | C4F,C5F | Edad = 50, No casado,Masculino | 400 |
| TC3 | D2T | C4T,C5F | Edad = 20, Casado,Male | 200 |
| TC4 | D2T | C4F,C5T | Edad = 20, No casado,Femenino | 200 |
| TC2 | D3T | C6T,C7T | Edad = 50, No casado,Masculino | 400 |
| TC5 | D3F | C6F,C7T | Edad = 30, No casado,Masculino | 500 |
| TC6 | D3F | C6T,C7F | Edad = 70, No casado,Masculino | 500 |

**Tabla 10 Casos de prueba obtenidos. Nota: C#T significa las pruebas de las salidas verdaderas de cada condición relacionada. C#F significa las pruebas de las salidas falsas de las decisiones relacionadas**

*Nota 1: Los casos de prueba fueron obtenidos siguiendo los consejos brindados en el ejemplo 6.4 para este criterio de cobertura.*

*Nota 2: Recordar que algunas veces la misma configuración de entrada puede probar más de un caso. Así, tratar de siempre optimizar el número de casos de prueba.*

|  |
| --- |
| **Pruebas unitarias** |
| **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);  } |

**Tabla 11 las pruebas unitarias satisfacen el 100% de la cobertura de decisión y condición. Pero la herramienta de cobertura utilizada no mide este criterio. Sin embargo, existen otras herramientas que pueden medir este criterio.**

Con este conjunto de pruebas unitarias, cada decisión y condición toma cada salida posible al menos una vez.

Como se puede ver, para satisfacer criterios más fuertes se necesitan usualmente más casos de prueba. Pero estos componentes se vuelven mejor probados y con menos agujeros donde se puedan esconder los defectos.

## **6.6 Pruebas unitarias siguiendo Técnicas de Pruebas Estructuradas y mapeando los pasos de la tarea "Diseñar o actualizar casos de pruebas unitarias y aplicarlos".**

El propósito de las pruebas estructuradas es garantizar automáticamente ambas coberturas de sentencia y de rama/decisión. El proceso de pruebas estructuradas consiste en los siguientes pasos:

1. **Derivar el grafo de control de flujo desde el módulo software**

Todos los programas estructurados pueden ser construidos por tres estructuras básicas: secuencial (por ejemplo, sentencias de asignación), decisión (por ejemplo, sentencias if/then/else), e iterativas (por ejemplo, bucles while, for).

|  |  |  |
| --- | --- | --- |
| Secuencia | Condición | Iteración |
|  |  |  |

**Figura 5 Representaciones gráficas de las tres estructuras básicas**

Usando estas estructuras y una combinación de ellas, se puede desarrollar un grafo de control de flujo para el software bajo pruebas.

|  |  |
| --- | --- |
| **Rutina** | **Grafo de control de flujo** |
| 1 **public** **int** getPremium() {  2**int** premium = 500;  3**if** ((age < 25) && !married && sex.equals(“M”)) {  4premium += 1500;  5} **else** {  6**if** (married || sex.equals(“F”)) {  7premium -= 200;  8}  9**if** ((age >= 45) && (age < 65)) {  10premium -= 100;  11}  12}  13**return** premium;  14 } |  |

*Nota: Existen herramientas comerciales que generarán grafos de control de flujo desde el código y en algunos casos desde pseudo código.*

1. **Computar la complejidad ciclomática del grafo ©**

Una vez que se tenga el grafo de control de flujo de la unidad que se quiere probar, se determina el mínimo número de casos de prueba necesarios para cubrir todas las sentencias y decisiones. Este número se llama complejidad ciclomática10 y se calcula con la siguiente ecuación (asumiendo un único nodo de entrada y un único nodo de salida):

C = aristas – nodos + 2

En nuestro ejemplo, la complejidad ciclomática es: C = 9 – 7 + 2 = 4. Entonces se necesitan al menos 4 casos de prueba para satisfacer este criterio.

*Nota: El paso 1 "Obtener el criterio de salida" de la tarea "Diseñar o actualizar casos de pruebas unitarias y aplicarlos" se establece por defecto para satisfacer Cobertura Rama/Decisión, puesto que es el objetivo de las pruebas estructuradas.*

1. **Seleccionar un conjunto de rutas base**

Para determinar el conjunto de rutas base11 se debería seguir este proceso:

* Tomas una ruta de "referencia". Esta ruta debería ser una "típica" razonable ruta de ejecución en lugar de una ruta de procesamiento de excepción.
* Para elegir la siguiente ruta, cambiar la salida de la primera decisión a lo largo de la ruta de referencia mientras se mantiene el número máximo de otras decisiones del mismo como la ruta de referencia.
* Para generar la tercera ruta, empezar nuevamente con la de referencia pero variar la salida de la segunda decisión en lugar de la primera.
* Para generar la cuarta ruta, empezar nuevamente con la de referencia pero variar la salida de la tercera decisión en lugar de la segunda. Continuar variando cada decisión, una por una, hasta que se alcance el fondo del grafo.
* Se continúa este patrón hasta que el conjunto de rutas base esté completo.

Siguiendo estos pasos desde arriba y eligiendo como ruta de referencia A-B-G las rutas bases resultantes son:

* A-B-G
* A-C-D-E-F-G
* A-C-E-F-G
* A-C-D-E-G

1. **Crear un caso de prueba para cada ruta base**

El siguiente paso es crear un caso de prueba para cada ruta. Este conjunto de casos de prueba garantizará ambos la cobertura de sentencias y ramas. Notar que se pueden crear múltiples conjuntos de rutas base y no son necesariamente únicas. Cada conjunto, sin embargo, tiene la propiedad de que un conjunto de casos de prueba basados en ello ejecutará cada sentencia y cada rama.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID de Caso de Prueba** | **Descripción** (Ruta cubierta) | **Entrada** |  |  | **Salida esperada** |
|  |  | **Edad** | **Casado** | **Sexo** |  |
| 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 |

*Nota: El paso 2 "Diseñar los casos de prueba" de la tarea "Diseñar o actualizar los casos de pruebas unitarias y aplicarlos" está cubierto con este paso.*

*Nota: Este conjunto de casos de prueba nunca prueba si la entrada Sexo con el valor "Femenino" y aún así cubre todo el grafo. Para ser estricto, el criterio de Decisión y Condición puede cubrir este problema.*

1. **Ejecutar las pruebas**

Este paso puede ser dividido en dos sub-pasos:

* 1. ***Codificar las pruebas unitarias***

Las pruebas unitarias para los casos de prueba son

|  |
| --- |
| **Pruebas unitarias** |
| **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);  } |

*Nota: El paso 3 "Codificar las pruebas unitarias" de la tarea "Diseñar o actualizar los casos de pruebas unitarias y aplicarlos" está cubierto con este sub-paso.*

* 1. ***Ejecutar las pruebas unitarias***

Para la ejecución de las pruebas unitarias, se puede usar un ejecutor de pruebas de línea de comando o gráfico. Al final, si alguno de ellos falla, el framework mostrará cual falló. Así en ese caso, se necesitan hacer las correcciones pertinentes. (Revisar el ejemplo 6.7 para la corrección de defectos)

Para este ejemplo, todas las pruebas unitarias aprueban.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **ID de Caso de Prueba** | **Descripción: Ruta cubierta** | **Entradas** |  |  | **Salida Esperada** | **Aprueba/Falla** |
|  |  | **Edad** | **Casado** | **Sexo** |  |  |
| 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 |

*Nota: El paso 4 "Ejecutar las pruebas unitarias" de la tarea "Diseñar o actualizar los casos de pruebas unitarias y aplicarlos" está cubierto con este sub-paso.*

*Nota: El paso 5 "Analizar los resultados" de la tarea "Diseñar o actualizar los casos de pruebas unitarias y aplicarlos" podría ser cubierto luego de este paso. El proceso de pruebas estructuradas no lo menciona pero estaría implícito.*

## **6.7 Corrección de defectos**

El siguiente cambio en el código fuente del ejemplo 6.5 introduce un defecto:

|  |
| --- |
| **Rutina** |
| 1 **public** **int** getPremium() {  2**int** premium = 500;  3**if** ((age < 25) && !married && sex.equals(“M”)) {  4premium += 1500;  5} **else** {  6**if** (married || sex.equals(“F”)) {  7premium -= 200;  8}  9**if** ((age > 45) && (age < 65)) {  10premium -= 100;  11}  12}  13**return** premium;  14 } |

Cuando las pruebas unitarias diseñadas para Cobertura de Decisión son ejecutadas, se muestra un error y una de las pruebas falla.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID de Caso de Prueba** | **Descripción** | **Entradas** | **Salidas esperadas** | **Aprueba / Falla** |
| CP1 | Un conductor no casado masculino de 20 años | Edad = 20, No casado, Masculino | 2000 | P |
| CP2 | Un conductor casado masculino de 45 años | **Edad = 45**, Casado, Masculino | 200 | **F** |
| CP3 | Un conductor no casado masculino de 35 años | Edad = 35, No casado, Masculino | 500 | P |

|  |  |
| --- | --- |
| **Pruebas unitarias** | **Salida de ejecutor de pruebas para JUnit** |
| **public** **void** testDecision(){  // TC1  pa1.setAge(20);  pa1.setMarie(**false**);  pa1.setSexe(« M »);  *assertEquals*(pa1.getPrime(), 2000);  // TC2  pa1.setAge(45);  pa1.setMarie(**true**);  pa1.setSexe(“M”);  *assertEquals*(pa1.getPrime(), 200);  // TC3  pa1.setAge(35);  pa1.setMarie(**false**);  pa1.setSexe(“M”);  *assertEquals*(pa1.getPrime(), 500);  } |  |

**Tabla 12 cuando se ejecutan las pruebas unitarias, el Ejecutor de pruebas muestra un defecto. Salida esperada: 200, Salida obtenida: 300**

Puesto que el error no está en el método de prueba sino en el código de implementación, el siguiente paso es encontrar la ubicación del defecto.

**Line 9 if** ((age > 45) && (age < 65)) {

Una vez que se ha encontrado el defecto, se debería ver que no exista otro defecto alrededor del código. Si este no es el caso, se debería realizar la corrección y entonces ejecutar el conjunto de pruebas unitarias nuevamente para verificar que no se haya "roto" funcionalidad que trabajaba previamente.

**Line 9 if** ((age >= 45) && (age < 65)) {

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID de Caso de Prueba** | **Descripción** | **Entradas** | **Salidas esperadas** | **Aprueba / Falla** |
| CP1 | Un conductor no casado masculino de 20 años | Edad = 20, No casado, Masculino | 2000 | P |
| CP2 | Un conductor casado masculino de 45 años | Edad = 45, Casado, Masculino | 200 | P |
| CP3 | Un conductor no casado masculino de 35 años | Edad = 35, No casado, Masculino | 500 | P |

|  |  |
| --- | --- |
| **Pruebas unitarias** | **Salida de ejecutor de pruebas para JUnit** |
| **public** **void** testDecision(){  // TC1  pa1.setAge(20);  pa1.setMarie(**false**);  pa1.setSexe(« M »);  *assertEquals*(pa1.getPrime(), 2000);  // TC2  pa1.setAge(45);  pa1.setMarie(**true**);  pa1.setSexe(“M”);  *assertEquals*(pa1.getPrime(), 200);  // TC3  pa1.setAge(35);  pa1.setMarie(**false**);  pa1.setSexe(“M”);  *assertEquals*(pa1.getPrime(), 500);  } |  |

**Tabla 13 una vez que el defecto se corrige, las pruebas unitarias se ejecutan nuevamente y ellas aprueban satisfactoriamente.**

Ahora que los componentes aprobaron todas las pruebas, se debería decidir si se debe dejar de probar o continuar creando más casos de prueba. Esto depende del criterio de salida que el Gestor de Proyecto solicite.

## **6.8 Ejemplos de ejecutores de pruebas**

**Ejecutor de pruebas de Línea de Comando**

El siguiente es un ejemplo de ejecución de pruebas unitarias en rubyUnit desde la línea de comandos:

>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

La primera línea es la invocación en el símbolo del sistema. En este ejemplo, se ejecutan las pruebas definidas en MySuiteTest. Las siguientes dos líneas son la retroalimentación inicial una vez que inicia. La serie de puntos indica el progreso, uno por prueba completada. Las últimas dos líneas son el resumen estadístico que provee una visión general de lo que sucedió. Usualmente, el código de salida es el número de pruebas que fallaron.

**Ejecutor de pruebas Gráfico**

Las siguientes son imágenes de dos ejecutores de pruebas gráficos.

|  |  |
| --- | --- |
| U,{8d77fe8c-a6f6-41d4-b640-b6bbfdeaa059}{140},0.6666666666666666,0.6666666666666666  **Figura 6. Ejecutor de pruebas para Ruby on Rails** | **Figura 7. Ejecutor de pruebas para Junit4 en Eclipse Java EE IDE** |

## **6.9 Ejemplo de macro de aserción en C++**

#define ASSERT ( condition, message ){

if( !(condition) ){

fprintf (stderr, “Asercion %s fallo: %s\n”,

#condition, message);

exit( EXIT\_FAILURE );

}

}

## **6.10 Ciclo de vida de las pruebas unitarias**

U,{8d77fe8c-a6f6-41d4-b640-b6bbfdeaa059}{179},0.6666666666666666,0.6666666666666666

# **7. Listas de Verificación**

## **Listas de Verificación de Tareas**

## **7.1 Asignar tareas a los miembros del Equipo de Trabajo**

|  |  |
| --- | --- |
| o | ¿Se ha obtenido la documentación del Diseño de Software? |
| o | ¿Se ha seleccionado una estrategia de secuencia de integración? |
| o | ¿Se ha detallado el cronograma del proyecto? |
| o | ¿Se han asignado tareas a los miembros del Equipo de Trabajo? |
| o | ¿Se han definido los criterios de salida para las pruebas unitarias? |

## **7.2 Construir o actualizar los Componentes de Software**

|  |  |
| --- | --- |
| o | ¿Se han entendido el Diseño Detallado y la contribución de componentes? |
| o | ¿Se ha buscado funcionalidades disponibles en las librerías estándar? |
| o | ¿Se ha definido la lógica del componente? |
| o | ¿Se han seguido los estándares de construcción mientras se codificaba? |
| o | ¿Se han verificado los componentes? |

## **7.3 Diseñar o actualizar los casos de pruebas unitarias y aplicarlas**

|  |  |
| --- | --- |
| o | ¿Se han obtenido los criterios de salida? |
| o | ¿Se han diseñado los casos de prueba? |
| o | ¿Se han codificado las pruebas unitarias? |
| o | ¿Se han ejecutado las pruebas unitarias? |
| o | ¿Se han analizado los resultados? |

## **7.4 Corregir los defectos**

|  |  |
| --- | --- |
| o | ¿Se han confirmado los defectos? |
| o | ¿Se ha determinado la ubicación del error sospechoso? |
| o | ¿Se ha buscado por errores similares al que se encontró alrededor del código? |
| o | ¿Se está corrigiendo el error desde la fuente y no parchándose? |
| o | ¿Se han verificado las correcciones? |
| o | ¿La corrección cambia el diseño del componente? (De ser así, comunicarlo al Líder Técnico). |

## **Listas de Verificación de Soporte**

## **7.5 Lista de verificación de revisión de código**

Nota: Esta lista de verificación puede ser adaptada para aplicarse al lenguaje en que se está programando.

Se agrega una etiqueta delante de cada elemento de la lista de verificación para ayudar a acelerar las anomalías de registro.

|  |  |  |
| --- | --- | --- |
| **Etiqueta - Tema** |  | **Descripción** |
| o | **CR1 Completo** | * Verificar que todas las funciones en el diseño están codificadas y que todas las funciones y procedimientos necesarios han sido implementados. |
| o | **CR2 Lógico** | * Verificar que el flujo del programa y la lógica de todos los procedimientos y funciones es consistente con el diseño detallado. |
| o | **CR3 Bucles** | * Asegurar que cada bucle está iniciado y terminado apropiadamente. * Verificar que cada bucle se ejecuta la cantidad de veces correcta. |
| o | **CR4 Llamadas** | * Verificar cada llamada a función y procedimiento para asegurar que coincida exactamente con las definiciones para formatos y tipos. |
| o | **CR5 Declaraciones** | Verificar que cada variable y parámetro:   * tiene exactamente una declaración * es utilizado solo dentro del alcance declarado * está escrito correctamente donde se usa |
| o | **CR6 Inicialización** | * Verificar que cada variable está inicializada. |
| o | **CR7 Límites** | * Verificar todas las variables, arreglos e índices para asegurar que su uso no excede los límites declarados. |
| o | **CR8 Inicio-fin** | * Verificar todos los pares de inicio-fin o equivalentes, incluyendo casos donde ifs anidados pueden ser malinterpretados. |
| o | **CR9 Boolean** | * Verificar las condiciones Booleanas. |
| o | **CR10 Formato** | * Verificar cada línea del programa por formato de instrucción, ortografía y puntuación. |
| o | **CR11 Punteros** | * Verificar que todos los punteros se usen apropiadamente. |
| o | **CR12 Entrada-salida** | * Verificar todos los formatos de entrada y salida. |
| o | **CR13 Ortografía** | * Verificar que cada variable de entrada, parámetro y trabajo clave está escrito propiamente. |
| o | **CR14 Comentarios** | * Asegurar que todos los comentarios son precisos y están acorde al estándar. |
| o | **CR15 Cálculos** | * Verificar que todas las operaciones aritméticas y ecuaciones expresan lo que indica el algoritmo. |
| o | **CR16 Base de datos** | * Asegurar que todas las conexiones sean cerradas. * Verificar que los manejadores de conexiones están escritos propiamente. |

## **7.6 Lo que deberían proveer el arquitecto y el diseñador**

### **Requerimientos**

|  |  |
| --- | --- |
| o | ¿Cualquier requerimiento debería ser especificado en más detalle antes de la construcción? |
| o | ¿Se especifican las entradas, incluyendo su fuente, precisión, rango de valores y frecuencia? |
| o | ¿Se especifican las salidas, incluyendo su destino, precisión, rango de valores, frecuencia y formato (incluyendo páginas web, reportes, entre otros)? |
| o | ¿Se especifican todas las interfaces externas de hardware y software? |
| o | ¿Se especifica el máximo de memoria/almacenamiento? |
| o | ¿Se especifican consideraciones de tiempo, tales como tiempo de respuesta, tiempo de procesamiento, velocidad de transferencia de datos y rendimiento del sistema? |
| o | ¿Se especifica el nivel de seguridad? |

### **Arquitectura y Diseño**

|  |  |
| --- | --- |
| o | ¿Los componentes o los datos de diseño más críticos están descritos y justificados? |
| o | ¿Se especifica la organización de la base de datos y el contenido? |
| o | ¿Se describe una estrategia para el diseño de interfaces de usuario? |
| o | ¿Se describe y justifica una estrategia para manejar Entrada/Salida? |
| o | ¿Se describen los requerimientos de seguridad de la arquitectura? |
| o | ¿Se provee una estrategia coherente de manejo de errores? |

## **7.7 Sub-tarea: Seleccionar el estándar de interfaz de usuario**

|  |  |
| --- | --- |
| o | ¿Se ha planificado la sub-tarea? |
| o | ¿Se han obtenido estándares de interfaz disponibles? |
| o | ¿Se ha seleccionado el estándar de interfaz? |
| o | ¿Se ha obtenido la aprobación del cliente? |
| o | ¿Los programadores han adoptado el estándar? |
| o | ¿Se ha verificado la adopción del estándar? |

# **8. Herramientas**

## **8.1 Matriz de Trazabilidad**

* Objetivos:
  + Para mantener el vínculo del origen de cada requerimiento hacia su descomposición a la implementación y prueba (verificación).
  + Para asegurar que todos los requerimientos son dirigidos y que solo se desarrollará lo que es requerido.
  + Útiles cuando se conduce el impacto del asesoramiento de los requerimientos, diseño u otros cambios en elementos configurados.

Nota: Una Matriz de Trazabilidad es una herramienta ampliamente usada para implementar el Registro de Trazabilidad.

U,{2e9a6781-a8a6-4099-b670-f9e0e4b25273}{178},0.6666666666666666,0.6666666666666666

|  |  |
| --- | --- |
| **Instrucciones** |  |
| La tabla anterior debe ser creada en una hoja de cálculo o una base de datos que sea fácil de ordenar por cada columna para alcanzar la trazabilidad bidireccional entre las columnas. Los identificadores únicos para los elementos deberían ser asignados en un formulario de esquema jerárquico de tal forma de que los elementos de bajo nivel (es decir, más detallados) puedan ser trazados con los elementos de alto nivel. |  |
| Identificación Única del Requerimiento (ID) | El ID Único del Requerimiento / Declaración de Requerimiento del Sistema donde el requerimiento es referenciado, y/o el identificador único (ID) para requerimientos descompuestos |
| Descripción de Requerimientos | Ingresar la descripción del requerimiento (por ejemplo, Descripción de Solicitud de Cambio). |
| Referencia de Diseño | Ingresar el número de párrafo donde la SC es referenciada en la documentación de diseño |
| Módulo / Referencia de elemento configurado | Ingresar el identificador único que el módulo de software o el elemento configurado donde el diseño es realizado. |
| Referencia de Liberación | Ingresar el número de versión de la liberación/entregable donde el requerimiento está satisfecho |
| Nombre de Script de Prueba /Referencia de Número de Paso | Ingresar el nombre del script de prueba/número de paso donde el requerimiento está referenciado (por ejemplo, Paso 1) |

|  |  |
| --- | --- |
| **Guía** | La trazabilidad de requerimientos debería:   * Asegurar la trazabilidad para cada nivel de descomposición realizado en el proyecto. En particular:   + Asegurar que cada requerimiento de bajo nivel puede ser trazado a un requerimiento de alto nivel o fuente original   + Asegurar que cada elemento de diseño, implementación, y prueba puede ser trazado a un requerimiento   + Asegurar que cada requerimiento está representado en el diseño e implementación   + Asegurar que cada requerimiento es representado en las pruebas/verificación * Asegurar que cada trazabilidad es usada en conducir el asesoramiento de impactos de los cambios del requerimiento en el plan de proyecto, actividades y productos de trabajo * Estar mantenida y actualizada si algún cambio ocurre. * Ser consultada durante la preparación del Asesoramiento de Impactos para cada cambio propuesto en el proyecto * Estar planeada, ya que el mantenimiento de los links/referencias es un proceso de labor intensiva que debería ser seguido/monitoreado y debería ser asignado a un miembro del equipo de proyecto * Ser mantenida como un documento electrónico |

## **8.2 Herramientas de Cobertura de Código**

Vínculos a las herramientas más populares de cobertura

|  |  |  |
| --- | --- | --- |
| **Herramienta** | **Lenguaje** | **Fuente** |
| 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 Frameworks de Pruebas Unitarias**

Vínculos a los frameworks más populares para Pruebas Unitarias

|  |  |  |
| --- | --- | --- |
| **Herramienta** | **Lenguaje** | **Fuente** |
| 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. Referencias a Otros Estándares y Modelos**

Esta sección provee referencias de este paquete de despliegue para los estándares ISO e ISO/IEC seleccionados y para Capability Maturity Model IntegrationSM versión 1.2 del Software Engineering Institute (CMMI®12).

Notas:

* Esta sección es provista exclusivamente para propósitos de información.
* Solo las tareas cubiertas por este Paquete de Despliegue están listadas en cada tabla.
* Las tablas usan la siguiente convención:
* Cobertura Total = F
* Cobertura Parcial = P
* Sin Cobertura = N

## **Matriz de Referencia ISO 9001**

|  |  |  |  |
| --- | --- | --- | --- |
| **Título de la Tarea y Paso** | **Cobertura**  **F/P/N** | **Cláusula de ISO 9001** | **Comentarios** |
| IS.2.2 Documentar o actualizar la Especificación de Requerimientos. | P | 7.3.2 Entradas del diseño y desarrollo  d) Otros requerimientos esenciales para el diseño y desarrollo. |  |
| Sub-tarea: Definir Estándares de Construcción. |  |  | En la mayoría de casos, ellos no son especificados por los clientes, pero son esenciales para algunos componentes. |
| IS.4.1 Asignar tareas a los miembros del Equipo de Trabajo asociadas a su rol, de acuerdo al *Plan de Proyecto* actual. | P | 7.3.1 Planeamiento de diseño y desarrollo | Solo incluye las tareas de comunicación. |
| IS.4.3 Construir o actualizar los *Componentes de Software* basado en la parte detallada en el *Diseño de Software*. | P | 7.3.3 Salidas del diseño y desarrollo  a) Cumplir los requerimientos de entrada para el diseño y desarrollo. |  |
| IS.4.4 Diseñar o actualizar los casos de pruebas unitarias y aplicarlos para verificar que los Componentes de Software implementan las partes detalladas en el *Diseño de Software.* | P | 7.3.4 Revisión del diseño y desarrollo  a) para evaluar la habilidad de los resultados de diseño y desarrollo para cumplir requerimientos, y  b) para identificar cualquier problema y proponer acciones necesarias. |  |
| IS.4.5 Corregir los defectos encontrados hasta lograr la prueba unitaria exitosa (logrando el criterio de salida). | P | 7.3.4 Revisión del diseño y desarrollo  a) para evaluar la habilidad de los resultados de diseño y desarrollo para atender requerimientos, y  b) para identificar cualquier problema y proponer acciones necesarias. |  |
| IS.4.6 Actualizar el *Registro de Trazabilidad* incorporando los *Componentes de Software* construidos o modificados. | P | 7.3.7 Gestionar los cambios de diseño y desarrollo. |  |

Matriz de Referencia **ISO/IEC 12207**

|  |  |  |  |
| --- | --- | --- | --- |
| **Título de la Tarea y Paso** | **Cobertura**  **F/P/N** | **Cláusula de ISO/IEC 12207** | **Comentarios** |
| IS.2.2 Documentar o actualizar la Especificación de Requerimientos. | P | 7.1.2.3.1 Análisis de requerimientos de Software.  7.1.2.3.1.1 El implementador deberá establecer y documentar los requerimientos de software (incluyendo las especificaciones de características de calidad) descritas debajo.  b) Interfaces externas al elemento de software.  k) Requerimientos de mantenimiento de usuario. | a) Solo cubre los requerimientos que afectan la fase de Construcción.  b) Está relacionado con la Interfaz de Usuario, y k) con los Estándares de Construcción. |
| Sub-tarea: Seleccionar el estándar de Interfaz de Usuario. |  | 7.1.4.3.1 Diseño detallado de Software.  7.1.4.3.1.2 El implementador deberá desarrollar y documentar un diseño detallado para las interfaces externas al elemento de software, entre los componentes de software y entre las unidades de software. El diseño detallado de las interfaces deberá permitir codificar sin necesidad de mayor información. | El estándar de Interfaz de Usuario será útil cuando se realice 7.1.4.3.1.2 |
| Sub-tarea: Definir Estándares de Construcción. |  | 7.1.5.3.1 Construcción de Software.  7.1.5.3.1.5 El implementador deberá evaluar código de software y resultados de prueba considerando los criterios listados debajo. Los resultados de las evaluaciones deberán ser documentados.  e) Oportunidad de codificar métodos y estándares utilizados. | Estándares de Construcción serán útiles cuando se realice 7.1.5.3.1.5 |
| IS.4.1 Asignar tareas a los miembros del Equipo de Trabajo asociadas a su rol, de acuerdo al *Plan de Proyecto* actual. | P | 6.3.1.3.3 Activación de proyecto.  6.3.1.3.3.3 El gestor deberá iniciar la implementación y el conjunto de criterios, ejerciendo control sobre el proyecto. | Esta tarea solo activa la fase de Construcción. |
| IS.4.3 Construir o actualizar los *Componentes de Software* basado en la parte detallada en el *Diseño de Software*. | P | 7.1.5.3.1 Construcción de Software.  7.1.5.3.1.1 El implementador deberá desarrollar y documentar lo siguiente:  a) Cada unidad de software y base de datos. |  |
| IS.4.4 Diseñar o actualizar los casos de pruebas unitarias y aplicarlos para verificar que los Componentes de Software implementan las partes detalladas en el *Diseño de Software.* | P | 7.1.5.3.1 Construcción de Software.  7.1.5.3.1.1 El implementador deberá desarrollar y documentar lo siguiente:  b) Procedimientos de prueba y datos para probar cada unidad de software y base de datos. |  |
| IS.4.5 Corregir los defectos encontrados hasta lograr la prueba unitaria exitosa (logrando el criterio de salida). | P | 7.1.5.3.1 Construcción de Software.  7.1.5.3.1.2 El implementador deberá evaluar cada unidad de software y base de datos asegurando que satisface sus requerimientos. Los resultados de las pruebas deberán ser documentados. |  |
| IS.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 implementador deberá evaluar código de software y los resultados de las pruebas considerando la lista de criterios debajo. Los resultados de las evaluaciones deberán ser documentados.  a) Trazabilidad a los requerimientos y al diseño del elemento de software. |  |

## **Matriz de Referencia CMMI**

|  |  |  |  |
| --- | --- | --- | --- |
| **Título de la Tarea y Paso** | **Cobertura**  **F/P/N** | **Objetivo / Práctica de CMMI V1.2** | **Comentarios** |
| IS.2.2 Documentar o actualizar la Especificación de Requerimientos. | P | Desarrollo de Requerimientos (DR)  SG 1 Desarrollar Requerimientos del Cliente  SP 1.2 Desarrollar los Requerimientos del Cliente | Limitaciones para verificación y validación no están cubiertas. |
| Sub-tarea: Seleccionar el estándar de Interfaz de Usuario. |  |  | Los stakeholders de la fase de Construcción pueden proveerlos |
| Sub-tarea: Definir Estándares de Construcción. |  |  | Los stakeholders de la fase de Construcción pueden proveerlos |
| IS.4.1 Asignar tareas a los miembros del Equipo de Trabajo asociadas a su rol, de acuerdo al *Plan de Proyecto* actual. | P | Solución Técnica (ST)  SG 3 Implementar el Diseño del Producto  GP 2.8 Monitorear y Controlar el Proceso | Solo activa la ejecución del plan. |
| IS.4.3 Construir o actualizar los *Componentes de Software* basado en la parte detallada en el *Diseño de Software*. | P | Solución Técnica (ST)  SG 3 Implementar el Diseño del Producto  SP 3.1 Implementar el Diseño | Sub prácticas de cobertura  1. Usar métodos efectivos para implementar los componentes del producto.  y  2. Adherirse a estándares y criterios aplicables. |
| IS.4.4 Diseñar o actualizar los casos de pruebas unitarias y aplicarlos para verificar que los Componentes de Software implementan las partes detalladas en el *Diseño de Software.* | P | Solución Técnica (ST)  SG 3 Implementar el Diseño del Producto  SP 3.1 Implementar el Diseño | Sub prácticas de cobertura  4. Realizar las pruebas unitarias del componente del producto tal como sea apropiado. |
| IS.4.5 Corregir los defectos encontrados hasta lograr la prueba unitaria exitosa (logrando el criterio de salida). | P | Solución Técnica (ST)  SG 3 Implementar el Diseño del Producto  SP 3.1 Implementar el Diseño | Sub prácticas de cobertura  4. Realizar las pruebas unitarias del componente del producto tal como sea apropiado. |
| IS.4.6 Actualizar el *Registro de Trazabilidad* incorporando los *Componentes de Software* construidos o modificados. | P | Gestión de Requerimientos (GREQ)  SG 1 Gestionar Requerimientos  SP 1.4 Gestionar la Trazabilidad Bidireccional de los Requerimientos | Solo cubre la trazabilidad a 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] | Vincent 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](http://shop.ieee.org/ieeestore/), IEEE Computer Society |
| [ISO/IEC 12207] | ISO/IEC 12207:2008 Systems and software engineering – Software life cycle processes. |
| [ISO/IEC TR 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 – Generic Profile Group -Basic Profile |
| [ISO/IEC 24765] | ISO/IEC 24765:2010 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 Despliegue Construcción y Pruebas Unitarias Versión 0.1**  Su retroalimentación nos permitirá mejorar este paquete de despliegue, sus comentarios y sugerencias son bienvenidos. |
| **1. ¿Cuán satisfecho se encuentra con el CONTENIDO de este paquete de despliegue?**   *Muy Satisfecho x Satisfecho*  *Ni Satisfecho ni Insatisfecho*  *Insatisfecho*  *Muy Insatisfecho* |
| **2. ¿La secuencia en que se discuten los temas, es lógica y fácil de seguir?**   *Muy Satisfecho*  *Satisfecho x* *Ni Satisfecho ni Insatisfecho*  *Insatisfecho*  *Muy Insatisfecho* |
| **3. ¿Cuán satisfecho se encontraría con la APARIENCIA/FORMATO de este paquete de despliegue?**   *Muy Satisfecho*  *Satisfecho x* *Ni Satisfecho ni Insatisfecho*  *Insatisfecho*  *Muy Insatisfecho* |
| **4. ¿Cree que se ha incluido algún tema innecesario? (Favor de describir)** |
| **5. ¿Qué temas faltantes le gustaría ver en este paquete? (Favor de describir)**   * Tema propuesto: Desarrollo de software más optimo * Razón fundamental para el nuevo tema |
| **6. ¿Cualquier error en este paquete de despliegue?**   * + Favor de indicar:     - * Descripción del error:       * Ubicación del error (# sección, # figura, # tabla): |
| **7. Otros comentarios:** |
| **8. ¿Recomendaría este Paquete de Despliegue a algún colega de otra PO?**   *Definitivamente*  *Probablemente*  *No está Seguro*  *[x]* *Probablemente No*  *Definitivamente No* |

**Opcional**

* Nombre:
* Dirección de correo electrónico: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_