Obligatorio 1

Nicolas Damiani - 192489  
Gonzalo Kurin - 192488

Ingeniería de Software 2



Tabla de contenido

Introducción 3

Plan de Aseguramiento de Calidad 4

Objetivos 4

Actividades Preventivas 4

Actividades Correctivas 4

Gestión de cambios/nuevos requerimientos 5

Gestion de defectos 5

Plan de Métricas 6

Esfuerzo Real 6

Cobertura de código: 6

Densidad de defectos: 7

Heuristicas de Nielsen 7

Código Duplicado 8

Complejidad Ciclomática 8

Evaluación inicicial de la calidad del proyecto 10

Tamaño inicial del proyecto 10

Calidad de los casos de prueba 10

Calidad de los Requerimientos 10

Calidad del código 12

Usabilidad 12

Cobertura de pruebas unitarias: 14

Complejidad Ciclomática 15

Analisis de Impacto de los Cambios 16

Realización de cambios 17

Actividades preventivas: 17

Revisiones de a pares 17

Utilización de estándares 17

Actividades Correctivas 17

Pruebas unitarias 17

Pruebas Funcionales 19

Control de cambios 19

# Introducción

El propósito de este trabajo es el de elaborar detalladamente un plan de aseguramiento de calidad para el software de DescansoApp, para luego ejecutarlo y asi mantener dicho software de manera correcta y ordenada.

El plan de aseguramiento de calidad cuenta con distintos objetivos, los cuales se detallarán más adelante, para lograr cumplir con dichos objetivos, se dividirá el proyecto en distintas etapas, de esta manera logrando una mayor organización y facilitando el trabajo a realizar.

Como primer punto, se construirá un plan de aseguramiento de calidad aplicado en el proceso de mantenimiento. Al utilizar este plan, se podrá controlar la calidad del producto y de las modificaciones que serán realizadas sobre el mismo. Se van a definir actividades para prevenir errores, asi como tambien actividades correctivas para cuando se encuentre alguno. También se definirán planes de gestion de cambios y defectos, y un plan de métricas.

Luego de terminado el plan de aseguramiento de calidad, se pasará a realizar una evaluación inicial de la calidad del software. De esta evaluación, se podrá conocer el estado actual del software, y de esta manera ver en que aspectos se va a tener que mejorar para lograr elevar la calidad del producto. En esta etapa se van a evaluar aspectos como la cobertura de las pruebas, la calidad de los casos de prueba, el tamaño inicial del producto, la calidad del código y de la documentación, el nivel de usabilidad del sistema, la complejidad ciclomática del código, y el nivel de código duplicado.

Una vez que se termina la evaluación inicial del software y se tiene claro que aspectos del software hay que mejorar o incorporar, se realizará un analisis del impacto de los cambios a incorporar en el sistema. Este analisis permitirá anticiparse a posibles problemas que puedan surgir a partir de la implementación de los cambios, y de esta manera escoger la solución mas adecuada para codificar nuevas funcionalidades. Se van a priorizar y categorizar los diferentes cambios a realizar, y luego se realizará una estimación del esfuerzo (tiempo[?]) que se necesitará de cada uno de los integrantes del trabajo para poder realizar los cambios solicitados para el software.

Luego, se van a incorporar las nuevas funcionalidades al sistema, y se van a corregir los defectos encontrados previamente. Al mismo tiempo que se realizan los cambios se irá registrando el esfuerzo realizado para cada tarea, y se van a documentar los datos pertinentes que serán definidos en el plan de métricas. A lo largo de toda la realización de esta tarea, se va a tener siempre en cuenta el plan de aseguramiento de calidad, para mantener siempre la calidad del trabajo lo mas elevada posible.

Como último paso, se va a realizar una evaluacion final de la calidad del software, de esta manera, se va a comparar con los datos de la primera evaluación, y así se corroborará si efectivamente se han cumplido o no los objetivos planteados al comienzo del trabajo. Sumado a esto, se realizará una reflexión sobre el proyecto, y se va a definir un proceso de mantenimiento y un plan de aseguramiento de calidad para futuros proyectos.

# Plan de Aseguramiento de Calidad

La utilización de un plan de aseguramiento de calidad en el proyecto permitirá especificar las cualidades deseables del software y como poder evaluarlas para reconocer si verdaderamente estas mismas se hayan cumplido con lo especificado. Este plan establecerá un modo de trabajo, de forma de tener una guía para ir realizando el mantenimiento del software.

## Objetivos

Se plantean los principales objetivos a cumplir:

* Mejorar la calidad general de la aplicación en cuanto a densidad de defectos. Mantener registros documentados de los defectos encontrados y de cómo fueron solucionados.
* Mejorar la usabilidad de aplicación. Realizar mejoras utilizando técnicas conocidas de usabilidad para poder crear una interfaz más atractiva y más comprensible para el usuario.
* Mejorar la calidad de código de la aplicación. Utilizando herramientas y estándares, se desea poder corregir el código para poder facilitar el mantenimiento en un futuro.

Teniendo como referencia estos objetivos, el desarrollo del trabajo y su cumplimiento serán pautados de acuerdo a una las métricas que serán especificadas más adelante en el plan de métricas.

## Actividades Preventivas

Refiere a las actividades que se utilizan como precaución para en un futuro poder evitar y prevenir futuros errores. Para poder cumplir con este punto se utilizaran dos técnicas:

* Utilización del documento de estándares y buenas prácticas de codificación Java. Utilizando estos estándares, luego el software puede ser mantenido por alguien por fuera del grupo de programadores ya que los mismos son conocidos más universalmente, y no necesariamente conocer las prácticas del entorno que maneja este software. Además el tener un código prolijo puede evitar errores.
* Revisiones. Las revisiones permiten detectar defectos a nivel estático, enfocadnos en su estructura y contenido. El método a utilizar para las revisiones serán las revisiones de a pares, considerando el tamaño del proyecto y el tiempo a proyectar del mismo, podemos considerar que estas revisiones son las más adecuadas. A su vez para poder implementar las revisiones se utilizara la Java Code Inspection Checklist (Anexo 1).

## Actividades Correctivas

Las actividades correctivas son aquellas que detectan fallos o disconformidades en el software y terminan corrigiéndose para el correcto funcionamiento. En referencia al proyecto se modificaran las pruebas unitarias existentes para que estas representen el correcto funcionamiento de los métodos. También se implementaran la cantidad de pruebas unitarias que parezcan pertinentes para poder satisfacer las clases del dominio. Las pruebas unitarias se probaran con la herramienta JUnit. Tambien se incoporaran pruebas funcionales que verifiquen el correcto funcionamiento de las nuevas funcionalidades del proyecto. Las mismas serán verificadas mediante casos de prueba.

## Gestión de cambios/nuevos requerimientos

Es importante tener un plan para definir la realización de cambios en el sistema. Asi mismo tener bien especificado la importancia de cada uno para poder enfocar las actividades a realizar. La metodología para esta gestión será de la siguiente manera:

1. Definir posible cambio a realizar
2. Registrar petición de cambio
3. Categorizar y priozar cambios de acuerdo a su importancia
4. Analizar y determinar, de acuerdo a factores como tiempo, costo e importancia, si implementar cambios o no
5. Realizar cambio
6. Ejecutar pruebas. Al ejecutar las pruebas es importante verificar que el cambio se haya realizado de manera satisfactoria, como también poder comprobar que el resto del proyecto siga funcionando correctamente.
7. Respaldar cambio. Una vez comprobado que el cambio se haya implementado correctamente, respaldarlo correctamente mediante un commit, utilizando la herramienta Git.

## Detección de defectos funcionales

En relación a la detección de defectos, para poder tener un buen aseguramiento de calidad, es de suma importancia tener definida una estrategia de detección de defectos con el objetivo de probar el sistema y corregirlo de acuerdo a estas métricas establecidas. A partir de este planteo y de los objetivos estipulados, se detalla un plan para poder detectar los defectos y poder corregirlos, principalmente para poder tener una trazabilidad de los defectos corregidos y no corregidos en relación a las metas y pruebas establecidas. Este plan además ayuda en la categorización y priorización de los defectos, para poder analizar el costo-tiempo de corregir un defecto de la manera más efectiva.

La detección de defectos se estipulará a partir de tres fuentes relevantes:

* Reporte de Defectos de la documentación inicial del proyecto
* Pruebas funcionales de la documentación inicial del proyecto
* Pruebas funcionales que parezcan pertinentes generadas propiamente de acuerdo a los casos de uso de la documentación inicial y también a partir de nuevas funcionalidades agregadas.

A partir de las distintas fuentes se recopilaran los distintos defectos encontrados y se gestionaran de manera adecuada, fundamentando la corrección o no de los mismos.

## Gestión de defectos funcionales

Al igual que los cambios, es importante estipular una estrategia para poder registrar y corregir los defectos que se van encontrando y que van surgiendo durante el desarrollo del proyecto. Detalles de plan:

1. Se identifica el defecto
2. Se registra detalladamente el defecto
3. Analizar si cambio es viable con respecto a costo-beneficio
4. En caso de aprobar el cambio se siguen los pasos 5 a 7 de gestión de cambios. Caso contrario, se documenta que no se pudo realizar.

En el trabajo se realizará una tabla donde se tendrán que completar los siguientes campos:

* Numero: sirve para poder idenfificar el defecto
* Nombre: breve sentencia del problema a resolver
* Descripcion: Detalle mas especifico del defecto
* Prioridad: Alta, media o baja. Esto se define para poder organizar la importancia de los defectos de acuerdo al costo-beneficio.
* Caso de Prueba: Referencia del código de caso de prueba del cual se encuentra el defecto.
* Resultado: Se indica si efectivamente el defecto fue resuelto. Si no pudo corregirse, se explica brevemente la razon.

## Plan de Métricas

(TODO: relacionarlo con los objetivos)

### Esfuerzo Real

Metrica que relaciona la cantidad de horas necesarias para poder cumplir con lo solicitado para el mantenimiento del sistema. Primero se estimará el esfuerzo requerido y luego se comparará con el esfuerzo real consumido.

|  |  |
| --- | --- |
| **Nombre** | Esfuerzo del Proyecto |
| **Información**  **que brinda** | Esfuerzo dedicado en el cumplimiento de lo solicitado |
| **Entidad** | Proyecto |
| **Unidad** | Horas |
| **Atributos a recolectar** | Nombre del proyecto Fecha  Cantidad de recursos  Esfuerzo dedicado |
| **Fórmula** | Esfuerzo total = ∑ esfuerzo por recurso |

|  |  |  |
| --- | --- | --- |
|  | **Método** | **Frecuencia** |
| **Recolección** | Manual | Diaria |
| **Análisis** | Comparación con el esfuerzo  estimado | Al finalizar el proyecto |

### Cobertura de código:

Medir la cobertura del código del proyecto nos es útil para luego poder realizar una comparación con la cobertura de código final. De esta manera, se va a notar si hubo o no, un incremento en la calidad del software del proyecto.

|  |  |
| --- | --- |
| **Nombre** | Cobertura de código |
| **Información**  **que brinda** | Indica qué porcentaje de código del dominio se está probando |
| **Entidad** | Proyecto |
| **Unidad** | Porcentaje |
| **Atributos a**  **recolectar** | - |
| **Fórmula** | Resuelto por la herramienta utilizada para la recolección |

|  |  |  |
| --- | --- | --- |
|  | **Método** | **Frecuencia** |
| **Recolección** | JaCoCo | Al comienzo y al final |
| **Análisis** | Comparación con la  cobertura de código inicial | Al finalizar el proyecto |

### Densidad de defectos:

La densidad de defectos es una métrica de calidad que permite conocer la densidad de defectos por línea de código de un proyecto. Normalmente no son de gran utilidad para comparar proyectos, ya que las líneas de código varían mucho según el lenguaje que se utiliza para codificar un proyecto, por eso no es productivo comparar la densidad de defectos entre dos proyectos codificados en distintos lenguajes. Sin embargo, en este trabajo se utilizará la métrica para comparar la densidad de defectos inicial del proyecto, con la densidad de defectos final, luego de realizar las mejoras. Al trabajar sobre el mismo proyecto, y por consecuencia, sobre el mismo lenguaje, no se presenta el conflicto mencionado anteriormente.

|  |  |
| --- | --- |
| **Nombre** | Densidad de defectos |
| **Información**  **que brinda** | Cantidad de defectos encontrados por línea de código en Java |
| **Entidad** | Proyecto |
| **Unidad** | Defectos/LOC |
| **Atributos a**  **recolectar** | Cantidad de defectos  Cantidad de líneas de código |
| **Fórmula** | Cantidad de defectos/LOC |

|  |  |  |
| --- | --- | --- |
|  | **Método** | **Frecuencia** |
| **Recolección** | Defectos -> FindBugs  Loc -> Simple Code Metrics | Al comienzo y al final |
| **Análisis** | Comparación con la densidad  de defectos inicial | Al finalizar el proyecto |

### Heuristicas de Nielsen

Las heurísticas de nielsen es un concepto introducido por Jacob Nielsen que se basa en un conjunto de 10 reglas establecidas que permiten evaluar de manera objetiva la interfaz de usuario. Para poder utilizar correctamente las heurísticas y poder realizar una evaluación de mejora de interfaz, primero se evaluara el sistema inicial y luego se realizara el mismo análisis comparando todas las heurísticas. Para esto se realizara una tabla con que contendrá todas las heurísticas y se indicará el cumplimiento de cada una de ellas con respecto a tres categorías.

* Se cumple: Para que la heurística se cumpla tiene que ser aplicada en todo el software.
* Se cumple parcialmente: La heurística fue aplicada en algunos casos del software.
* No se cumple: no aparecen rastros de que la heurística se cumpla en alguna parte del software.

Para poder realizar el análisis de una manera mas cuantificable se le asignara un puntaje a cada heurística, de la siguiente manera:

* Se cumple: 10 puntos
* Se cumple parcialmente: 5 puntos
* No se cumple : 0 puntos

|  |  |
| --- | --- |
| **Nombre** | Nivel de usabilidad del sistema |
| **Información**  **que brinda** | Cantidad de heurísticas de Nielsen que respeta el software |
| **Entidad** | Proyecto |
| **Unidad** | Porcentaje |
| **Atributos a**  **recolectar** | Cumplimiento de las heurísticas |
| **Fórmula** | Usabilidad = ∑ puntaje de cada heurística |

|  |  |  |
| --- | --- | --- |
|  | **Método** | **Frecuencia** |
| **Recolección** | Manual | Al comienzo y al final |
| **Análisis** | Comparación con el porcentaje inicial de  heurísticas respetadas | Al finalizar el proyecto |

### Código Duplicado

Dentro de los lineamientos de Clean Code, uno de los mas importantes es la prevención de codigo de duplicado. El tener codigo duplicado tiene muchas desventajas. Primero que al tener dos códigos iguales, muchas veces puede ser confuso para buscar ciertas partes de codigo. Tambien al utilizar dos veces el mismo codigo, los cambios de uno se tendrían que implementar en el otro, por lo tanto puede llegar a errores en caso de no realizarse. Finalmente la redundancia de codigo implica que el software tenga mas tamaño de manera innecesaria.

|  |  |
| --- | --- |
| **Nombre** | Porcentaje de código duplicado |
| **Información**  **que brinda** | Cantidad de código duplicado existente en el software. |
| **Entidad** | Proyecto |
| **Unidad** | Porcentaje |
| **Atributos a**  **recolectar** | Código duplicado |
| **Fórmula** | - |

|  |  |  |
| --- | --- | --- |
|  | **Método** | **Frecuencia** |
| **Recolección** | SonarQube | Al comienzo y al final |
| **Análisis** | Comparación con el porcentaje de código  duplicado inicial | Al finalizar el proyecto |

### Complejidad Ciclomática

La complejidad cilcomatica es una métrica introducida por Thomas McCabe, la cual intenta de medir de manera simple y cuantificable la complejidad del código. Cuando hablamos de complejidad de codigo, nos referimos específicamente a la mantenibilidad del mismo. Esta herramienta permite detectar distintos caminos por los cuales un proceso puede tomar, midiendo en cada uno la probabilidad de fallo y/o el esfuerzo requerido para probar todos los mismos. Si tenemos un codigo y una lógica compleja, mas difícil será su comprensión, resultando en una difícil mantenibilidad.

Los valores de referencia de según McCabe son los siguientes:

|  |  |
| --- | --- |
| Complejidad Ciclomática | Riesgo |
| <=10 | Métodos sencillos sin mucho riesgo |
| >10 & <=20 | Métodos medianamente complejos con riesgos moderados |
| >20 & <=50 | Métodos complejos con alto riesgo |
| >50 | Métodos inestables de altísimo riesgo |

Una vez terminado el proyecto, estos valores van a servir de referencia para poder evaluar el código del proyecto de una manera cuantificable.

|  |  |
| --- | --- |
| **Nombre** | Complejidad Ciclomática |
| **Información**  **que brinda** | Cantidad de caminos independientes en una fracción de código.  Habla de la complejidad del mismo. |
| **Entidad** | Proyecto |
| **Unidad** | - |
| **Atributos a**  **recolectar** | - |
| **Fórmula** | - |

|  |  |  |
| --- | --- | --- |
|  | **Método** | **Frecuencia** |
| **Recolección** | Simple Code Metrics | Al comienzo y al final |
| **Análisis** | Comparación con la  complejidad ciclomática inicial | Final del proyecto |

# Evaluación inicial de la calidad del proyecto

## Tamaño inicial del proyecto

Para poder medir el tamaño de un proyecto de manera cuantificable se utilizará la herramienta Simple Code Metrics, la cual nos permitirá tener una noción mas especifica del tamaño del proyecto que estamos trabajando.

La herramienta se basa en medir la cantidad de líneas de código que el proyecto contiene, como también brindar estadísticas específicas, por ejemplo cantidad de clases, métodos, etc.

Al ejecutar la herramienta se obtuvieron los siguientes resultados:

Cantidad de líneas de código (LOC): 6056

Cantidad de clases: 37

Cantidad de metodos: 285

Más allá de la cantidad de clases y métodos, este dato no es muy relevante ya que una clase y un método pueden tener muchas líneas de código, y esta estadística no asemeja la realidad. Yendo a un caso extremo una clase podría contener un programa entero.

Se podría decir que la cantidad de líneas de codigo es el dato mas relevante, resultando en 6056. Este dato nos proporciona la información de que este proyecto es relativamente pequeño. Un proyecto puede ser pequeño lo cual puede resultar mas fácil de poder encontrar partes o secciones del codigo. Sin embargo el codigo puede ser mas pequeño pero menos claro, por lo tanto su mantenibilidad puede resultar mucho mas compleja. Esta herramienta no nos brinda ese tipo de información.

## Calidad de los casos de prueba

(TODO:)

## Calidad de los Requerimientos

(TODO:)

Para analizar la calidad de los requerimientos funcionales, se va a seguir el estándar IEEE 830. En este documento se especifican una serie de 8 características que deben cumplir los requerimientos funcionales y no funcionales para que se consideren correctamente especificados. A continuación una lista con una breve descripción de cada característica.

¿El requerimiento es…?

1. Correcto: sí y solo sí, cada requisito especificado es un requisito que el software debe cumplir.
2. No ambiguo: sí y solo sí, cada requisito especificado tiene solo una interpretación.
3. Completo: sí y solo sí, todos los requisitos son significativos, se contemplan valores de entrada tanto válidos como inválidos, y se definen valores junto a sus unidades de medida.
4. Consistente: sí y solo sí, no se contradice a sí mismo, es decir, ningún subconjunto de requisitos se contradicen.
5. Jerarquizado de acuerdo a la importancia y/o estabilidad: si cada requisito tiene un identificador que indique la importancia o estabilidad del requisito.
6. Verificable: sí y solo sí existe un proceso finito de costo-efectivo con el cual una persona o una máquina puede verificar que el producto de software cumple con el requisito.
7. Modificable: sí y solo sí, su estructura y estilo son tales que, cualquier cambio a los requisitos se puede hacer fácil, completa y consistentemente sin perder la estructura y el estilo.
8. Rastreable: si el origen de cada uno de sus requisitos es claro y si se facilita la referencia de cada requisito en el desarrollo futuro o mejora de la documentación.

A continuación se listarán los requerimientos indicando si cumplen o no con cada característica, también se tendrán en cuenta los casos de uso, en los cuales se especifica información adicional para los requerimientos. Como la lista de requerimientos es extensa, se realizará una categorización para facilitar el análisis de los mismos.

Requerimiento correcto: Si cumple con 7 o más de las características mencionadas.

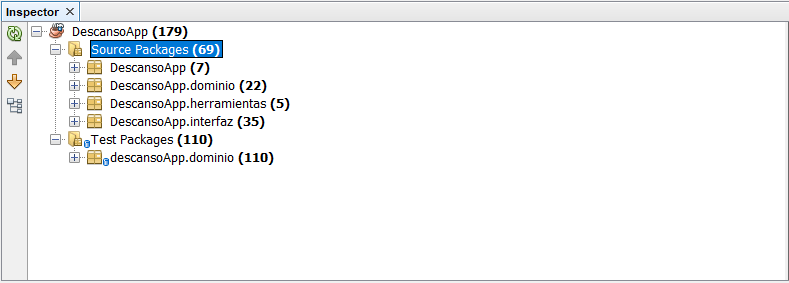
Requerimiento parcialmente correcto: Si cumple con más de 2 de las características mencionadas.

Requerimiento incorrecto: Si cumple con 2 o menos de las características mencionadas.

**PARECE QUE TODOS DAN BIEN, CONFIRMAR CON OTROS.**

## Calidad del código

La calidad del código será evaluada según la cantidad de defectos por línea de código que se encuentren. Para ello se utilizará el plugin FindBugs que, combinado con la métrica de tamaño recogida anteriormente, dará una idea de la densidad de defectos en el código. Los resultados son los siguientes:



Se observa que la densidad de defectos es muy baja, lo cual es un buen resultado. Una alta densidad de defectos implica un código más propenso a errores y a causar problemas al momento de mantenerlo. Durante el desarrollo del trabajo, se intentará reducir este resultado para continuar mejorando la calidad del producto.

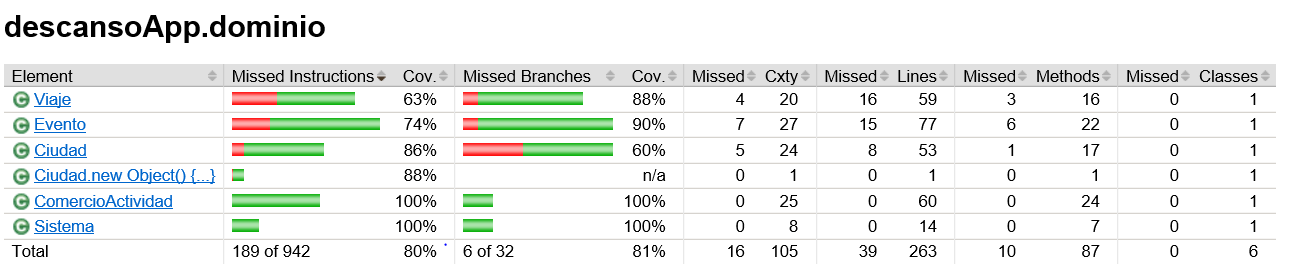
## Usabilidad

Recurriendo a las heurísticas de Nielsen, se evaluara la usabilidad del sistema como fue estipulado previamente. Considerando los pocos recursos, la usabilidad será evaluada por los miembros del equipo. A continuación, la tabla de heurísticas para la evaluación inicial del sistema:

|  |  |
| --- | --- |
| **Heurística** | **Estado** |
| Visibilidad del estado del sistema | Se cumple parcialmente |
| Utilizar el lenguaje de los usuarios | Se cumple |
| Control y libertad para el usuario | Se cumple parcialmente |
| Consistencia y cumplimiento de estándares | Se cumple parcialmente |
| Prevención de errores | Se cumple parcialmente |
| Minimizar la carga de la memoria del usuario | Se cumple |
| Flexibilidad y eficiencia de uso | Se cumple |
| Diálogos estéticos y diseño minimalista | No se cumple |
| Ayudar a los usuarios a reconocer, diagnosticar y  recuperarse de los errores | Se cumple |
| Ayuda y documentación | No se cumple |
|  |  |
| Puntaje total: | 60 |

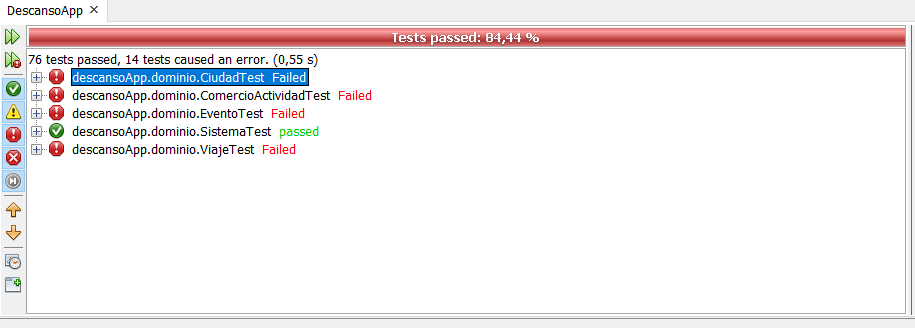
1. Visibilidad del estado del sistema: Parcial
   1. Se muestra que el usuario no tiene viajes asignados. Sin embargo también aparece un viaje guardado. Se agrega viajes y sigue apareciendo cartel.
   2. Cuando se guarda un nuevo viaje no aparece el estado de viaje guardado, el usuario puede no estar seguro de que su viaje se haya guardado.
   3. No se muestra las ciudades que están guardadas. El usuario tiene que buscar a prueba de acierto y error. Ni siquiera aparecen datos con los ejemplo dados
   4. Aparece que no hay resultados para la búsqueda
2. Utilizar el lenguaje de los usuarios: Cumple
   1. Se utiliza el lenguaje cotidiano y de manera amigable. Se utilizan ejemplos para las búsquedas.
   2. Se utiliza un calendario para elegir la fecha, concepto traido de la realidad
   3. El usuario no necesariamente tiene que conocer el sistema para poder utilizar lo que aparece en pantalla.
3. Control y libertad para el usuario: Parcial
   1. El usuario puede eliminar viajes que haya creado.
   2. En todo momento puede volver al estado inicial.
   3. No hay botón de atrás cuando esta creando un nuevo viaje, tiene que cerrar la ventana.
4. Consistencia y cumplimiento de estándares: Parcial
   1. Nombre del proyecto es distinto a la intención de la app. Nunca se refiere a descanso, es mas orientado a turismo.
   2. Dentro del programa, hay coherencia y consistencia con los términos utilizados. Los conceptos que se introducen son pocos, pero hay coherencia entre ellos
5. Prevención de errores: Parcial
   1. Se previenen errores en los ingresos del usuario. Validaciones de fechas, campos vacios, búsquedas con menos de tres caracteres. Se muestran mensajes de error.
   2. Cuando se intenta borrar un viaje muestra mensaje de alerta.
   3. Si el usuario cierra ventana mientras modifica o crea un viaje no se muestra mensaje de alerta que sus datos no van a ser guardados.
6. Minimizar la carga de la memoria del usuario: Cumple
   1. Para llegar a las distintas partes del programa el usuario no tiene que hacer mas de 3 por lo tanto el usuario no tiene que hacer mucho uso de la memoria para poder acordarse de los pasos.
7. Flexibilidad y eficiencia de uso: Cumple
   1. Es fácil y rápido llevar a cabo cada acción. El usuario las acciones que aparecen las puede realizar de manera rápida. No tiene complejidades que uno no pueda entender.
8. Diálogos estéticos y diseño minimalista: No se cumple
   1. El diseño no es simple ni minimalista. Hay sobrecarga de imágenes de fondo, que mezclan al usuario con los textos y colores elegidos.
   2. No hay una temática de colores definida, es decir, se utilizan muchos colores sin definir un patrón.
   3. Botón de nuevo viaje se encuentra cortado a la mitad, no se termina de leer la frase.
   4. Se superponen mensajes e iconos, por ejemplo con el mensaje “No hay viajes guardados!”
9. Ayudar a los usuarios a reconocer, diagnosticar y recuperarse de los errores: Cumple
   1. En caso de errores, los mensajes son claros y concisos. El usuario puede entender bien su error y poder resolverlo.
10. Ayuda y documentación: No cumple
    1. El sistema no cuenta con ningún tipo de ayuda al usuario que lo permita entender aún más el sistema. Por ejemplo: el Usuario no puede ver ninguna ciudad de ejemplo cuando busca. Todos los resultados obtenidos aparecen como, no hay resultados para su búsqueda.

## Cobertura de pruebas unitarias:

La cobertura de pruebas unitarias es una métrica muy importante ya que nos indica que porcentaje del código se está evaluando a través de las pruebas, así como también que porcentaje del mismo aprueba dichas pruebas. Como originalmente había muchos problemas con los nombres de los paquetes, la cobertura de pruebas daba toda 0%, luego, se hizo un renombre de los paquetes, y se pudo ver los resultados reales de la cobertura. Para realizar este análisis, se utilizó el plugin de Netbeans, JaCoCoverage, el cual dio los siguientes resultados:

Como se puede ver en la imagen, el sistema actualmente cuenta con una cobertura de pruebas unitarias del 80% y una cobertura de ramas del 81%. El nivel de las pruebas unitarias es bueno, pero igual va a ser mejorado para elevar la calidad general del código. La cobertura de las ramificaciones también es bastante alta, pero igualmente se va a intentar incrementar ese porcentaje. Se esperará en este trabajo aumentar los niveles de cobertura de código del dominio para mejorar la calidad del software.

Resultados de las pruebas:



En un principio al ver los resultados se creyó que, como se pasa el 84,44% de las pruebas, el trabajo a realizar en las pruebas unitarias iba a ser poco. Luego, al analizar las pruebas con mayor detalle, se vio que hay muchas pruebas “mentirosas” que parecen estar codificadas correctamente, pero tienen un assert(true) al final, lo cual significa que siempre van a devolver true, aunque no deberían hacerlo. En consecuencia de esto, se deberá realizar el arduo trabajo de revisar todas las pruebas y modificarlas para que realmente prueben lo que tienen que probar.

## Complejidad Ciclomática

Con respecto a la mantenibilidad de codigo, como se menciono anteriormente, se utiliza la métrica de complejidad ciclomatica para ver qué tan mantenible es el mismo. Esta métrica se basa en medir la cantidad de caminos independientes en una fracción de codigo. Ademas, esta métrica posee la ventaja de poder ser utilizada, independientemente del lenguaje del codigo. Según los resultados de Simple Code Metrics, los métodos con mayor complejidad clicomatica son:

|  |  |
| --- | --- |
| **Método** | **Complejidad ciclomática** |
| Informacion::ciudades; | 11 |
| Informacion::averiguarParte2; | 11 |
| Informacion::agregarCAs; | 10 |
| pnlItinerario::fechasIguales; | 6 |
| pnlEvento::lblGuardarMouseClicked; | 6 |

Según la tabla de referencia especificada anteriormente hay solo dos métodos que caerían como medianamente complejos con riesgos moderados. El resto de los métodos se categorizarian como métodos sencillos sin mucho riesgo. Incluso el promedio esta muy por debajo de estos valores.

Promedio: 1.34

Se podría concordar que el proyecto tiene niveles bajos de complejitud ciclomatica, por lo tanto dentro de estas cualidades, el proyecto se encuentra bien desarrollado. Sin embargo el objetivo de esta instancia de proyecto, es mantener o incluso disminuir el nivel de complejitud.

## Detección Inicial de defectos funcionales

La documentación pasada ya contaba con defectos encontrados a partir de las pruebas funcionales previamente estipuladas (Anexo 4). Esta lista de defectos se encontraba priorizada de acuerdo a los criterios de costo-tiempo que se consideró en ese momento.

Sin embargo además de considerar como defectos funcionales los defectos encontrados en la documentación, se considerarán otros defectos que fueron surgiendo a partir de las pruebas funcionales (Anexo 3) previamente definidas.

En la realización de cambios se mostrarán todos los defectos que fueron encontrados, sumados a los defectos que previamente estaban documentados. Estos mismos van a incluir una referencia, para detallar a que prueba funcional pertenecen, creando así una trazabilidad entre cada defecto con su respectiva prueba.

# Analisis de Impacto de los Cambios

Antes de implementar las nuevas funcionalidades, es necesario realizar un análisis del impacto de los cambios. De esta manera se lograra comprender que riesgo implica, y que cantidad de tiempo llevara incorporar cada cambio al sistema actual. Los cambios a realizar son los siguientes:

1. Modificar la aplicación para que permita el ingreso de localidades.
2. Se desea mejorar la usabilidad de la aplicación. La interfaz de usuario debe ser

reformulada para hacer la aplicación más atractiva y fácil de usar.

1. La aplicación puede contener defectos de distinto tipo todavía sin corregir. Se debe

mejorar la calidad general de la aplicación en cuanto a densidad de defectos. Se

deben documentar los defectos encontrados y corregidos.

**Cambio 1:**

*Cambio:* Permitir el ingreso de localidades.

*Impacto:*

* Inicialmente el programa carga las ciudades de manera hard-coded, desde la clase BaseDeDatos, donde la única ciudad encontrada es Fray Bentos. Con esta nueva modificación el programa sería capaz de poder agregar ciudades de manera dinámica al sistema.
* Además de poder agregar ciudades al sistema, se permite el ingreso de alojamientos, actividades, y sitios gastronómicos a cualquier ciudad que el usuario seleccione.
* Los datos de cada ciudad deberán de poder guardarse con persistencia en el sistema.

*Categorización:*

* Poder agregar ciudades de manera dinámica es un cambio perfectivo. Ya que mejora la usabilidad del sistema, permite al usuario poder ingresar mas información que le puede ser pertinente para futuros viajes.

*Prioridad:*

Alta

**Cambio 2:**

*Cambio:* Se desea mejorar la usabilidad de la aplicación. La interfaz de usuario debe ser

reformulada para hacer la aplicación más atractiva y fácil de usar.

*Impacto:*

* Modificaciones de paquetes y rutas tendrán que ser realizadas al proyecto ya que existen muchas asociaciones a imágenes e iconos, a paquetes que no existen en el proyecto. Es importante tener el registro de esta tarea ya que consumirá tiempo poder implementar estos cambios, para poder efectivamente realizar las modificaciones de diseño que se crean convenientes.
* Dentro de la usabilidad del proyecto es importante aclarar el uso de la opción jOptionPane para la muestra de mensajes de estado del sistema. Se va a intentar de agregar la mayor cantidad de referencias de estado, tanto mensajes de acciones correctas hechas por el sistema, como mensajes de error que el usuario puede tener para poder realizar una acción.

*Categorización:*

* Cambiar el nombre de los paquetes y las distintas asociaciones se consideran cambios correctivos porque se realizan a partir de fallas de que surgen dentro de las relaciones del programa. El resto de las operaciones son perfectivos ya que mejoran

*Prioridad:*

Media/Alta

**Cambio 3**

*Cambio:* Se debe mejorar la calidad general de la aplicación en cuanto a densidad de defectos. Se deben documentar los defectos encontrados y corregidos.

* Para mejorar la calidad se corregirán varios defectos en el código del proyecto. Esto se realizará utilizando herramientas estáticas de código que permitan la correcta identificación de los mismos, a la misma vez comparando con el Java Code Inspection List. Por lo visto en el análisis inicial del estado del producto, estos cambios van a impactar en todo el proyecto ya que son varias las clases que contienen alguno de estos defectos. Sin embargo no se espera que impacten en la funcionalidad ya que serán solo correcciones estáticas de código y no cambios en su comportamiento.
* A su vez para poder realizar una mejora funcional de la aplicación, se intentara de corregir los defectos del proyecto a nivel runtime, es decir, estos defectos surgen y se identifican ejecutando el programa. Para realizar esto se emplea un análisis de las pruebas funcionales encontradas en la documentación y se prueban en la aplicación, verificando en cada una la correctitud de las mismas. Más allá de esto se realizaran las correcciones que parezcan pertinentes.
* Cambiar el nombre de los paquetes y las distintas asociaciones se consideran cambios correctivos porque se realizan a partir de fallas de que surgen dentro de las relaciones del programa. El resto de las operaciones son perfectivos ya que mejoran

*Categorización:*

* Corregir defectos estáticos dentro del código, son cambios perfectivos ya que son mejoras que se utilizan para poder realizar un código de manera más profesional, cumpliendo con estándares conocidos. Los defectos encontrados a nivel de sistema son correctivos ya que se realizan a partir de fallas en el sistema.

*Prioridad:*

Alta

# Realización de cambios

Para lograr implementar todos los cambios y funcionalidades solicitadas, se siguió el procedimiento que fue definido previamente en el plan de aseguramiento de calidad. De manera de prueba que se realizaron las actividades propuestas, a continuación se presenta evidencia.

## Actividades preventivas:

### Revisiones de a pares

Como fue especificado en el plan de QA, una de las actividades preventivas a realizar es la de las revisiones de a pares. Para realizarlas se utilizó como guía la Java Code Inspection Checklist, de manera que se tenga una referencia clara de los estándares a seguir. Debido a que este tipo de revisiones no suelen ser realizadas de manera formal, los resultados no fueron documentados. El proceso que se llevó a cabo fue que cada uno de los integrantes del equipo revisara el código del otro utilizando la guía para detectar errores y marcar puntos donde se debía mejorar.

### Utilización de estándares

En la siguiente captura, se puede ver un ejemplo de una porción de código donde se ve claramente que se respetan los estándares de codificación de Java.

(TODO:FOTO PROBANDO QUE SE UTILIZARON ESTANDARES)

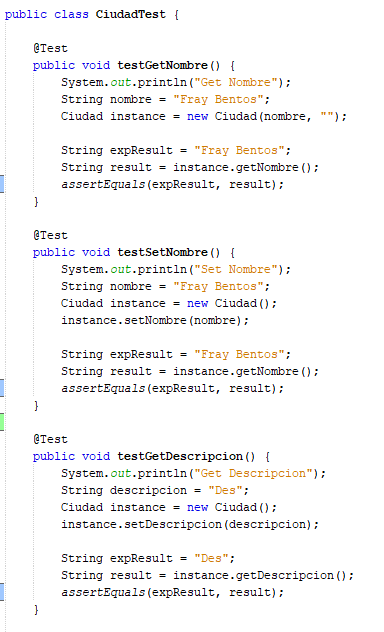
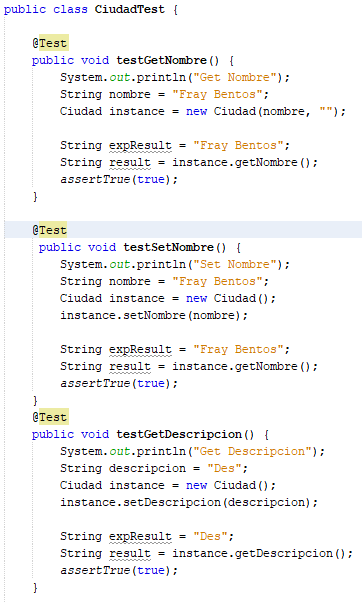
Se respetan convenciones como por ejemplo:

* Todos los nombres de los métodos comienzan con minúscula.
* Todos los nombres de las clases comienzan con mayúscula.
* Se utiliza camel case para nombres de clases o métodos que contengan más de una palabra.
* Llaves de clase y métodos colocadas en la posición correcta.
* Se definen las variables de instancia como privadas, y se accede a ellas mediante los get y set.
* Se utiliza indentación adecuada.
* Se define una variable por línea.
* Los nombres de variables de clase comienzan con minúscula.

## Actividades Correctivas

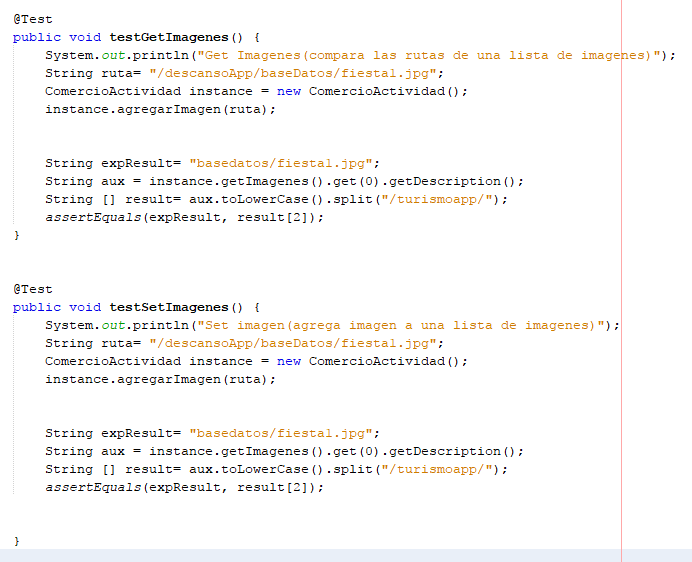
### Pruebas unitarias

Como fue mencionado en la evaluación inicial del proyecto, las pruebas unitarias no estaban codificadas correctamente, ya que habían muchas que devolvían true siempre, aunque no estén bien. En respuesta a eso se revisaron todas las pruebas y se modificaron, a continuación algunos ejemplos de las modificaciones realizadas en las clases de prueba CiudadTest y ComercioActividadTest. Se realizaron cambios de manera análoga en las demás clases de prueba.

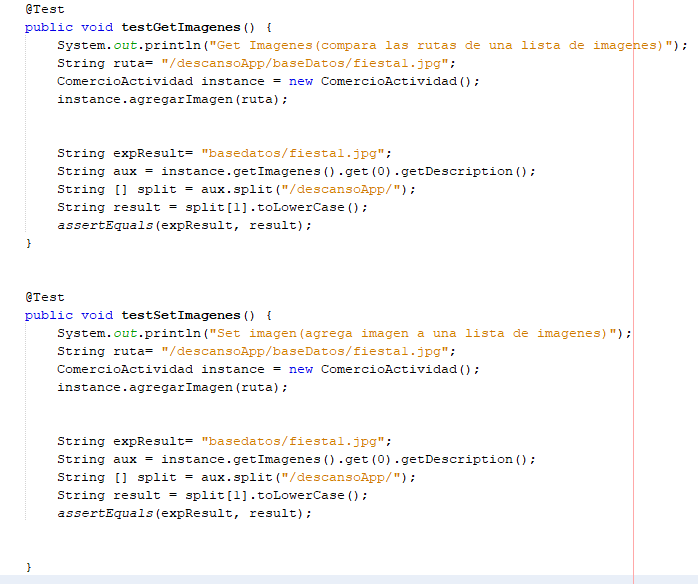
CiudadTest

En este caso se cambiaron los assertTrue(true) por el assert que efectivamente había que realizar comparando el resultado esperado con el obtenido.

ComercioActividadTest

Antes

Despues



En este caso se tuvo que hacer modificaciones más específicas ya que las pruebas previamente implementadas no eran acordes a lo que había que probar.

### Pruebas Funcionales

(TODO:AGREGAR PRUEBAS FUNCIONALES)

## Control de cambios

Como fue especificado en el plan de aseguramiento de calidad, se realizó un detallado control de cambios para cada cambio que fue realizado en el sistema. Además, a medida que se fueron realizando cambios, los mismos se fueron almacenando en el repositorio Git a través de diferentes commits. Como ejemplo del control de cambios, a continuación se litarán los pasos llevados a cabo para el cambio de agregar localidades.

**Paso 1: Definir posible cambio a realizar.**

El cambio de agregar localidades fue pedido explícitamente por el cliente, por lo cual el equipo no tuvo que dedicar tiempo a definir el cambio.

**Paso 2: Registrar petición de cambio.**

Se comunicó a todos los integrantes del equipo del cambio a realizar, para que todos estén al tanto.

**Paso 3: Categorizar y priorizar cambios de acuerdo a su importancia.**

Esto fue realizado en el análisis de impacto de los cambios, se puede ver en la tabla de dicha sección.

**Paso 4: Analizar y determinar, de acuerdo a factores como tiempo, costo e importancia, si implementar cambios o no.**

Como el cambio fue solicitado por el cliente, y fue de las pocas cosas que se solicitaron explícitamente, no fue analizada la posibilidad de no realizar el cambio en ningún momento. Igualmente, se calculó un estimado de tiempo y costo del cambio, y no se presentaron dificultades en ningún caso. El único detalle del cambio a tener en cuenta, que se discutió con el cliente, es el hecho de que las rutas de las imágenes de las localidades apuntan a una carpeta local en donde estamos trabajando, lo que significa que si se desea exportar el proyecto, en un principio, las imágenes de las localidades no se van a encontrar.

**Paso 5: Realizar cambio.**

Como las clases Ciudad y ComercioActividad ya estaban implementadas (aunque con defectos), lo que se tuvo que hacer fueron dos pantallas, AgregarCiudad y AgregarComercioActividad, además de corregir los defectos encontrados en las clases para que puedan correr correctamente. De esta manera, a través de las pantallas se le permite al usuario agregar localidades dinámicamente, a diferencia de antes, que solo se leía la localidad ya existente en base de datos.

**Paso 6: Ejecutar pruebas.**

Se crean pruebas unitarias para los métodos agregados, se modifican las pruebas ya existentes y se corren, de esta manera se verifica el correcto funcionamiento de la nueva funcionalidad del sistema. También se realizaron pruebas funcionales, las cuales fueron definidas y documentadas en la tabla de pruebas funcionales del ítem anterior.

**Paso 7: Respaldar cambio.**

Una vez que se terminó la implementación y prueba del cambio, se realizó un commit en el repositorio local para almacenar los cambios implementados al código. Se puede ver una captura de pantalla de todos los commits en el repositorio en el anexo del documento.

## Control de defectos

A continuación se muestra la tabla de defectos que fueron encontrados a partir de las pruebas funcionales pero que no se encontraban en la tabla de defectos de la documentación.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Número | Nombre | Descripción | Caso de Prueba | Prioridad | Resultado |
| 1 | No se encuentra ciudad Fray Bentos | Cuando se busca la ciudad fray bentos, como se especifica en el caso de uso como ¨playas¨ o ¨Fray Bentos¨. No aparecen coincidencias. | CP07-01 | Alta | Solucionado |
| 2 | No se encuentran alojamientos para dormir en Fray Bentos | Una vez elegida la ciudad Fray Bentos, no aparecen alojamientos cuando se hace click en ¿Dónde Dormir? | CP08-01 | Alta | Solucionado |
| 3 | No aparecen los lugares para comer | Una vez elegida la ciudad fray Bentos, no aparecen alojamientos cuando se hace click en ¿Dónde comer? | CP09-01 | Alta | Solucionado |
| 4 | No aparecen las actividades para hacer | Una vez elegida la ciudad fray Bentos, las actividades para hacer cuando se hace click en ¿Qué hacer? | CP10-01 | Alta | Solucionado |
| 5 | Viajes no ordenados por fecha de inicio | Al hacer click en Mis Viajes, los viajes no aparecen ordenadas por fecha de inicio, sino que por fecha de creación | CP11-01 | Baja | No solucionado |
| 6 | No poder ver itinerario de viaje | Al hacer click en los eventos de un viaje, no aparecen. Aparece que no hay eventos guardados | CP12-01 | Alta | Solucionado |

Posteriormente se detalla la tabla de defectos encontrada en la documentación, especificando cuáles de ellos fueron resueltos.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Número | Nombre | Descripción | Caso de Prueba | Prioridad | Resultado |
| 7 | Nuevo viaje con fechas incorrectas | Al intentar guardar el nuevo viaje, con errors en la fecha, muestra un mensaje de error, pero que no tiene ni titulo ni descripción, simplemente esta vacío. | CP01-06 | Media | Solucionado |
| 8 | Agregar evento con formato incorrecto de fecha | Al intentar guardar el nuevo evento, con error de formato sobre las fechas, el botón no realiza ninguna acción. | CP04-07 | Media | Solucionado |
| 9 | Modificar viaje con fechas incorrectas | Al intentar guardar el viaje modificado, con errores en la fecha, muestra un mensaje de error, pero que no tiene ni titulo ni descripción, simplemente está vacío. | CP02-06 | Media | Solucionado |
| 10 | Modificar un viaje con nombre del viaje ya existente | No realiza ninguna comprobación de si el nombre del viaje a modificar ya existe. Guarda el viaje modificado con el nombre repetido, y quedan ambos viajes en el sistema | CP02-09 | Alta | Solucionado |
| 11 | Agregar evento que sucede al mismo tiempo que otro | No se realiza ninguna comprobación entre las fechas de los eventos agregados y el evento a agregar. Agregando asi el nuevo evento | CP04-09 | Alta | Solucionado |
| 12 | Agregar evento con campos vacios | Al no completar todos los datos a ingresar. Muestra un mensaje con el nombre, fecha inicio y fin del viaje al cual agrego el evento | CP12-01 | Alta | Solucionado |

A modo de ejemplo, se presentará un caso particular de los pasos a seguir para la gestión de defectos. Se realizó, de manera análoga, el mismo procedimiento para todos los defectos.

**Paso 1: Identificación del defecto.**

Al investigar el código, se pudo ver que en la clase BaseDatos existía un string que tenía información acerca de la ciudad Fray Bentos, pero al buscarla en el buscador del sistema no se encontraba y se mostraba un mensaje de error en el sistema. Investigando mas profundamente, se pudo identificar que no se estaban cargando correctamente los datos del string, por lo que efectivamente no se estaba cargando la ciudad al sistema.

**Paso 2: Documentación del defecto.**

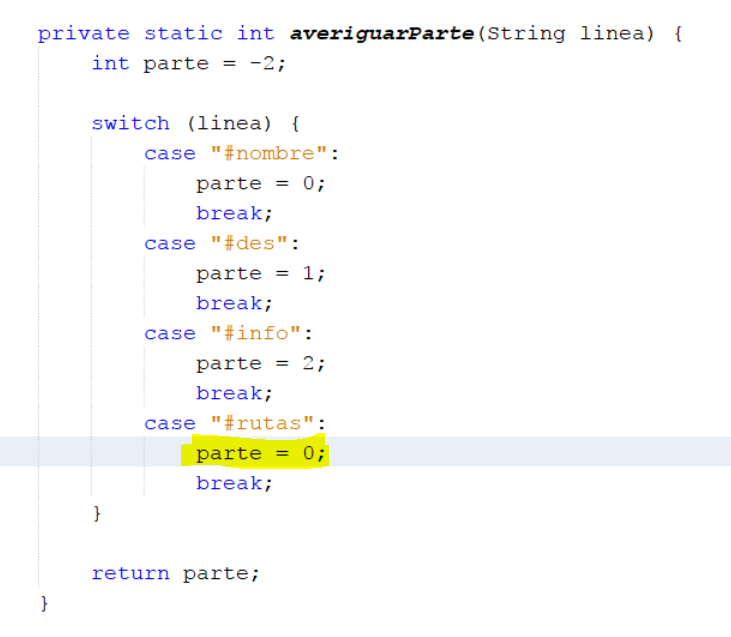
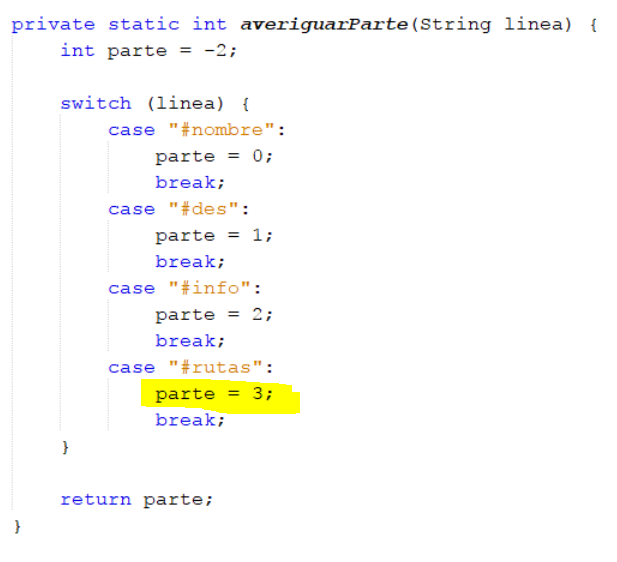
En la tabla de defectos listada anteriormente.

**Paso 3: Decidir si se va a corregir o no.**

El defecto es un defecto de importancia grande, ya que una de las funcionalidades más importantes del sistema es la de que los usuarios busquen lugares para irse de vacaciones, por lo cual, deben poder cargarse correctamente y agregarse nuevas ciudades al sistema. Además de esto, la solución del defecto es una solución sencilla, lo cual significa que el costo de solucionarlo va a ser bajo, pero va a tener un muy alto beneficio frente a la usabilidad del sistema. Por estos motivos se decidió que se va a corregir el cambio.

**Paso 4: Implementación de la solución.**

Se encontró la causa del error en la clase Informacion, más precisamente en el método averiguarParte(). Se encontró que en un switch del método, había un resultado que no tenia sentido, y se cambió (ver capturas).

Antes: Despues:

**Paso 5: Probar el funcionamiento de la solución**

(HACER PRUEBA FUNCIONAL QUE LO PRUEBE Y AGREGAR ALGO)

**Paso 6: Almacenar la solución.**

Se hace un commit al repositorio local para que se guarde la solución en el repositorio remoto al hacer push.

## Registro de métricas

### Esfuerzo real

(TODO:PONER TABLA DE ESFUERZO REAL DE CADA RECURSO)

### Densidad de defectos

Para medir la densidad de defectos después de realizar todos los cambios en el código, se necesita el nuevo tamaño del código, y la nueva cantidad de defectos encontrados.

LOC: INSERTAR

Volviendo a correr la herramienta FindBugs se obtuvo el siguiente resultado:

(FINDBUGS)

(TODO:INSERTAR RESULTADOS DE DENSIDAD DE DEFECTOS FINALES)

### Cobertura de código

Para obtener la cobertura final del código, se volvió a utilizar la herramienta JaCoCo. Como puede verse en la captura, los resultados de cobertura son superiores a los iniciales, se tiene un X% de cobertura de código y un X% en las ramificaciones.

(TODO:PONER RESULTADOS JACOCO FINALES)

### Heurísticas de Nielsen

(TODO:HACER ANALISIS DE HEURÍSTICAS FINAL)

### Código duplicado

(TODO:PONER RESULTADO SONARQUBE FINAL)

### Complejidad ciclomatica

(TODO:PONER RESULTADO SIMPLE CODE METRICS FINAL)

# Evaluación final de calidad

# Reflexión

Al ser la primera vez que se realiza un trabajo de mantenimiento de software, se puede afirmar que efectivamente se recogieron bastantes lecciones de este tipo de procesos.

Una de ellas es la importancia de realizar una evaluación inicial al momento de recibir un proyecto a mantener. En esta instancia, se tiene la oportunidad de identificar el estado del software, y poder entender a qué se va a enfrentar el equipo realizando el trabajo. Con estos resultados se intenta obtener una visión global del proyecto, por lo que se deben utilizar distintas técnicas y herramientas para poder analizar todos los aspectos del software.

Otra lección aprendida es que es de enorme utilidad definir un plan de QA y objetivos. Estos se utilizaron a lo largo de todo el proyecto constantemente, ya que sirvieron como guías para poder llevar a cabo el mantenimiento correcto del software. En este momento se especifican todas las actividades a realizar, para luego simplemente seguir el plan y trabajar de manera ordenada y completa, asegurándose de que no falte nada por realizar.

También se concientizó acerca de la importancia que tiene trabajar con métricas. Las métricas son algo que se conocen desde el comienzo de la carrera, pero que nunca se tuvieron que aplicar a rajatabla al trabajar, por lo que nunca se le dio mucha importancia. En este trabajo fue la primera vez que realmente se trabajó con un plan de métricas claramente definido y eso se vio completamente reflejado a la hora de analizar resultados, donde era muy fácil categorizarlos y trabajar a partir de dichas categorizaciones. Al utilizar las métricas también se pudo cuantificar realmente los objetivos propuestos, para luego determinar si se lograron cumplir o no.

Al ser la primera vez que se tuvo que mantener código de otras personas, se logró ver la real importancia que tiene el escribir código prolijo, o “clean code”. El hecho de tener que leer y entender un código ajeno que no fue hecho de la manera más prolija, nos demostró que realmente hubiese sido mucho más accesible el trabajo si dicho código se hubiese escrito de forma prolija, ya que se hubiese desperdiciado mucho menos tiempo intentando comprender el código.

Otra importante lección recogida a partir del trabajo, es la de la utilidad de la estimación de los esfuerzos. También fue la primera vez que se tuvo que realizar estimaciones de esfuerzos a dedicar, y no fue fácil. Igualmente, el hecho de que se haya realizado dicha estimación, permitió trabajar de una manera mucho más organizada y con tiempos más efectivos. HABLAR DE LA DIFERENCIA DE ESTIMACION DE TIEMPO.

Todas estas lecciones aprendidas van a ser muy útiles al momento de realizar un próximo trabajo de mantenimiento de software, o aun incluso para otro tipo de trabajo que implique un proyecto de software.

A partir de la experiencia ganada en este trabajo, se puede plantear un proceso de mantenimiento y aseguramiento de la calidad de un proyecto, para que pueda ser utilizado a futuro.