**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

Mas alla 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.

**Introduccion de complejidad ciclomatica:**

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 |

**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.

Parte Gonchi

**PLAN DE METRICAS:**

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 |

**EVALUACION INICIAL DE LA CALIDAD DEL PROYECTO:**

Cubrimiento de las pruebas unitarias:

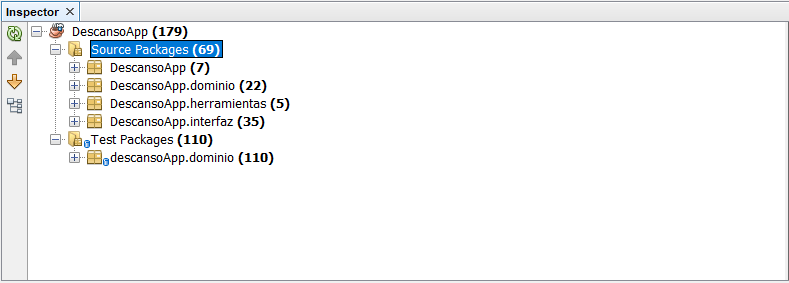
La cobertura de pruebas unitarias es una métrica muy importante ya que nos indica que porcentaje del código está siendo evaluado mediante las pruebas, así como también que porcentaje del mismo aprueba dichas pruebas. Para realizar este análisis, se utilizó el plugin de Netbeans, JaCoCoverage, el cual dio los siguientes resultados:

(RESULTADOS)

Se observa una cobertura de pruebas unitarias del 80% y una cobertura de ramas del 70%. 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 es baja, y sin dudas se intentará incrementar ese porcentaje. Se esperará en este trabajo aumentar los niveles de cobertura de código del dominio para mejorar la calidad del software.

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.

**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 estatico, enfocadonos 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 mas 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.

**Gestion de defectos**

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.
* Resultado: Se indica si efectivamente el defecto fue resuelto. Si no pudo corregirse, se explica brevemente la razon.

**Dentro de plan de métricas…**

**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 |

**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 |

**Usabilidad**

Recurriendo a las heurísticas de Nielsen, se evaluara la usabilidad del sistema como previamente fue estipulado. 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. 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. 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. 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. 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. Parical
   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. 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. 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. 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. Cumple
   1. En caso de errores, los mensajes son claros y concisos. El usuario puede entender bien su error y poder resolverlo.
10. 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.