**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:

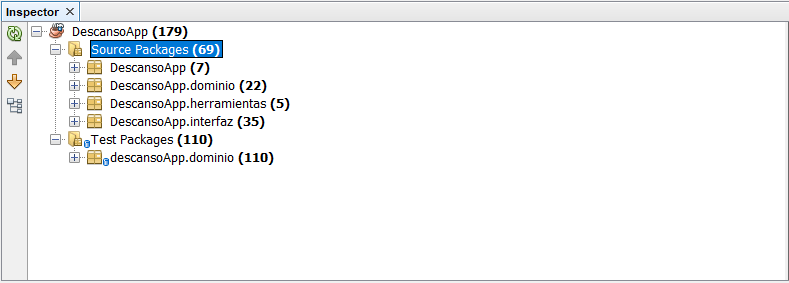
La cobertura de pruebas unitarias es una métrica muy importante ya que nos indica que porcentaje del código está siendo evaluado mediante las pruebas, así como también que porcentaje del mismo aprueba dichas pruebas. Para realizar este análisis, se utilizó el plugin de Netbeans, JaCoCoverage, el cual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