**UNIVERSIDAD ORT URUGUAY**

**Ingeniería de Software 2**  
**Primer Obligatorio**

**Participantes:**

Gabriel Bursztein (195581)  
Fabián Grobert (194738)  
**Docentes:**

Álvaro Ortas

Índice

[Í](#c30e4i6sefm)**[ndice](#c30e4i6sefm)** 2

[1)Introducción](#dse52uvnmi1) 4

[2)Plan de SQA](#cbejr8cm0ro2) 4

[2.1)Objetivo](#br8ln2cneu1u) 4

[2.2)Calidad](#qukr7u2nc7k7) 4

[2.3)Plan de Métricas](#lowstutczbwz) 4

[2.3.1)Cubrimiento de las pruebas unitarias](#wg2mxh8fbz0l) 5

[2.3.2)Cubrimiento de las pruebas funcionales](#gsl4bnnu3o4r) 5

[2.3.3)Calidad de código](#p055wezhvy8p) 5

[2.3.4)Tamaño del sistema](#qokuzmb2w9c) 7

[2.4)Usabilidad](#v0tmtkryffxj) 7

[2.5)Documentación](#p24en4cwuws3) 8

[2.6)Actividades Preventivas](#guf7786qllib) 9

[2.6.1)Revisiones](#n2ctsgvnv0ek) 9

[2.6.2)Auditorías](#rwuf3d26rfso) 9

[2.7)Gestión de cambios](#i2h2xn4olnp4) 9

[2.8)Gestión de defectos.](#r11921h3zedo) 9

[3) Evaluación de calidad inicial](#brxkzdb14txi) 11

[3.1)Descripción del producto](#jfnjhg1z528) 11

[3.2) Plan de métricas](#t0rodz12s5e) 11

[3.2.1) Cubrimiento de las pruebas unitarias](#jbbl4lto4jcp) 11

[AGREGAR TABLA](#qxcfvoecr1qp) 11

[3.2.2) Calidad de código](#xu4nuchobso) 12

[3.2.2.1) Cumplimiento de estándares](#ryuybomot8xx) 12

[3.2.2.2) Código duplicado](#t512fgeb591k) 13

[3.2.2.3) Complejidad del código](#rjtox1jlv1ui) 13

[3.2.3) Tamaño del sistema](#i9ejleracr43) 13

[3.3)Usabilidad](#ts5f4r13h0lx) 13

[4) Análisis de cambios](#se9unv556c8w) 15

[4.1) Cambios](#jjzz09pb70ax) 15

[4.1.1) Nuevas funcionalidades](#k7y90141k64y) 15

[4.1.2) Calidad de código](#df9fubaglkwv) 15

[4.1.3) Usabilidad](#t0ti7osri37y) 16

[4.1.4) Pruebas](#ywys8u3euuqn) 16

[4.2) Impacto de cambios](#pka0t0b1w6ai) 16

1)Introducción

El objetivo de este documento es indicar directivas para poder garantizar la calidad del software en un contexto de mantenimiento del mismo.

Se evaluará la calidad inicial del software y se indicarán diversas formas a través de las cuales se podrá garantizar la calidad de los cambios que se desean realizar. Se evidencian los cambios realizados y se realiza un análisis de calidad posterior a los mismos.

Al final, se llevará a cabo un análisis enfocado en el contraste entre el software inicial y el que resulte de realizarle los cambios deseados.

2)Plan de SQA

2.1)Objetivo

El plan de aseguramiento de calidad está enfocado en guiar mediante una serie de pasos a tener en cuenta a la hora de realizar el mantenimiento del software, teniendo como objetivo mejorar la calidad del producto.

2.2)Calidad

Para que un software cumpla con un estándar de calidad, este debe ser mantenible. extensible en cuanto a desarrollo, eficiente y de fácil uso. Mediante este documento se busca que los requisitos de calidad mencionados se cumplan y que se lleve a cabo un desarrollo de software óptimo.

2.3)Plan de Métricas

El objetivo del plan de métricas consta en medir las diferentes áreas del proyecto. Dentro de ellas se encuentra la cobertura a nivel de pruebas unitarias y funcionales, el tamaño del sistema, la calidad del código, entre otras. Las herramientas que nos resultaron de utilidad fueron: JaCo Coverage(a nivel de cobertura de pruebas), Java hints y SonarQube(mediante la cual analizamos diferentes aspectos del proyecto).

2.3.1)Cubrimiento de las pruebas unitarias

Para lograr un óptimo desarrollo de software, es necesaria la codificación de pruebas unitarias. Son importantes ya que nos ayudan tanto a validar bloques de código, como a encontrar errores en el mismo, antes de integrarlo equívocamente al resto del sistema.

Las pruebas unitarias nos indican claramente donde está el error, ya que estas son minimales, por lo tanto deben tener un único assert.

Se espera que el conjunto de las pruebas de cierta clase cubran todos los caminos posibles, tanto básicos como alternativos, de los métodos de la misma.

Las pruebas unitarias deben ser independientes entre sí, no pueden relacionarse unas con otras. Mediante el nombre de las pruebas se debe poder inferir específicamente qué es lo que se va a probar, por lo tanto este debe ser claro y conciso

2.3.2)Cubrimiento de las pruebas funcionales

Las pruebas funcionales son las que prueban el sistema en su totalidad. Estas pruebas verifican que los requerimientos especificados previamente fueron implementados correctamente. Se realiza una prueba por cada camino posible dentro de la funcionalidad. Estas pruebas se realizan en los casos de uso, mediante los cursos normales y alternativos.

2.3.3)Calidad de código

Cumplimiento de los estándares

Para cumplir con los estándares de calidad es fundamental que el código este escrito de forma prolija y entendible. Esto lleva a que en un futuro este software sea más fácil de entender como de mantener.

Al estar implementando la solución en Java, utilizaremos buenas prácticas como los estándares de codificación de Clean Code. Listamos algunos de los puntos principales:

* Nombres

Los nombres de las distintas clases, métodos, etc. deben ser descriptivos y deben representar su contenido.

* Los nombres de las clases deben ser sustantivos, nunca verbos.
* Los métodos deben ser o contener verbos.

Cada componente tiene sus propias reglas de cómo debe ser su nombre

* Los nombres de las clases van con la primera letra en mayúscula.
* Los nombres de los métodos, atributos y variables todos en minúscula.
* El nombre del paquete con la primer letra en mayúscula.
* Se debe usar camelCase para diferenciar palabras.
* Comentarios

Se debe intentar no escribir comentarios innecesarios o que describan obviedades, de esta manera se evita posibles futuras confusiones. Es correcto utilizarlos cuando se quiere advertir sobre un comportamiento o consecuencia de algo.

* Funciones

Las funciones deben de ser cortas, mantener un mismo nivel de abstracción y hacer una única cosa. Se intenta que funciones que retornan un elemento a la vez, no se comporten como un procedimiento y modifiquen datos internos del sistema. No se debe retornar valores en NULL sino que se debe utilizar las excepciones.

* Parámetros

Se intenta evitar funciones que reciben múltiples parámetros, es recomendable no superar más de 3, aunque es entendible en ciertos casos excederse.

* Principio de responsabilidad unica

Conlleva una gran importancia cuando se quiere desarrollar un software óptimo. Debe respetarse cada sección diferente del programa, de tal manera que tengan una única responsabilidad, por ende un único eje de cambio. De esta manera se reduce el impacto de cambio y el código se hace más mantenible.

* Excepciones

Se recomienda la implementación de excepciones propias para marcar errores efectivamente.

Código duplicado

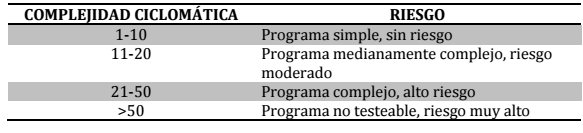
Un aspecto de suma importancia a la hora de desarrollar un software trata acerca de las líneas de código duplicadas. Se debe llevar un control ya que estas aumentan significativamente la complejidad del sistema, dificultan su entendimiento y lo convierten en menos mantenible, ya que si se intenta cambiar algo del sistema, deberá cambiarse repetidas veces en cada línea duplicada. A su vez reduce la eficiencia del sistema ya que se deberá leer repetidas veces la misma línea.

Complejidad del código

El código debe ser comprensible. El Objetivo es que todas las personas que se vean involucradas en el proyecto puedan entender con el menor esfuerzo posible lo que se está programando. A más complejidad en el sistema , el mismo se vuelve menos mantenible. Si la complejidad es menor, esto lo convierte en más entendible, por lo tanto es más propenso a cambiarlo fácilmente.

La complejidad ciclomática se utiliza para hacer referencia a la complejidad que un código posee en sus diferentes métodos. Esta es la cantidad de caminos posibles dentro de un método y determina el máximo de pruebas que se deben realizar para probar cierto módulo de código, para que cada uno de los caminos se ejecuten como mínimo una vez. A su vez, la complejidad ciclomática es independiente del lenguaje en el cual se está implementando la solución.

Nos basaremos en la siguiente tabla para evaluar la complejidad de un programa.



2.3.4)Tamaño del sistema

El tamaño del sistema refiere a la cantidad de líneas de código que posee el sistema y cómo se distribuyen en las diferentes clases. Las lineas de codigo no tienen en cuenta comentarios.

El número de líneas nos indica si hay alguna clase o método muy cargado y que se pueda dividir. Las clases y métodos extensos se vuelven más complejos, se hacen más difíciles de entender y menos mantenibles.

2.4)Usabilidad

El sistema debe ser intuitivo y de fácil uso. Para esto es recomendable tomar en cuenta aplicaciones existentes conocidas para que los usuarios ya tengan ciertos conocimientos dada las las similitudes. Para evaluar la usabilidad se pueden utilizar las heurísticas de usabilidad de Nielsen. Estas nos hacen saber cuando un software tiene mejor usabilidad, las más importantes son:

* Control y libertad del usuario: Dejar a los usuarios decidir qué quieren hacer pero darles una salida(deshacer y rehacer) fácil en caso de que cometan algún error.
* Consistencia y estándares: El usuario debe seguir las normas y convenciones del sistema para entender y evitar confusiones en cuanto a significado de palabras,o acciones del sistema, entre otras.
* Prevención de errores: Hay que llevar un control estricto sobre acciones que pueden provocar errores, o preguntarle al usuario si está seguro de realizar una acción, para ayudar al usuario a no caer en ellos.
* Visibilidad del estado del sistema: Debe existir una retroalimentación entre el sistema y el usuario. El primero deberá informar al segundo acerca de su estado actual. Dicha retroalimentación debe ser realizada en un tiempo razonable.
* Ayudar a los usuarios a reconocer, diagnosticar y recuperarse de los errores: El sistema deberá mostrar mensajes de error sin ambigüedades, se deben entender fácilmente y deben indicando el problema para poder ser más constructivos.

2.5)Documentación

El producto final debe contar con una documentación prolija y clara. Esta debe contener un índice, sus hojas numeradas y el texto justificado.

Debe estar bien diagramada, con títulos y subtítulos, de manera de ayudar al lector a comprender su contenido.

Cada elemento en la documentación debe cumplir con sus atributos de calidad.

* ESRE: Es la documentación vinculada a la especificación de requerimientos. Este documento posee la aplicación para cumplir con lo esperado por parte del cliente. Se divide en un análisis, la identificación de los actores, la especificación de requerimientos junto a sus casos de uso y pruebas funcionales.
* Reporte de estado previo al mantenimiento: Se indicará y comentará acerca del estado actual de la aplicación, se analizarán las métricas a utilizar para reportar defectos.
* Reporte de mantenimiento: La realización del mantenimiento produce cambios y estos se deben de establecer en forma resumida. Contiene el reporte de las métricas y el análisis de los cambios,comparando todo con la versión anterior.
* Plan de calidad: Establece una serie de pasos a seguir para poder asegurar que el software producido es de calidad. Contiene el plan de calidad, el plan de métricas, la gestión de defectos y la gestión de cambios.

2.6)Actividades Preventivas

2.6.1)Revisiones

Al no ser un proyecto amplio y al estar el equipo formado por dos personas, en lo que refiere al desarrollo de las revisiones consideramos que deberíamos utilizar la revisión de pares y la programación de a pares.

* Revisión de pares: Se envía un artefacto a un par, este la revisa y devuelve la lista de los defectos que encontró. Se adecúa a artefactos de bajo riesgo. Lleva poco tiempo y pocos recursos.
* Programación de a pares: Dos personas trabajan juntas para crear el código e implementar una funcionalidad. Funciona como solución para implementar soluciones en tiempo real. Se dividen las tareas, por ejemplo, una piensa en detalles del código mientras la otra en los estándares de la codificación.

Para llevar a cabo el control de calidad a la hora de hacer las revisiones, se utilizarán las métricas que definimos en el plan de aseguramiento de calidad.

2.6.2)Auditorías

Para garantizar que se ha seguido la guía de calidad para los procesos que se aplicaron realizaremos auditorías por parte del equipo de aseguramiento de calidad.

2.7)Gestión de cambios

Para gestionar los cambios que puedan surgir,consideramos apropiado evaluar una serie de indicadores para cada cambio, con el objetivo de que el análisis gane profundidad.

Para comenzar, cuestionamos la necesidad del cambio y los riesgos que puede conllevar su implementación. Luego, evaluaremos su facilidad de implementar y la forma en la que impactaría en el resto del proyecto. Una vez que decididos estos puntos, se determinará la prioridad, el encargado y los recursos necesarios.

2.8)Gestión de defectos.

A la hora de gestionar los defectos trabajaremos de la siguiente forma.

Realizaremos una tabla donde se listen los defectos que se encontraron y le indicaremos una prioridad(baja media o alta). A su vez la tabla contendrá una columna indicando el estado del defecto y la fecha de su corrección para facilitar la identificación de si fueron realizados o no a la hora de generar cambios.

Luego de la etapa de identificación, se corregirán los de prioridad media y alta para completar la tabla que se creó. Los de prioridad baja no se tomarán en cuenta a menos que se disponga de tiempo y recursos una vez completos los de mayor prioridad.

3) Evaluación de calidad inicial

3.1)Descripción del producto

El producto es un el famoso juego de mesa “Monopoly” adaptado a formato digital. Permite que entre 1 y 8 personas jueguen en la misma partida. Respecto a el juego original, cuenta con un interfaz mejorado y con unas nuevas e inéditas funcionalidades.

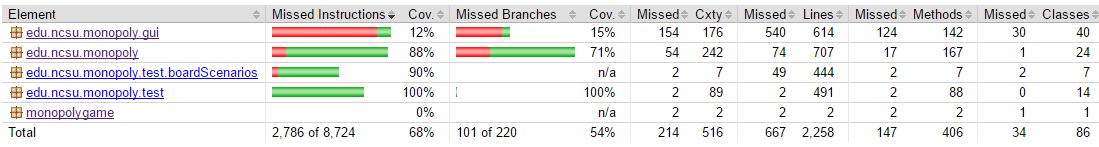
3.2) Plan de métricas

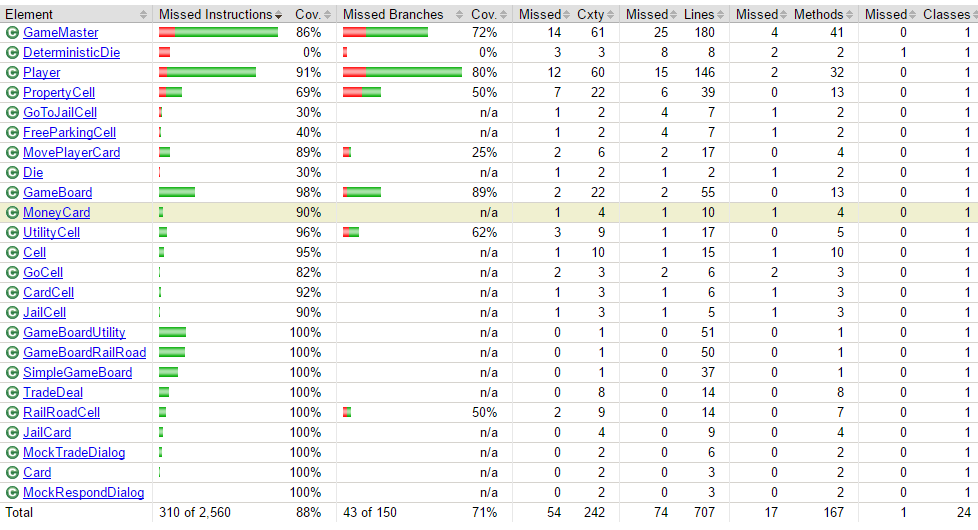
Para comenzar la evaluación del producto y su mantenimiento, realizaremos un análisis completo de su calidad para entender mejor qué áreas hay que mejorar. También estableceremos los defectos y cambios a realizar. Para este análisis, seguiremos las métricas que establecimos en el plan de calidad. Volveremos a hacer el mismo análisis de las métricas una vez finalizado el proceso de cambio y mantenimiento.

3.2.1) Cubrimiento de las pruebas unitarias

Para hacer el análisis de las pruebas unitarias utilizamos la herramienta Java-

CodeCoverage, que nos da un informe sobre la cobertura de las pruebas.





En cuanto a la cobertura, podemos ver que se realizó un total de XX pruebas, apenas cubriendo un 88% del código. Hay algunos casos que no han sido probados lo que nos da a entender que se necesita dedicar un poco de tiempo en las pruebas. Podemos concluir que faltó trabajo en cuanto a pruebas unitarias.

3.2.2) Calidad de código

#### 3.2.2.1) Cumplimiento de estándares

En general el código es prolijo y cumple con los estándares. A continuación se indica un análisis de las distintas áreas analizadas que comprometen al código en cuanto a calidad.

* Nombres:

Los nombres de las distintas clases son representativos de lo que las mismas representan. Por otro lado, las variables y los métodos tienen nombres correctos, que representan correctamente al objeto, en caso de las variables y en el caso de los métodos a la acción que realizan. También en el caso de los métodos, se muestra de forma clara a través del nombre cuando el método retorna un booleano. Por ejemplo, el método isInJail().

* **Diferencia en nombres:**

Se realizó correctamente la forma en la que se escribió nombres de los distintos elementos. Los nombres de las clases comienzan con mayúscula, los de los métodos atributos y variables en minúscula(a no ser por CamelCase). Acerca de los paquetes. estos deben comenzar con mayúscula, al igual que las clases, pero están escritos en minúscula.

* **Comentarios:**

No se encontraron gran cantidad de comentarios a lo largo del proyecto, sin embargo se encontró un comentario explicando el comportamiento de un método puntual y también se encontró código comentado, lo cual es considerado un error.

* Funciones:

Las funciones en general no superan más de las 10 líneas de código, lo cual es considerado una buena práctica, y suelen explicarse por sí solas.

* Pará**metros**:

No se encontraron métodos que reciban demasiados parámetros.

* **Entendimiento del código:**

No se encontraron ifs cargados de muchas condiciones, 2 o 3 como mucho. En algunos casos pudimos ver líneas de código compuestas únicamente por un“;” y también “return” al final de métodos void.

* **Utilización de excepciones:**

Consideramos que se utilizó un número correcto de excepciones, pero en ciertos casos, no están del todo bien manejadas. Por ejemplo el caso que pide ingresar el número de jugadores, si se ingresa una letra se muestran 2 mensajes de error. El primero pide que sea un número y el segundo que sea un número entre 1 y 8.

* **Lógica:**

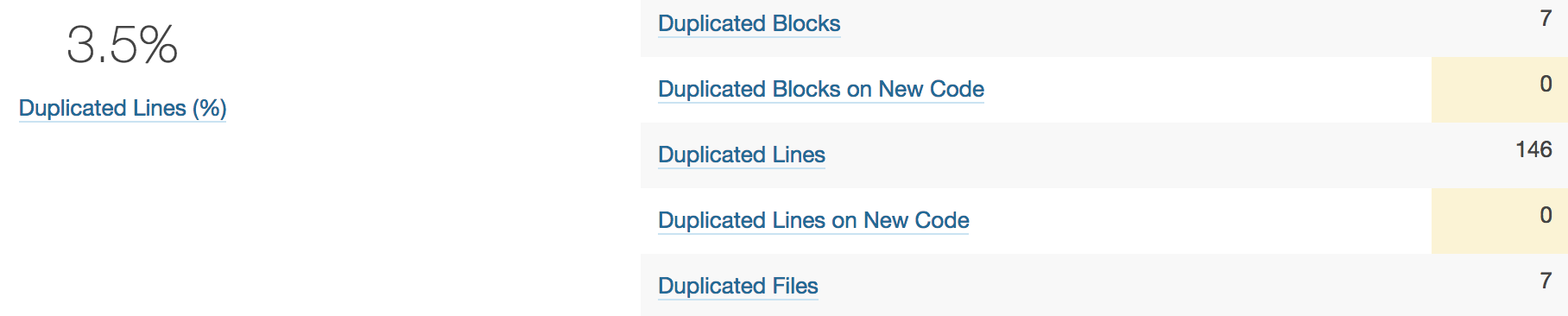
Creemos que las funcionalidades se implementaron correctamente. Tampoco encontramos parte de lógica del programa en la interfaz gráfica, lo cual es está bien.

Encontramos algunos métodos, como purchaseHouse, que creemos deberían ir en gameMaster PREGUNTAR.

#### 3.2.2.2) Código duplicado

En este proyecto podemos ver una cantidad alta de codigo duplicado. La repeticion del codigo hace que este sea mas dificil de mantener y mas propenso a errores.

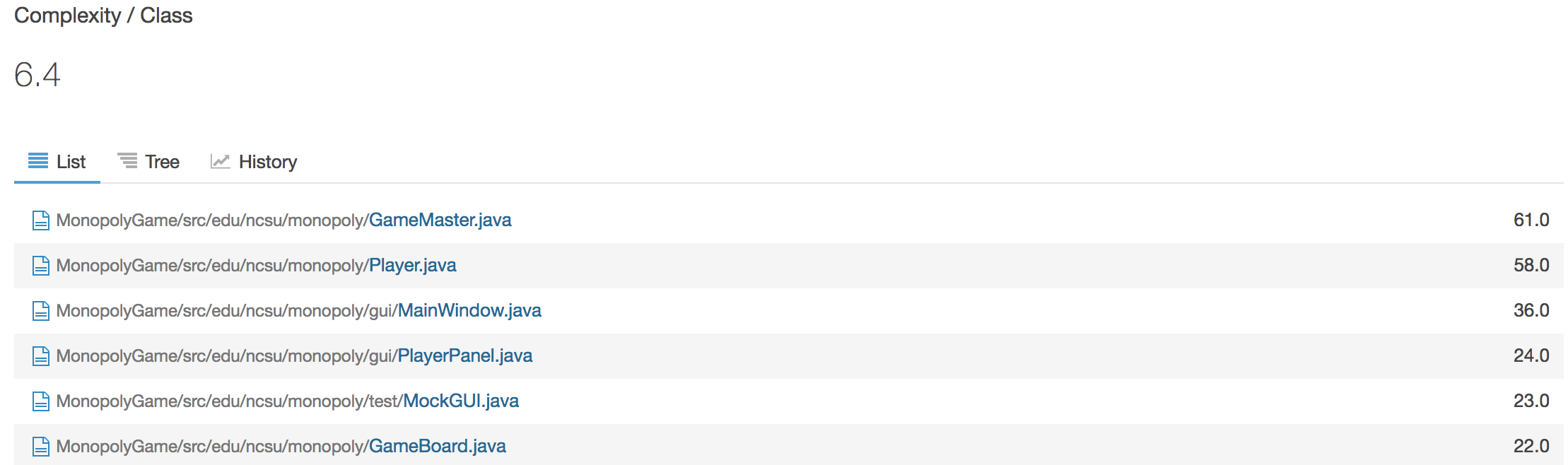
(Agregar que toma sonarqube como código duplicado)



La mayor repetición se encuentra en la clase SimpleGameBoard, con 62% de código duplicado. Es de suma importancia reducir este número lo más posible.

#### 3.2.2.3) Complejidad del código





Del análisis podemos ver que aunque el promedio de complejidad es 6.4, hay algunas clases con complejidad muy alta. Para que el codigo se considere aceptable deberíamos reducir la complejidad de las clases **por debajo de 10(chequear esto)**. A su vez, se debe asegurar que estas tengan un alto nivel de testeo y que sea mantenible.

VER QUE ES EL NUMERO 452. ES ESE EL QUE SE COMPARA CON LA TABLA PREVIA?

3.2.3) Tamaño del sistema



Como indica el plan de métricas especificado en el Plan de SQA, el tamaño del sistema de mide en base a la cantidad de líneas de código total. Utilizando la herramienta SonarQube, pudimos concluir que en el sistema hay 3413 lineas de código, distribuidas en 71 clases. A simple vista, parece un software mediano (Agregar tabla lineas de codigo tamano, por que es mediano?)y vemos que el promedio de líneas por clase es razonable. Sin embargo, haciendo un análisis más concreto, la mayor cantidad de código se encuentra en la clase GameMaster. Creemos que es un error darle tanta responsabilidad a una sola clase. Seguramente, este código pueda ser dividido en otras clases, y dividir así la responsabilidad.

3.3)Usabilidad

En cuanto a la usabilidad del sistema, lo encontramos de fácil uso e instintivo. Aunque se podría mejorar en varios aspectos.

4) Análisis de cambios

4.1) Cambios

Para una mejor organización de los cambios, podemos dividirlos en cuatro grandes secciones. En primer lugar, las nuevas funcionalidades que hay que implementar. Luego, lo que respecta al mantenimiento, mejorando la calidad del código y la aplicación de estándares de codificación. Por otro lado, mejorar la usabilidad del sistema y los cambios relacionados a las pruebas.

Para analizar los cambios que realizaremos, establecimos una tabla para cada área, que indica para cada uno de los cambios, el riesgo de realizarlo, el esfuerzo y el costo.

TABLA POR ÁREA,

4.1.1) Nuevas funcionalidades

|  |  |  |
| --- | --- | --- |
| Cambio | Riesgo | Esfuerzo y costo |
| Agregar carta Taxi |  |  |
| Permitir elegir color de ficha |  |  |
| Subir foto del usuario |  |  |
| Permitir almacenar un perfil de usuario |  |  |

4.1.2) Calidad de código

|  |  |  |
| --- | --- | --- |
| Cambio | Riesgo | Esfuerzo y costo |
| Mejora global del codigo segun estandares de codificacion |  |  |

4.1.3) Usabilidad

|  |  |  |
| --- | --- | --- |
| Cambio | Riesgo | Esfuerzo y costo |
| Agregar y corregir mensajes de error y de validación para facilitar el uso del usuario | Que el mensaje no sea el adecuado para el caso a considerar |  |
| Mejorar la usabilidad del software | Al intentar hacer que el sistema sea más fácil de usar, puede ocurrir que una funcionalidad deje de funcionar correctamente |  |

4.1.4) Pruebas

|  |  |  |
| --- | --- | --- |
| Cambio | Riesgo | Esfuerzo y costo |
| Mejorar cobertura de pruebas unitarias | - |  |
| Eliminar los asserts duplicados por prueba | Que dejen de funcionar las pruebas |  |

4.2) Impacto de cambios

Dado que el proyecto tiene errores a nivel general, en las distintas agrupaciones lógicas que se hacen, el impacto de los cambios es en todo el sistema.

Según las diferentes tablas, en las que evaluamos los cambios a realizar, se estimó que los cambios llevarán un esfuerzo de XX horas y tendrán un costo total de U$ . El costo total fue calculado a U$350 la hora de trabajo.