

Gestión de la Calidad de Software



TEMA

Gestión de la Calidad de Software

PROFESORES:

Ing.Martín M. Miceli <martinmiceli@gmail.com>

PROFESOR ADJUNTO:

Julián Nonino <julian.nonino@unc.edu.ar>

GRUPO:

BTM

INTEGRANTES:

Esteban Tissot <egtissot@gmail.com>

Matias Manero <matiasmanero@gmail.com>

CARRERA:

Ingeniería en Computación

AÑO:

2017

ÍNDICE

Enunciado	2
Introducción	4
Plan de Remoción de Defectos	5
Plan Standard vs Plan Integral	7
Plan Standard de SQA	8
Plan Integral de SQA	9
Revisión de los Requerimientos	10
Reporte de Revisión	12
Revisión del Diseño	13
Pruebas Unitarias	16
Pruebas de Sistema	21
Integración Continua	23
Conclusión	30
Referencias	31
Apendice	32
Jenkins	32
Java	32

Enunciado

Utilizando el trabajo desarrollado en la materia Ingeniería de Software, complete las siguientes tareas.

1. Plan de Remoción de Defectos
 - 1.1. Utilice una herramienta como [LocMetrics](#) para contar la cantidad de líneas de código de su proyecto.
 - 1.2. Elabore un plan de remoción de defectos, estimando la cantidad de defectos que deberían encontrarse en cada etapa.
 - 1.3. Compare la cantidad de defectos estimada con la cantidad de defectos que encontró al finalizar la materia Ingeniería de Software.
2. Revisión de los requerimientos.
 - 2.1. ¿Faltan requerimientos?
 - 2.2. ¿Sobran requerimientos?
 - 2.3. ¿Está correctamente escritos los requerimientos?
 - 2.4. Elabore los reportes de revisión necesarios.
 - 2.5. ¿En ésta etapa, se encontró la cantidad de defectos esperada de acuerdo a su plan de remoción de defectos?
3. Revisión del diseño.
 - 3.1. Revise todos los documentos de diseño elaborando los reportes de revisión.
 - 3.2. ¿En ésta etapa, se encontró la cantidad de defectos esperada de acuerdo a su plan de remoción de defectos?
4. Revisión de código.
 - 4.1. Elija una herramienta de revisión de código fuente y utilicela para revisar el código fuente de su proyecto. (por ejemplo, [Review Board](#)).
 - 4.2. ¿Cuántos defectos encontraron? ¿Fueron encontrados todos los defectos que esperaba encontrar?
5. Pruebas Unitarias
 - 5.1. ¿Qué cantidad de defectos espera encontrar en ésta etapa?
 - 5.2. Configure una herramienta que mida el porcentaje de código fuente cubierto por pruebas unitarias, por ejemplo [JaCoCo \(Java Code Coverage\)](#). ¿Cuál es su porcentaje de cobertura?
 - 5.3. Analice la diferencia entre los conceptos “*line coverage*”, “*instruction coverage*” y “*branch coverage*”. ¿Cuál considera más útil como métrica para un proyecto de software?
 - 5.4. ¿Qué valor de “*branch coverage*” tiene su proyecto? Considerando métricas como la complejidad ciclomática (McCabe) determine la cantidad de pruebas unitarias que debería realizar para cubrir los caminos independientes en su proyecto (*branch coverage*). ¿Cuántos casos de prueba adicionales necesitaría? Automatice los casos de pruebas necesarios para alcanzar el valor de cobertura deseado.
 - 5.5. Durante el proceso de implementación de las pruebas unitarias necesarias para ampliar el valor de cobertura, contabilice la cantidad de defectos que va encontrando en su producto.
 - 5.6. ¿Qué porcentaje de “*branch coverage*” es deseable para un proyecto de software? Justifique.
6. Pruebas de Sistema.

- 6.1. Determinar que los casos de prueba de sistema planteados en su trabajo final de Ingeniería de Software son suficientes para cubrir toda la funcionalidad o si se necesitan agregar más escenarios. En este último caso, especifique todos los nuevos escenarios.
 - 6.2. ¿En ésta etapa, se encontró la cantidad de defectos esperada de acuerdo a su plan de remoción de defectos?
7. Considerando que el valor de 1 (hora) de trabajo tiene un costo de \$ 100, y que su producto ya fue entregado al finalizar la materia Ingeniería de Software, es decir, el producto ya fue entregado al cliente, ¿qué cantidad de dinero podría haber ahorrado si hubiera aplicado estas técnicas para encontrar y remover defectos durante el desarrollo del proyecto, en lugar de hacerlo después de entregado?
8. Integración Continua.
Instale un servidor de integración continua como [Jenkins](#), [TeamCity](#), [TravisCI](#), etc. y establezca las configuraciones necesarias para lanzar un nuevo “build” con cada commit. Cada “build” debe incluir las siguientes etapas:
 - a. Compilación del código fuente. Haga que el “build” falle cuando existan errores de compilación.
 - b. Correr [CheckStyle](#) sobre el código fuente. Haga que el “build” falle cuando la cantidad de errores supere la cantidad que había en el “build” anterior.
 - c. Correr [PMD](#) sobre el código fuente. Haga que el “build” falle cuando la cantidad de errores supere la cantidad que había en el “build” anterior.
 - d. Correr [PMD CPD](#) sobre el código fuente. Haga que el “build” falle cuando la cantidad de errores supere la cantidad que había en el “build” anterior.
 - e. Correr [FindBugs](#) sobre el código fuente. Haga que el “build” falle cuando la cantidad de errores supere la cantidad que había en el “build” anterior.
 - f. Correr las pruebas unitarias. Calculando la cobertura de código. Haga que el build falle cuando alguna prueba no pase o el nivel de cobertura de código baje.

Elaboren un informe a entregar en formato PDF mostrando todo el trabajo realizado y detallando todo lo que crea necesario.

Elaborar una presentación de 15/20 minutos de duración más 5 minutos de preguntas para presentar ante el curso el trabajo realizado.

Enviar por mail un archivo ZIP con el informe, la presentación y todos los elementos que considere necesarios para mostrar su trabajo.

Introducción

Actualmente es difícil considerar un mundo sin que el software se encuentre inmerso en él ya que es considerado uno de los pilares de organizaciones y de la sociedad en general, este brinda productos y servicios que dependen de un correcto funcionamiento, como un marcapaso.

A medida que pasa el tiempo el software cobra cada vez más una complejidad que está asociada a múltiples fuentes como las metodologías utilizadas, tecnologías de apoyo, capacidad y competencias de las personas, productividad de los equipos de trabajo, requerimientos cambiantes de los clientes, presupuesto disponible, entre otras.

Por lo que hoy en día predomina una búsqueda de software que contenga una alta calidad. Pero, ¿a qué nos referimos cuando hablamos calidad de software?. Según Pressman es el cumplimiento con la funcionalidad y con los requerimientos de desempeño explícitamente enunciados, con los estándares de desarrollo explícitamente documentados, y con las características implícitas que son esperadas del desarrollo profesional de software.

Podemos decir que la calidad del software es una actividad que se aplica a lo largo de todo el proceso de ingeniería de software y utiliza herramientas como las revisiones, y las pruebas a lo largo del ciclo de desarrollo para asegurar que cada producto cumple con los requisitos que le han sido asignados.

En este caso en particular se implementa técnicas para mejorar la calidad de un software llamado Arkanoid, desarrollado anteriormente por los integrantes del grupo. Se comenzó por desarrollar un plan de remoción de defectos para luego hacer una minuciosa revisión de los requerimientos, que fueron planteados en el momento del desarrollo, seguidamente se desarrolló los test unitarios para asegurar una buena calidad.

Por último se logró una integración continua a través de un software denominado Jenkins.

Plan de Remoción de Defectos

Es de suma importancia desarrollar un plan que nos permita detectar y remover defectos tempranamente debido que el costo de reparación crece con el tiempo y el momento de desarrollo en que se encuentre el software, esto quiere decir que un defecto de requerimientos encontrado en la etapa de test es más costoso que si ese defecto hubiera sido encontrado en la etapa donde se introdujo o inmediatamente después.

En nuestro caso se trabajó sobre un software llamado Arkanoid desarrollado en la materia Ingeniería de Software. Se comienza por analizar el código con una herramienta denominada "LocMetrics" para tener conocimiento de la cantidad de líneas de código que posee el proyecto y empezar un plan de remoción de defectos para el mismo.

La utilización de la herramienta arroja los siguientes resultados:

Progress			
Source Files	38	C&SLOC, Code & Comment	1
Directories	22	CLOC, Comment Lines	19
LOC, Lines of Code	1885	CWORD, Comment Words	45
BLOC, Blank Lines	409	HCLOC, Header Comments	0
SLOC-P, Executable Physical	1457	HCWORD, Header Words	0
SLOC-L, Executable Logical	1159		
McCabe VG Complexity	66		

De aquí se puede saber que el proyecto cuenta con 1885 líneas de código(sin comentar). Según varios estudios como el de la Universidad Carnegie Mellon un el software comercial tiene de promedio entre 20 y 30 errores por cada 1.000 líneas de código.

Basado en esto el análisis se hará la suposición de que cada 1000 líneas de código hay entre 20 y 40 errores(Agregando 10 errores mas por la inexperiencia de los programadores), por lo que se estima que el proyecto contiene entre 38 y 79 errores.

Con base a este resultado se opta por tomar el peor de los casos(79 errores) para realizar el análisis correspondiente.

Partiendo de los 79 errores finales, se suponen con un porcentaje asignado cuantos errores introduce cada etapa del desarrollo de un software. A continuación se mostrará en la tabla los errores introducidos en cada etapa.

Al final la materia Ingeniería de Software los desarrolladores encontraron solo dos defectos en el proyecto contrastado con los 79 errores que se espera encontrar en un software de este tamaño.

Fase de Desarrollo	% Promedio de defectos originados	Cantidad de Defectos
Especificación de Requerimientos	15%	12
Diseño	35%	28
Codificación Unitaria	30%	24
Codificación de la Integración	10%	8
Documentación	10%	8
TOTAL	100%	80

Una vez conocidos los errores introducidos en cada fase se procede a calcular el costo de eliminarlos. El costo de cada error depende de en qué etapa fueron encontrados dichos errores, es decir mientras más temprano se encuentre el error menor costo tiene la solución del mismo.

Para el costo de remoción de los errores se tienen en cuenta dos parámetros, el sueldo de los programadores, el tiempo para eliminar el error y las horas mensuales que trabaja.

Para el primer parámetro nos basamos en una noticia de principios del año 2017, donde en promedio un programador sin experiencia (Junior) gana \$17000 mensuales, un programador con poca experiencia (Semi Junior) gana \$25000 mensuales, por lo que vamos a asumir que el sueldo de cada integrante del grupo es de \$22000 y que corregir un defecto en la etapa de requerimiento lleva una hora en promedio, por lo tanto para un ingeniero trabajando 160 hs. mensuales, podemos calcular el costo por unidad de detección de defecto con la siguiente ecuación:

$$\text{Costo por unidad de detección de defecto} = (\text{Sueldo} / \text{Horas Mensuales}) * \text{Costo De Un Defecto}$$

$$\text{Costo por unidad de detección de defecto} = \$138.$$

En este punto con los datos de la cantidad de líneas de código, la estimación de errores y el costo por detectar errores, se está en condiciones de formular un **Plan de SQA**.

Plan Standard vs Plan Integral

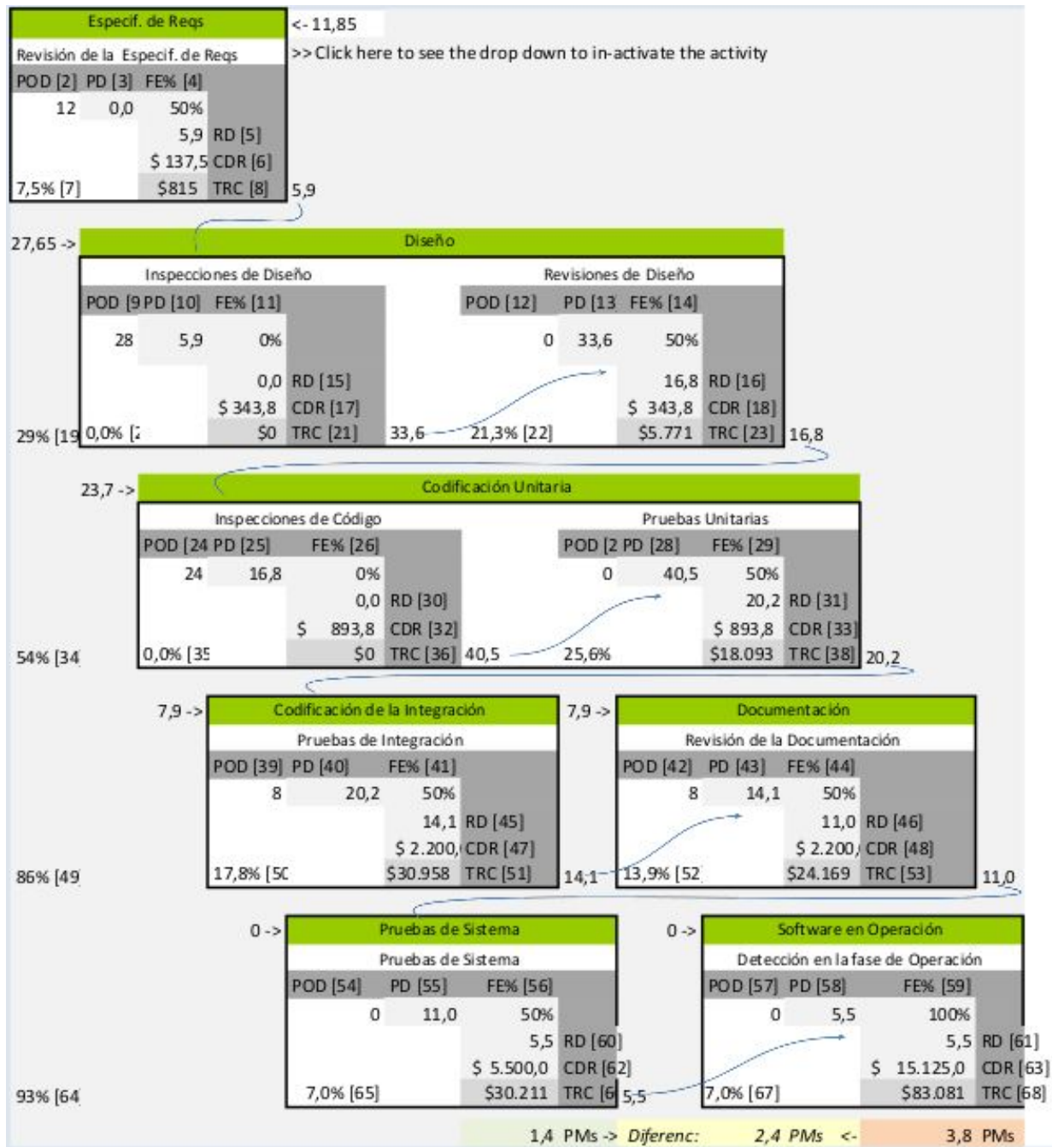
Se procede a desarrollar un plan de remoción de defectos teniendo en cuenta los grados de efectividad del cuadro de a continuación.

Actividades de Aseguramiento de la Calidad	Efectividad de remoción de defectos para un plan standard de SQA	Efectividad de remoción de defectos para un plan de SQA Integral
Revisión de la Especificación de Requerimientos	50%	60%
<i>Inspecciones de Diseño</i>	-----	70%
Revisiones de Diseño	50%	60%
<i>Inspecciones de Código</i>	-----	70%
Pruebas Unitarias	50%	40%
Pruebas de Integración	50%	60%
Revisión de la Documentación	50%	60%
Pruebas de Sistema	50%	60%
Detección en la fase de Operación	100%	100%

Como podemos observar en el cuadro se comparan dos planes de remoción de defectos, uno standard y el otro integral. Se puede apreciar que el plan integral es mucho más efectivo a la hora de remover defectos y a su vez agrega dos actividades que el plan standard no las contempla que son las de Inspecciones de Diseño y las Inspecciones de Código.

Con ayuda de una plantilla en excel podemos observar los defectos introducidos y removidos en cada etapa del desarrollo. A continuación se mostraran los resultados obtenidos utilizando los dos planes.

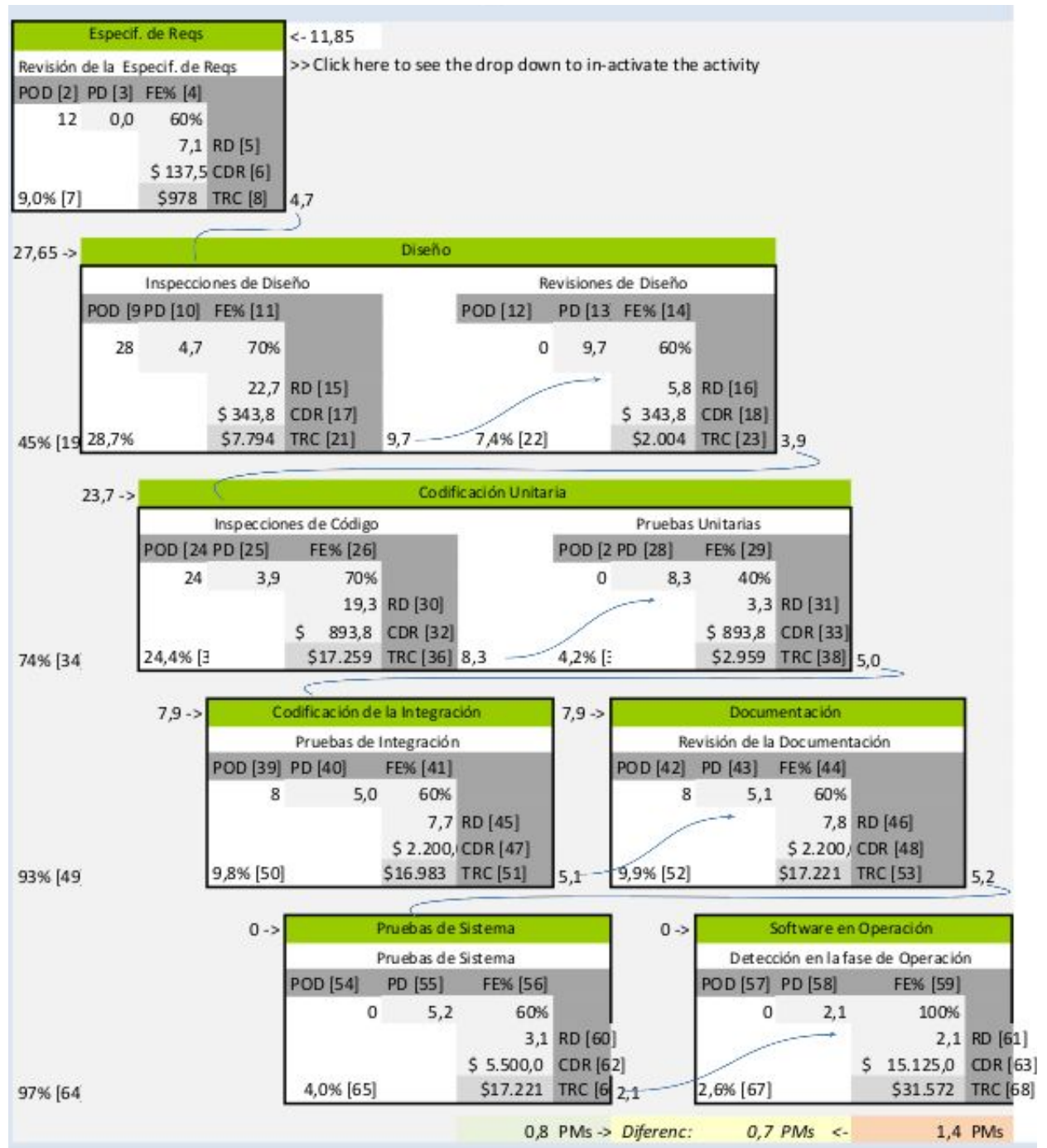
Plan Standard de SQA



Fuente: Adaptado del modelo de Daniel Galin, 2003

Autor Template: Martín Miceli v2.0.1, 2013

Plan Integral de SQA



Fuente: Adaptado del modelo de Daniel Galin, 2003

Autor Template: Martín Miceli v2.0.1. 2013

Como se observa en ambas gráficas, el costo de remover un defecto cuando se están ejecutando pruebas de sistemas es de \$5500 mientras que el precio asciende a \$15125 cuando se trata de remover un defecto teniendo el software en operación.

Si comparamos los dos planes podemos decir que para el caso de un plan standard de SQA con un 50% de eficacia en el filtrado de defectos para la etapa de pruebas de sistemas y un 100% de eficacia en la etapa de software en operación, se estima que será necesario un esfuerzo de 3.8 PMs (persona mes) para detectar y fixear los defectos

mientras que para el caso de un plan integral de SQA bajo los mismos costos pero teniendo un 60% de eficacia en el filtrado de defectos para la etapa de pruebas de sistemas y un 100% de eficacia en la etapa de software en operación se determina que es necesario un esfuerzo de 1.4 PMs para fixear el software.

Revisión de los Requerimientos

El planteamiento de requerimientos es la fase más importante en cada inicio de un proyecto debido que todo lo que se obtenga en esta fase será la base para la construcción del sistema. En esta etapa los analistas de requerimientos deberán trabajar junto al cliente para descubrir el problema que el sistema debe resolver. .

Luego de una definición de los requerimientos se debe refinar, analizar, y examinar/escudriñar los requerimientos obtenidos para asegurar que todos los clientes involucrados entienden lo que pidieron, y para encontrar errores, omisiones y otras deficiencias.

En esta etapa se leen los requerimientos, se conceptúan, se investigan, se intercambian ideas con el resto del equipo, se resaltan los problemas, se buscan alternativas y soluciones, y luego se van fijando reuniones con el cliente para discutir los requerimientos.

Las actividades a contemplar durante esta etapa son:

- **Reducir ambigüedades en los requerimientos.** En esta actividad se realizan las tareas que permiten eliminar los términos que tienen más de una acepción.
- **Traducir a lenguaje técnico los requerimientos.** Los requerimientos, ya con menos ambigüedades, deben ser tratados a los efectos de llevarlos a un lenguaje que se vaya aproximando al lenguaje técnico.
- **Plantear un modelo lógico.** Se debe construir un modelo del problema ya sea en términos de diagramas de flujo o cualquier otro tipo de representación que se considere conveniente para el modelado y que permite además, establecer un vínculo con la Etapa de Especificación.

Siguiendo estas pautas se analizó los requerimientos del sistema Arkanoid descriptos a continuación.

Requerimientos del proyecto Arkanoid
<p>Requerimientos Funcionales</p> <ul style="list-style-type: none">- El juego debe mostrar la puntuación durante la partida.- El juego debe mostrar el SCORE final al terminar la partida.- El juego debe tener dos bolas.- El juego debe terminar cuando una de las bolas se pierda.- Se debe manejar la barra con el teclado.- Deberá tener efectos de sonido acordes a los eventos en el juego.- La ventana debe ser de 400 x 580 píxeles.- Se debe poder elegir el fondo del juego.- El juego se debe poder pausar con una tecla. <p>Requerimientos NO Funcionales</p> <ul style="list-style-type: none">- El juego debe ser portable.- El software será para todo tipo de usuarios y edades, con una interfaz amigable que no tardará más de 5 minutos en amigarse con el mismo.- Se debe poder empezar a jugar a los 5 segundos o menos de haber inicializado el programa.- El desplazamiento de la barra debe ser fluido.- El software es realizado con fines de entretenimiento.- El juego debe finalizar al cerrarse la ventana del mismo.

Podemos concluir que los requerimientos no fueron bien redactados, usan un lenguaje general y pueden ser interpretados de diferentes formas por ejemplo en el requerimiento **“se debe poder elegir el fondo del juego”** ¿que estilo de fondo? , ¿un color, una imagen, un fondo animado? , ¿de donde se cambia? , ¿cuántos fondos distintos debe haber?. Por otro lado son pocos, por lo que es más difícil entender, desglosar y planificar bien el proyecto derivando en una mayor probabilidad de error. Por ejemplo faltan requerimientos de:

- Eficiencia. Ej: el software debe poder correr en un celular con un procesador Quad-core 1.2GHz Cortex-A7, 1gb de ram y 8gb de almacenamiento.
- Flexibilidad. Ej: El software debe poder permitir la extensión futura para poder agregar nuevos niveles de dificultad.
- Portabilidad. Ej: El software debe poder correr tanto en Android como en Linux.

Según el plan de remoción de defectos integral que se realizó en el apartado anterior en esta etapa hay 12 errores y se deben encontrar 7, pero debido a la mala redacción de los requerimientos se concluye que hay más de 12 errores y que deberían reescribirse.

Reporte de Revisión

Por último se elabora un reporte de Revisión detallando los temas discutidos y las acciones llevadas a cabo para la remoción de los defectos de esta Etapa. En la parte baja del documento se encuentra marcada la opción que se deben rehacer todos los requerimientos para pasar esta etapa.

Reporte de Revisión de Requerimiento			
Fecha:		23 de Octubre de 2017	Preparado por: Esteban
Nombre del Proyecto:		Arkanoid	
Documento revizado:		SI	Versión: 1
1. Resumen de las discusiones			
#	Temas de la discusión		Nro. acciones
1	Definición errónea de requerimientos		
2	Requerimientos muy poco específicos		
3	Falta de Requerimientos		
4	El software es realizado con fines de entretenimiento - No es requerimien		
5			
6			
2. Listado de Acciones			
#	Acciones	Responsable	Fecha de Fin
1	Redactar nuevamente los Requerimientos		
2			
3			
4			
5			
6			
7			
8			
3. Decisión sobre el documento revizado (ponga una X en donde corresponda)			
<input type="checkbox"/> Aprobado			
<input type="checkbox"/> Parcialmente Aprobado.			
Las partes aprobadas para proseguir a la siguiente fase son las siguientes:			
<div></div>			
<input checked="" type="checkbox"/> Desaprobado			
4. Participantes			
#	Nombre y Apellido	Rol	Firma
	Tissot Esteban		
	Manero Matias		

Revisión del Diseño

Así como se realizó la revisión de requerimientos, se emplea la misma técnica para el diseño. En nuestro caso se concluye que se ha logrado un diseño completo con diagramas de mucha utilidad a la hora de pasar a la etapa de programación, pero que el uso de un diagrama de actividad o de secuencia hubiera sido de una gran utilidad para lograr una mayor comprensión del funcionamiento del software. Por lo que para que esta etapa sea aprobada y se pueda pasar a la siguiente se recomienda que se desarrollen dichos diagramas.

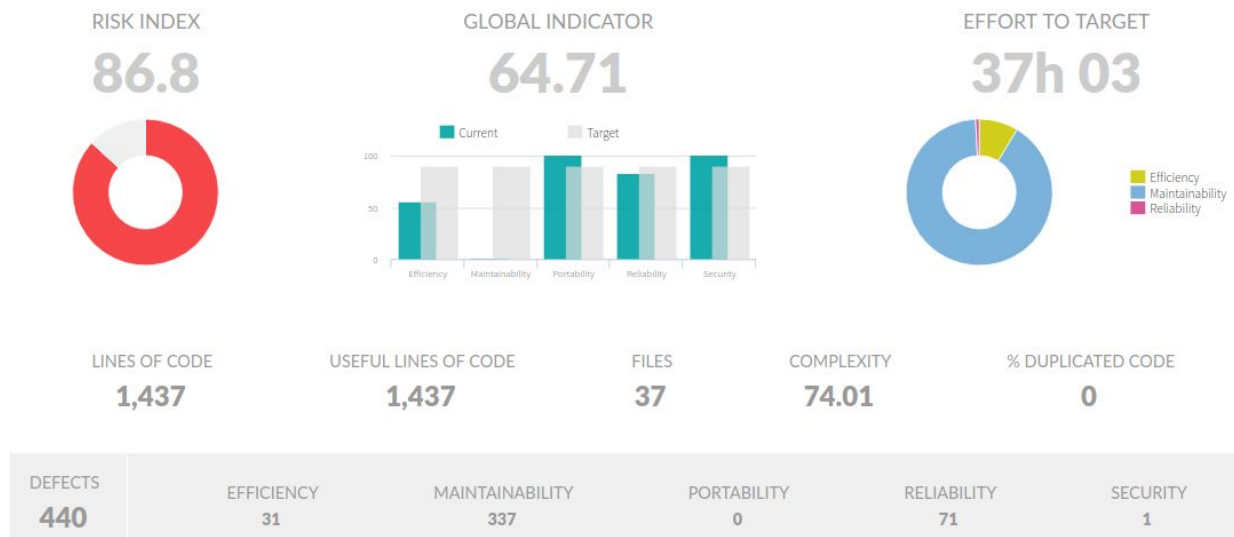
Reporte de Revisión de Diseño			
Fecha:	23 de Octubre de 2017	Preparado por: Matias	
Nombre del Proyecto:	Arkanoid		
Documento revizado:	SI	Versión: 1	
1. Resumen de las discusiones			
#	Temas de la discusión	Nro. acciones	
1	Faltan Diagramas		
2			
3			
4			
5			
6			
2. Listado de Acciones			
#	Acciones	Responsable	Fecha de Fin
1	Hacer diagrama de Secuencia		
2			
3			
4			
5			
6			
7			
8			
3. Decisión sobre el documento revizado (ponga una X en donde corresponda)			
<input type="checkbox"/> Aprobado			
<input checked="" type="checkbox"/> Parcialmente Aprobado.			
Las partes aprobadas para proseguir a la siguiente fase son las siguientes:			
Todo lo que esta realizado			
<input type="checkbox"/> Desaprobado			
4. Participantes			
#	Nombre y Apellido	Rol	Firma
	Tissot Esteban		
	Manero Matias		

Según el plan de remoción de defectos integral en esta etapa, hay 10 errores y se deben encontrar y remover 6 de ellos. La revisión del documento se aprobó de forma condicional, es decir que el equipo de desarrollo debe hacer un diagrama de secuencia antes de comenzar con la siguiente etapa.

Revisión de Código

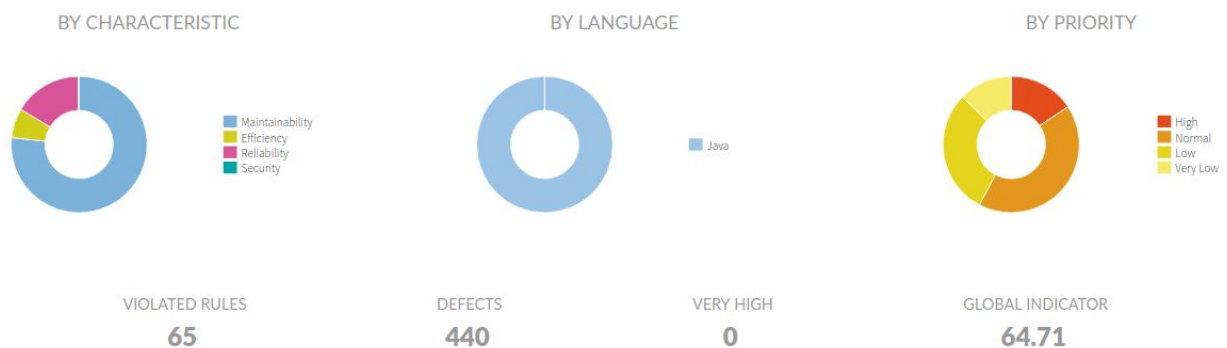
Para la revisión de código fuente se utilizó la herramienta [Kiuwan](#) que nos permite loguearnos con nuestra cuenta de github y analizar el proyecto de forma online. a continuación se muestra un resumen de los resultados que se obtuvieron al ejecutar la herramienta.

QUALITY



El indicador de riesgo es un valor propio del software que integra el resto de los indicadores, su rango es de 0 a 100 y tiene sentido si se observa a lo largo del tiempo para medir la evolución del proyecto hablando en términos de calidad.

DEFECTS



Se puede observar que se encontraron 440 defectos de los cuales 337 corresponden a la categoría mantenimiento. En la imagen posterior se muestra un poco más en detalle las reglas violadas, la clasificación a la que corresponden, la prioridad y el esfuerzo necesario para corregir el defecto. Se observa que la regla violada de mayor prioridad es debido a que el código no está comentado y que la mayor cantidad de defectos (91) es porque se utilizaron números literales en vez de variables como parámetros (hardcodear).

Files	Defects	Rule		Priority	Characteristic	Language	Effort
	440						251h
37	37	Provide Javadoc comments for public classes and interfaces Agile Alliance:Clear-CLDO Agile Alliance:Clear-CMTD Documentation	?	High	Maintainability	Java	18h 30
5	8	Avoid using method calls in a loop	?	High	Efficiency	Java	4h 00
4	4	Avoid catch blocks with empty bodies	?	High	Reliability	Java	2h 00
4	4	Avoid using try statements in loops	?	High	Efficiency	Java	2h 00
2	4	Do not use the printStackTrace method CERT-J-ERR01-J	?	High	Efficiency	Java	24m
3	3	Avoid using while() and sleep(), instead use wait() and notify() CERT-J-LCK09-J	?	High	Efficiency	Java	18m
2	2	Provide Javadoc comments for public fields Agile Alliance:Clear-CMTD	?	High	Maintainability	Java	12m
1	2	Use equals() when comparing Strings CERT-J-EXP03-J CERT-J-EXP50-J	?	High	Reliability	Java	12m
1	1	Avoid parameter method names that provoke conflicts with class members names	?	High	Maintainability	Java	03m
1	1	Private constructors in class with static members	?	High	Maintainability	Java	8h 00
1	1	Avoid using repeated cast over the same object or variable (Max 3 cast of every type) micro-optimization	?	High	Efficiency	Java	30m
1	1	Initialize all static fields	?	High	Reliability	Java	30m
1	1	Avoid calling at java.lang.Thread.run() CERT-J-THI00-J	?	High	Reliability	Java	06m
15	91	Avoid using numeric literals	?	Medium	Maintainability	Java	9h 06
8	16	Always check object type before cast essential potential-bug	?	Medium	Reliability	Java	8h 00

Según el plan de remoción de defectos integral en esta etapa hay 24 errores propios más 4 de la etapa anterior y se deben encontrar y remover 20 errores.

Si sacamos las reglas de mantenimiento en el software Kiuwan obtenemos que la cantidad de defectos son 103 y 28 de ellos son de alta prioridad. Gracias a la herramienta podemos identificar los defectos e implementar las acciones para removerlos rápidamente, por ejemplo, se debe evitar llamar a métodos dentro de bucles.

Files	Defects	Rule		Priority	Characteristic	Language	Effort
	103						102h
5	8	Avoid using method calls in a loop	?	High	Efficiency	Java	4h 00
4	4	Avoid catch blocks with empty bodies	?	High	Reliability	Java	2h 00
4	4	Avoid using try statements in loops	?	High	Efficiency	Java	2h 00
2	4	Do not use the printStackTrace method CERT-JERR01-J	?	High	Efficiency	Java	24m
3	3	Avoid using while() and sleep(), instead use wait() and notify() CERT-JLCK09-J	?	High	Efficiency	Java	18m
1	2	Use equals() when comparing Strings CERT-JEXP03-J CERT-JEXP50-J	?	High	Reliability	Java	12m
1	1	Avoid using repeated cast over the same object or variable (Max 3 cast of every type) micro-optimization	?	High	Efficiency	Java	30m
1	1	Initialize all static fields	?	High	Reliability	Java	30m
1	1	Avoid calling at java.lang.Thread.run() CERT-JTHI00-J	?	High	Reliability	Java	06m
8	16	Always check object type before cast essential potential-bug	?	Medium	Reliability	Java	8h 00
6	8	Avoid unused local variables CERT-JMSC56-J essential	?	Medium	Reliability	Java	48m
2	2	Declare classes where all constructors are private and 'final'	?	Medium	Reliability	Java	16h 00










Pruebas Unitarias







En la etapa de pruebas Unitarias podemos decir que el software arkanoid tiene una carpeta dedicada al desarrollo de las misma. Dentro de la carpeta podemos encontrar 6 archivos con extensión .java, en donde cada uno de ellos representa un Unitest.

En nuestro plan de remoción de defectos integral calculamos que deberíamos encontrar cerca de 9 errores y remover 4 errores de ellos.

Procedimos a correr una software denominado JaCoCo que nos proporciona la creación y recolección de reportes de la ejecución de pruebas unitarias y de integración. Los reportes son utilizados para mostrar el porcentaje de cubrimiento de pruebas del proyecto y a su vez indicar cuales clases no han sido cubiertas por las pruebas unitarias.

En las imagen a continuación veremos el porcentaje de cobertura del sistema total como tambien asi otras métricas que proporciona el software.

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
▼ arkanoid_MVC	 32,0 %	1.099	2.331	3.430
▼ src	 32,0 %	1.099	2.331	3.430
▶ Menu	 0,0 %	0	47	47
▶ arkanoid_beat	 0,0 %	0	160	160
▶ Strategy	 0,0 %	0	315	315
▶ UnitTest	 96,6 %	311	11	322
▶ Heart	 35,5 %	123	223	346
▶ Beat	 0,0 %	0	921	921
▶ arkanoid_MVC	 50,4 %	665	654	1.319

Counter	Coverage	Covered	Missed	Total
Instructions	 31,7 %	1.086	2.344	3.430
Branches	 36,0 %	49	87	136
Lines	 29,4 %	261	626	887
Methods	 27,4 %	58	154	212
Types	 31,1 %	14	31	45
Complexity	 24,6 %	69	211	280

Como se puede observar en la imagen de arriba el reporte generado contiene 6 métricas que ayudan a tener un conocimiento sobre la cobertura del código alcanzada por los unitest.

Para nuestro análisis nos he de importancia las siguientes métricas:

1. **Line Coverage:** Una sola línea de un código puede referirse a múltiples instrucciones, métodos o múltiples clases. Una línea se considera ejecutada cuando se ha ejecutado al menos una instrucción asignada a esta línea. Existen tres estados diferentes para cada línea que contiene el código:
 - Sin cobertura: no se ha ejecutado ninguna instrucción en la línea (fondo rojo)

- Cobertura parcial: solo se ha ejecutado una parte de la instrucción en la línea (fondo amarillo)
- Cobertura completa: todas las instrucciones en la línea se han ejecutado (fondo verde)







2. **Instruction Coverage:** Proporciona información sobre la cantidad de código que se ha ejecutado o perdido.
3. **Branch coverage:** Esta métrica tiene en cuenta el número total de ramas en un método y determina cuál de ellas fueron ejecutadas o pérdidas.
4. **Complexity:** La idea de esta métrica no es contar los bucles (for, while, do...) en el código de un programa sino en el resultado de contar el número de ciclos diferentes que se siguen en un fragmento de código de un programa habiendo creado una rama imaginaria desde el nodo de salida al nodo de entrada del diagrama de flujo correspondiente a este fragmento de código. El análisis matemático ha demostrado que la complejidad ciclomática indica el número máximo de casos de prueba necesarios para probar cada punto de decisión en un programa.

Por lo que podemos observar en las imágenes anteriores, el total del código cubierto es del 32%, este es un valor muy bajo por lo que se tendrían que desarrollar más unittest para lograr un nivel de cobertura aceptable. En nuestra opinión un nivel arriba del 80% ya se encuentra en una zona aceptable. Viendo el valor de la métrica de complejidad podemos concluir que se necesitan al menos 211 unittest para cubrir los caminos independientes faltantes.






Podemos hacer un análisis más minucioso y observar las distintas métricas para las diferentes clases que posee el proyecto, para poder encarar un plan de remoción correctamente.

En las imágenes a continuación veremos las métricas para las distintas clases, cabe aclarar que muchas de las clases no tiene cobertura por los unittest por lo que solo se agrego la imagen de una sola de esas clases.







Arkanoid_MVC

Properties for arkanoid_MVC					
Coverage					
Session: AllTests (28/10/2017 21:13:43)					
Counter		Coverage	Covered	Missed	Total
Instructions		49,8 %	657	662	1.319
Branches		53,0 %	35	31	66
Lines		46,4 %	148	171	319
Methods		52,6 %	40	36	76
Types		43,8 %	7	9	16
Complexity		45,9 %	50	59	109

Menu

Properties for Menu						×
Coverage						↩ ▼ ↪ ▼ ▼
Session: AllTests (28/10/2017 21:13:43)						
Counter		Coverage	Covered	Missed	Total	
Instructions		0,0 %	0	47	47	
Lines		0,0 %	0	12	12	
Methods		0,0 %	0	3	3	
Types		0,0 %	0	1	1	
Complexity		0,0 %	0	3	3	

UnitTest

Properties for UnitTest						×
Coverage						↩ ▼ ↪ ▼ ▼
Session: AllTests (28/10/2017 21:13:43)						
Counter		Coverage	Covered	Missed	Total	
Instructions		96,6 %	311	11	322	
Branches		50,0 %	5	5	10	
Lines		97,6 %	82	2	84	
Methods		92,3 %	12	1	13	
Types		85,7 %	6	1	7	
Complexity		66,7 %	12	6	18	

Heart

Properties for Heart					
Coverage					
Session: AllTests (28/10/2017 21:13:43)					
Counter		Coverage	Covered	Missed	Total
Instructions		34,1 %	118	228	346
Branches		37,5 %	9	15	24
Lines		26,7 %	31	85	116
Methods		15,8 %	6	32	38
Types		20,0 %	1	4	5
Complexity		14,0 %	7	43	50

Luego de hacer el análisis de las distintas métricas se pasó a desarrollar Unitest para lograr una mayor cobertura de los distintos caminos independientes que tiene el código.

En nuestro caso particular se buscó lograr una métrica de complejidad de cero para los códigos desarrollados por los estudiantes y no los códigos provisto por el profesor en la materia ingeniería de software.

Las clases que se busco optimizar se encuentra en la carpeta de Arkanoid_MVC y arkanoid_beat.







Se desarrollaron 3 clases nuevas, que se encuentran guardadas en la carpeta UniTest y las misma contiene un total de 19 Unitest logrando una cobertura total del código de 85,4% como se ve en la imagen a continuación.

Element	Coverage	Covered Instructions	Missed Instructions
▼ arkanoid_MVC	85,4 %	3.673	628
▼ src	85,4 %	3.673	628
▶ Menu	89,4 %	42	5
▶ arkanoid_beat	89,4 %	143	17
▶ Strategy	60,6 %	191	124
▶ Heart	63,3 %	219	127
▶ Beat	73,9 %	678	240
▶ UnitTest	98,6 %	1.150	16
▶ arkanoid_MVC	92,7 %	1.250	99







Y las carpetas de Arkanoid_MVC y arkanoid_beat que contiene las clases que se buscaron cubrir con los Unitest logramos pasar de una cobertura de 50,4% y 0% a tener 92,7% y 89,4% respectivamente.

A continuación se muestran como quedaron las métricas con los nuevos Unitest.

Sistema Total

Properties for arkanoid_MVC				
Coverage				
Session: AllTests (05/11/2017 23:52:59)				
Counter	Coverage	Covered	Missed	Total
Instructions	 85,4 %	3.673	628	4.301
Branches	 63,2 %	86	50	136
Lines	 81,2 %	879	204	1.083
Methods	 71,4 %	180	72	252
Types	 96,2 %	50	2	52
Complexity	 67,2 %	215	105	320

Arkanoid_MVC

Coverage				
Session: AllTests (05/11/2017 23:52:59)				
Counter	Coverage	Covered	Missed	Total
Instructions	 92,7 %	1.250	99	1.349
Branches	 83,3 %	55	11	66
Lines	 91,8 %	293	26	319
Methods	 88,6 %	70	9	79
Types	 93,8 %	15	1	16
Complexity	 85,7 %	96	16	112

Como para concluir esta etapa se puede decir que nos encontramos conforme con los valores de las métricas obtenidos. Podemos decir que se encuentran en un valor que consideramos aceptables, lograr valores de cobertura de 100% en métricas como el nivel de cobertura de branch requiere mucho tiempo de los programadores de pruebas y por lo tanto dinero que se precisa cuando en realidad con un porcentaje de cobertura mayor al 80% consideramos que ya suficiente.

Pruebas de Sistema

Las pruebas de sistema tienen un propósito particular: para comparar el sistema o el programa con su objetivos originales como los requerimientos. Es decir son un proceso que trata de demostrar cómo el programa en su totalidad resuelve sus objetivos o requerimientos.

Las pruebas son una fase de investigación en la que se asegura que cada componente unitario o módulo interactúa con otro tal como fue diseñado. Verifica el correcto funcionamientos de las interfaces entre los distintos subsistemas que lo componen y con el resto de sistema con los que se comunica.

En el caso del software Arkanoid no existe desarrollada ni planificada una prueba de sistema, por lo que se debería planificar una prueba coherente con los requerimientos que logre validar que los mismo hayan sido cumplidos por los desarrolladores del proyecto. Los posibles escenarios son:

ID	01
Nombre	Sistema de Puntuación
Descripción	Cada vez que se evita que una bola se caiga se debe sumar un punto y el juego debe mostrar el acumulado.
Pasos y Condiciones de Ejecución	1.Inicialización del Juego. 2.Evitar que la pelota se caiga.
Resultado Esperado	Se debe mostrar la puntuación acumulada en tiempo de ejecución centrada en el borde superior con letras de color negro y con el título de SCORE.

ID	02
Nombre	Puntuación Final
Descripción	Cuando se pierde se debe mostrar un mensaje de game over con la puntuación acumulada.
Pasos y Condiciones de Ejecución	1.Inicialización del Juego. 2.Evitar que la pelota se caiga por un tiempo. 3.Dejar que la bola caiga.
Resultado Esperado	Se debe mostrar la puntuación acumulada en una ventana nueva centrada en referencia a la ventana principal con letras de color negro y con el título de GAME OVER.

ID	03
Nombre	Recursos del Juego
Descripción	El juego debe inicializar con dos bolas.
Pasos y Condiciones de Ejecución	1.Inicialización del Juego.
Resultado Esperado	El juego debe inicializar con dos bolas en posiciones diferentes con movimientos independientes y con diámetro igual a 20 pixeles.

ID	04
Nombre	Game Over
Descripción	Cuando se pierde el juego debe finalizar.
Pasos y Condiciones de Ejecución	1.Inicialización del Juego. 2.Evitar que la pelota se caiga por un tiempo. 3.Dejar que la bola caiga.
Resultado Esperado	La puntuación debe dejar de incrementar y las bolas y la raqueta deben dejar de moverse

ID	05
Nombre	Raqueta
Descripción	La raqueta debe ser manipulada por el teclado del usuario.
Pasos y Condiciones de Ejecución	1.Inicialización del Juego. 2.Mover la raqueta
Resultado Esperado	La raqueta debe responder a las teclas de entrada (derecha/izquierda) y el movimiento solo puede ser horizontal.

ID	06
Nombre	Sonido
Descripción	El sonido debe ser acorde a los eventos del juego.
Pasos y Condiciones de Ejecución	1.Inicialización del Juego. 2.Evitar que la pelota se caiga por un tiempo. 3.Dejar que la bola caiga.
Resultado Esperado	Cada vez que la raqueta intersecta alguna de las bolas se debe reproducir un sonido. Cuando el juego finaliza el sonido debe ser otro.

ID	07
Nombre	Ventana Principal
Descripción	Ventana en la que se ejecuta el juego
Pasos y Condiciones de Ejecución	1.Inicialización del Juego.
Resultado Esperado	La ventana del juego debe ser de 400x580 píxeles ubicada en el borde superior izquierdo de la pantalla. Debe contener una imagen de fondo.

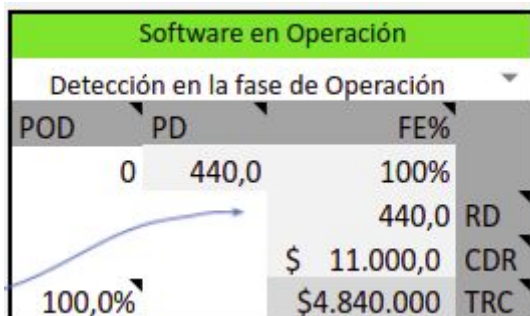
ID	08
Nombre	Fondo
Descripción	La imagen de fondo del juego se debe poder elegir
Pasos y Condiciones de Ejecución	1.Inicialización del Juego.
Resultado Esperado	El juego debe mostrar un menú al inicio que permita cambiar la imagen de fondo.

ID	09
Nombre	Pausa
Descripción	Presionando la tecla "P" se debe detener el juego.
Pasos y Condiciones de Ejecución	1.Inicialización del Juego. 2.Evitar que la pelota se caiga por un tiempo. 3.Presionar la tecla "P" 4.Volver a presionar la tecla "P"
Resultado Esperado	Al presionar la tecla se debe detener el movimiento de las bolas y de la raqueta y en la ventana solo se debe mostrar el puntaje y la palabra PAUSA en color negrita y centrada.

Según nuestro plan de remoción de defectos integral en esta etapa hay 6 errores y se deben encontrar y remover 4 errores. Sin embargo como no se realizaron estas pruebas no se puede encontrar los defectos esperados.

Costo

Si nos basamos en la cantidad de defectos que nos arrojó el software Kiuwan (440 defectos), en que el costo de 1 hora de trabajo tiene un valor de \$100 y en que el software ya fue entregado al cliente encontrar y remover los defectos nos costaría \$4.840.000.

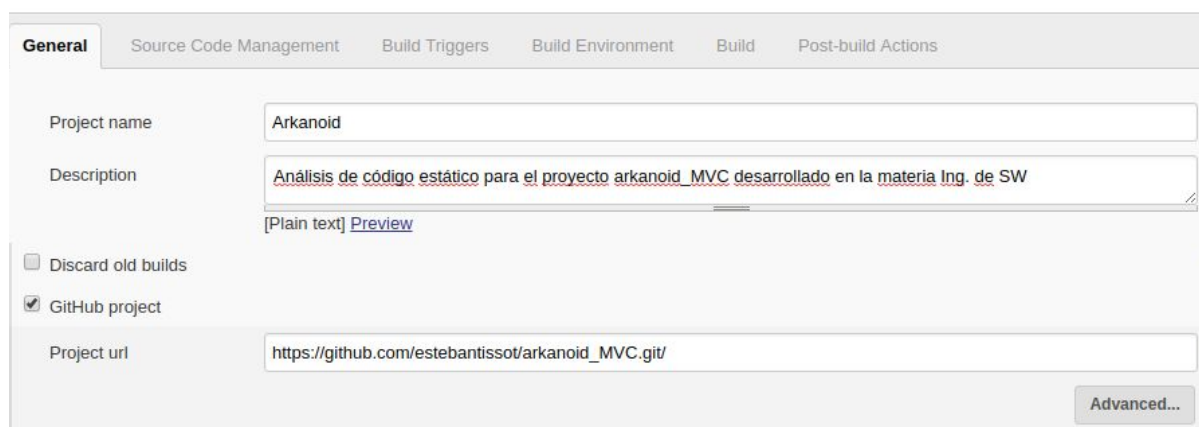


Por otro lado si se hubiera aplicado el plan integral, el costo total sería de sería de \$504.850. Por lo que se concluye que utilizando las prácticas de calidad de software se disminuye 5 veces el costo de detectar y solucionar defectos.

Etapa	Cantidad de Def. removidos	Costo Total
Especificación de Req	40	40000
Diseño	159	39750
Codificación Unitaria	126	81900
Codificación de la Int.	43	68800
Documentación	44	70400
Pruebas de Sistema	18	72000
Software en Operación	12	132000
Total	442	504.850

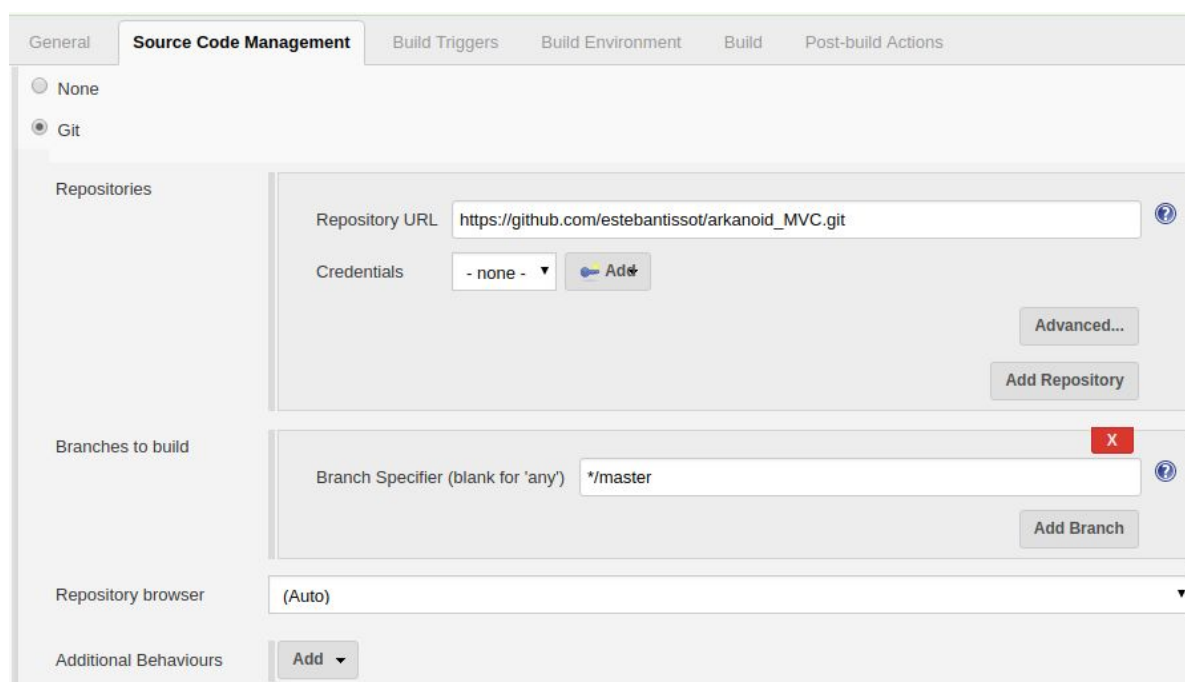
Integración Continua

Se instaló el servidor de integración continua Jenkins y se agregaron los plug-in necesarios para realizar un Análisis estático de código del proyecto “arkanoid_MVC” desarrollado en la materia Ingeniería de Software. A continuación se detallan las configuraciones necesarias:



The screenshot shows the 'General' tab of the Jenkins configuration interface. The 'Project name' is 'Arkanoid'. The 'Description' is 'Análisis de código estático para el proyecto arkanoid_MVC desarrollado en la materia Ing. de SW'. There are checkboxes for 'Discard old builds' (unchecked) and 'GitHub project' (checked). The 'Project url' is 'https://github.com/estebantissot/arkanoid_MVC.git/'. An 'Advanced...' button is at the bottom right.

El plug-in de GitHub permite que Jenkins descargue los archivos del repositorio donde se encuentra el proyecto, esta operación se realiza con cada “build” permitiendo así analizar siempre la última versión del código.



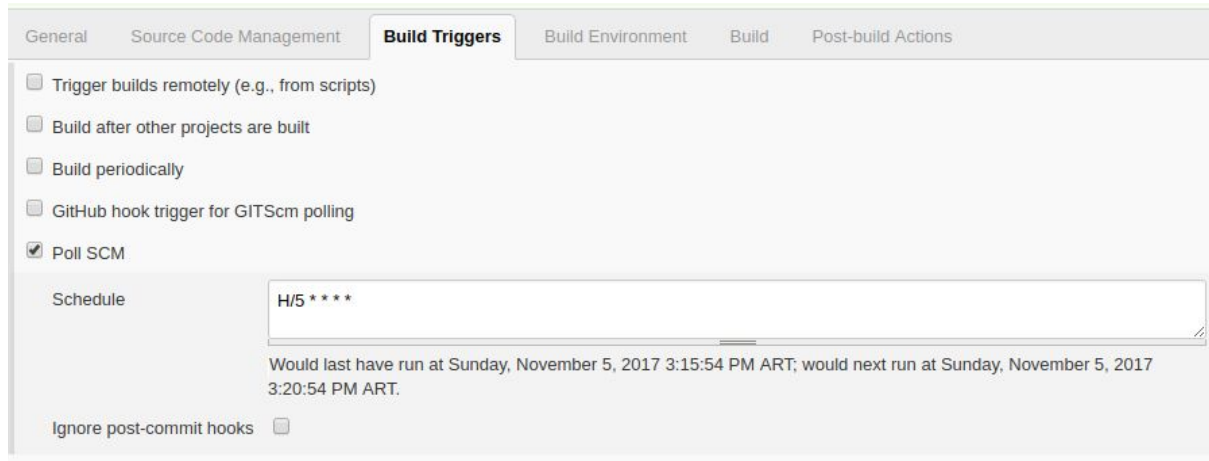
The screenshot shows the 'Source Code Management' tab. The 'Git' radio button is selected. Under 'Repositories', the 'Repository URL' is 'https://github.com/estebantissot/arkanoid_MVC.git' and 'Credentials' is set to '- none -'. There are 'Advanced...' and 'Add Repository' buttons. Under 'Branches to build', the 'Branch Specifier (blank for \'any\')' is '*/master', with an 'Add Branch' button. The 'Repository browser' is set to '(Auto)'. At the bottom, there is an 'Additional Behaviours' section with an 'Add' button.

Se pueden configurar “triggers” para lanzar los build, por ejemplo para que se lance un build cada vez que se hace un commit se debería configurar un hook en el repositorio de GitHub. Para ello desde el repositorio se pulsa sobre el enlace Settings -> Webhooks &

Services. Se mostrará la pantalla de configuración de los hooks, pulsamos sobre Add service y escribimos jenkins, seleccionamos el servicio Jenkins (GitHub plugin).

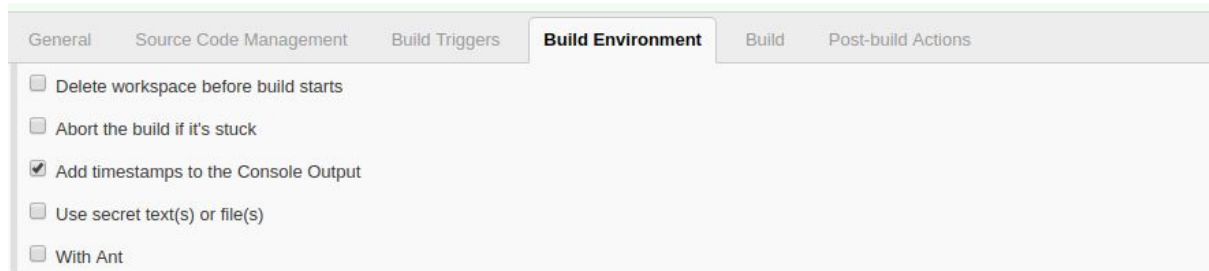
Se nos mostrará la pantalla de configuración del servicio Jenkins de GitHub, en esta pantalla únicamente tenemos que introducir la URL al hook de nuestro Jenkins que será de la forma `http://IP_JENKINS:8080/github-webhook/` y pulsamos el botón Add Service.

En este trabajo no se utilizó una ip pública para el servidor Jenkins, por lo que no se puede realizar la configuración anterior, como alternativa Jenkins puede revisar el repositorio, cada 5 minutos en este caso, si hay un nuevo commit y en base a eso lanzar o no un nuevo "build".



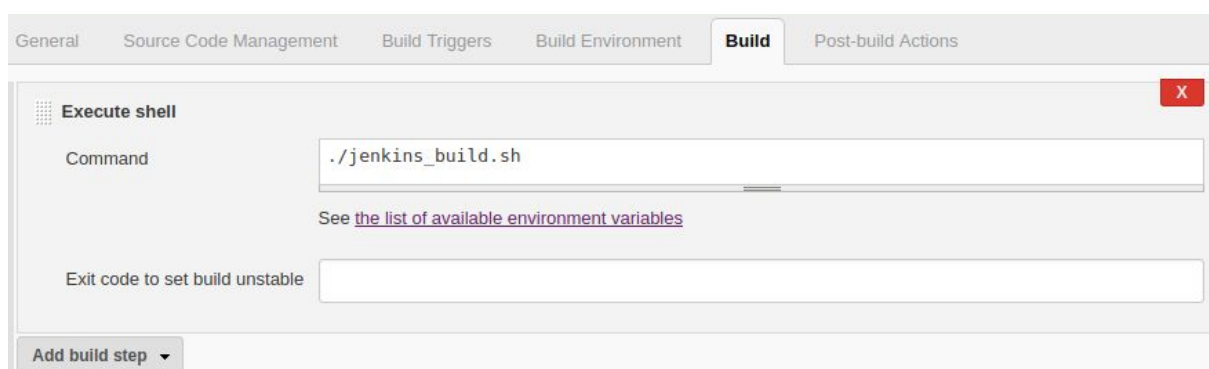
The screenshot shows the 'Build Triggers' tab in the Jenkins configuration interface. It includes several checkboxes: 'Trigger builds remotely (e.g., from scripts)', 'Build after other projects are built', 'Build periodically', 'GitHub hook trigger for GITScm polling', and 'Poll SCM' (which is checked). Below these is a 'Schedule' field containing the text 'H/5 * * * *'. A message below the schedule field states: 'Would last have run at Sunday, November 5, 2017 3:15:54 PM ART; would next run at Sunday, November 5, 2017 3:20:54 PM ART.' At the bottom, there is an 'Ignore post-commit hooks' checkbox which is unchecked.

Jenkins permite configurar el entorno donde se va a ejecutar el build.



The screenshot shows the 'Build Environment' tab in the Jenkins configuration interface. It includes several checkboxes: 'Delete workspace before build starts', 'Abort the build if it's stuck', 'Add timestamps to the Console Output' (which is checked), 'Use secret text(s) or file(s)', and 'With Ant'.

En este apartado se configura el build. Se puede integrar herramientas de gestión y construcción como Maven o Ant y también realizar un pipeline. En este caso se utilizó la consola de linux para correr un pequeño script.



The screenshot shows the 'Build' tab in the Jenkins configuration interface. It features a section titled 'Execute shell' with a red 'X' icon in the top right corner. Inside this section, there is a 'Command' field containing the text './jenkins_build.sh'. Below the command field is a link that says 'See the list of available environment variables'. At the bottom of the section is an 'Exit code to set build unstable' field. Below the 'Execute shell' section is a button labeled 'Add build step' with a dropdown arrow.

Por defecto Jenkins llama a la consola con los parámetros “-xe” (-x imprime todos y cada uno de los comandos y -e causa que el script deje de correr si alguno de los comandos falla, salida distinta de 0), para que se ejecuten todos los comandos se creó un script y se agregó “#!/bin/sh” en la primera línea del script.

```
jenkins_build.sh

#!/bin/sh
# Búsqueda y Compilación de todos los archivos .java del proyecto menos los de la carpeta UnitTest
cd ./src/
find ./ -iname *.java -not -path "./UnitTest/*" > files-java.txt
javac -d ../build @files-java.txt
cd ..
# Revision estatica deCodigo (CheckStyle - FindBugs -PMD_CPD - PMD )
java -jar ../checkstyle-8.4/checkstyle-8.4-all.jar -c /google_checks.xml
-f xml -o ./Documentacion/IntegracionContinua/checkstyle_report.xml ./src/
../findbugs-3.0.1/bin/findbugs -xml ./build/ >
./Documentacion/IntegracionContinua/findbugs_report.xml
../pmd-bin-6.0.0-SNAPSHOT/bin/run.sh cpd --minimum-tokens 100 --files
./src/ --format xml > ./Documentacion/IntegracionContinua/cpd_report.xml
../pmd-bin-6.0.0-SNAPSHOT/bin/run.sh pmd -d ./src/ -f xml -R
rulesets/java/unusedcode.xml > ./Documentacion/IntegracionContinua/pmd_report.xml
```

El script busca y compila los archivos del proyecto y luego ejecuta los software para el análisis de código (CheckStyle, findbugs, pmd y pmd-cpd), estos programas generan un reporte de resultados en un archivo xml. Estos archivos de reporte luego son interpretados y mostrados por los correspondientes plug-in que son configurados en la sección “Post-build Actions”.

The screenshot shows the Jenkins 'Post-build Actions' configuration page. It has tabs for General, Source Code Management, Build Triggers, Build Environment, Build, and Post-build Actions. The 'Post-build Actions' tab is active, showing two sections: 'Publish Checkstyle analysis results' and 'Publish FindBugs analysis results'. Each section has a text input field for the report file path, a 'Fileset includes' description, and an 'Advanced...' button. The 'Publish Checkstyle analysis results' section has a file path of 'Documentacion/IntegracionContinua/checkstyle_report.xml'. The 'Publish FindBugs analysis results' section has a file path of 'Documentacion/IntegracionContinua/findbugs_report.xml' and a checkbox for 'Use rank as priority' which is currently unchecked.

General Source Code Management Build Triggers Build Environment Build **Post-build Actions**

Publish Checkstyle analysis results

Checkstyle results Documentacion/IntegracionContinua/checkstyle_report.xml

[Fileset includes](#) setting that specifies the generated raw CheckStyle XML report files, such as `**/checkstyle-result.xml`. Basedir of the fileset is [the workspace root](#). If no value is set, then the default `**/checkstyle-result.xml` is used. Be sure not to include any non-report files into this pattern.

Advanced...

Publish FindBugs analysis results

FindBugs results Documentacion/IntegracionContinua/findbugs_report.xml

[Fileset includes](#) setting that specifies the generated raw FindBugs XML report files, such as `**/findbugs.xml` or `**/findbugsXml.xml`. Basedir of the fileset is [the workspace root](#). If no value is set, then the default `**/findbugsXml.xml` or `**/findbugs.xml` are used for maven or ant builds, respectively. Be sure not to include any non-report files into this pattern.

Use rank as priority ☐

Uses the bug rank when evaluating the priority of the warnings (otherwise the FindBugs priority is used).

Advanced...

Estos plugin permiten realizar acciones como por ejemplo setear el build como fallido si se detectan más errores que en el build anterior.

☒ Compute new warnings (based on the last successful build unless another reference build is chosen below)

Status thresholds (New warnings)	All priorities	Priority high	Priority normal	Priority low
	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>
	<input type="text" value="1"/>	<input type="text" value="1"/>	<input type="text" value="1"/>	<input type="text" value="1"/>

If the specified number of new warnings exceeds one of these thresholds then a build is considered as unstable or failed, respectively. I.e., a value of 0 means that the build status is changed if there is at least one new warning found. Leave this field empty if the state of the build should not depend on the number of new warnings.

Use delta for new warnings ☐

If set the number of new warnings is computed by subtracting the total number of warnings of the reference build from the total number of warnings of the current build. This may lead to wrong results if you have both fixed and new warnings in a build. If unset the number of new warnings is computed by a more sophisticated algorithm: instead of using totals an asymmetric set difference of the warnings in the current build and the warnings in the reference build is used. This will find all new warnings even if the number of total warnings has decreased. Note that sometimes false positives will be reported due to minor changes in a warning (e.g. refactoring of variables or method names). It is recommended to uncheck this option in order to get the most accurate results for new warnings.

Use previous build as reference ☒

Por último también se pueden instalar complementos que recopilan los diferentes resultados de análisis y los muestran en un gráfico de tendencias combinado. Además, proporcionan informes de salud y estabilidad de construcción basados en estos resultados combinados.

Publish combined static analysis results

X

Checkstyle warnings

☒

Duplicate code warnings

☒

FindBugs warnings

☒

PMD warnings

☒

Advanced...

Aggregate downstream test results

X

☒ Automatically aggregate all downstream tests

☒ Include failed builds in results

Add post-build action ▼

La siguiente imagen muestra la interfaz del proyecto donde se puede observar que en el menú de la izquierda se agregan links de programas de análisis de código estático para obtener más información del análisis, también se observa un resumen de los resultados del análisis, un historial de build y la carpeta workspace donde se encuentra el código fuente.

Jenkins

Jenkins > Arkanoid >

[Back to Dashboard](#)
[Status](#)
[Changes](#)
[Workspace](#)
[Build Now](#)
[Delete Project](#)
[Configure](#)
[Git Polling Log](#)
[GitHub](#)
[Checkstyle Warnings](#)
[FindBugs Warnings](#)
[PMD Warnings](#)
[Duplicate Code](#)
[Static Analysis Warnings](#)

Project Arkanoid

[Workspace](#)
[Recent Changes](#)
[Latest Aggregated Test Result](#) (no tests)

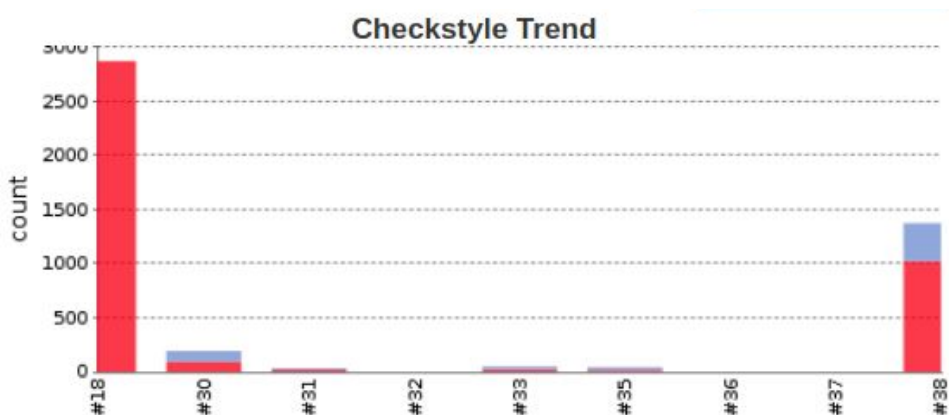
Analysis results

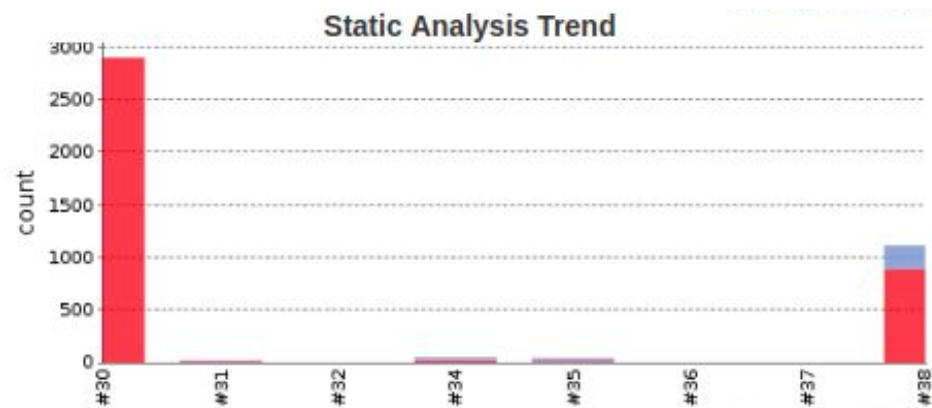
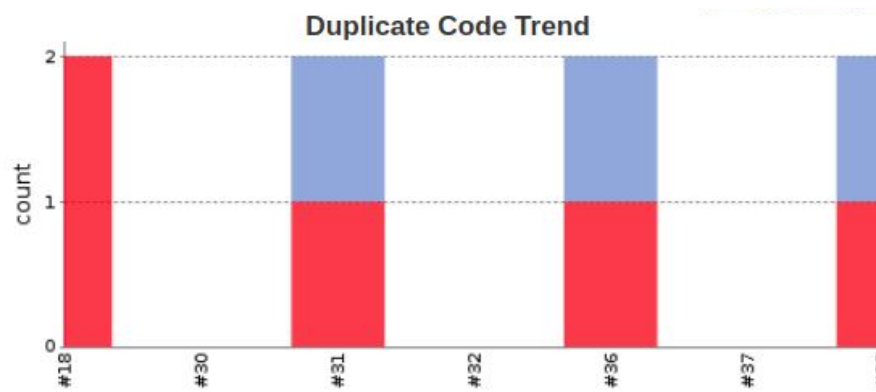
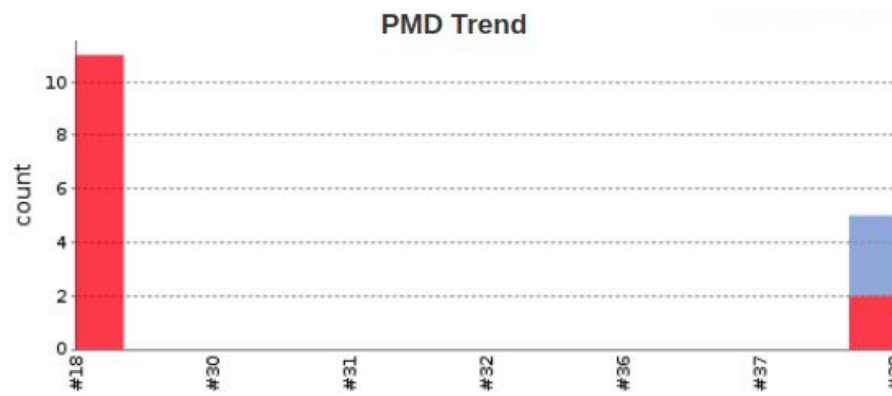
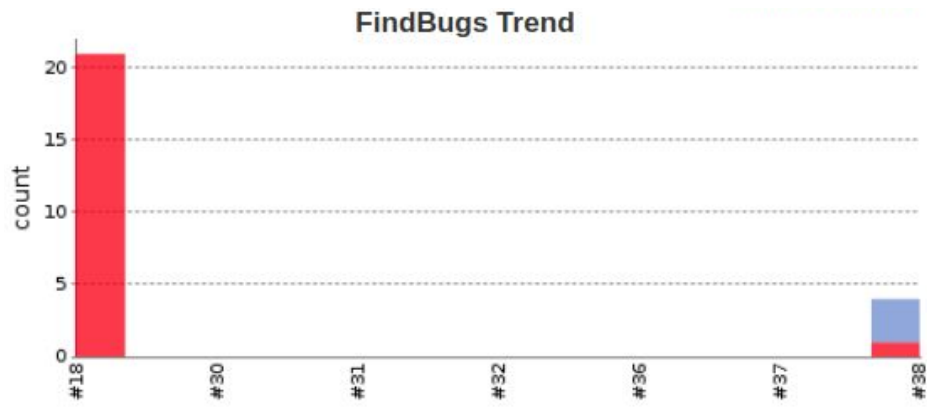
- Checkstyle Warnings: [3523](#)
- Duplicate Code: [2](#)
- FindBugs Warnings: [19](#)
- PMD Warnings: [10](#)

Permalinks

[Build History](#)
[trend](#)

En la interfaz también se muestran los gráficos correspondientes de las herramientas integradas, en el eje de las abscisas se muestran el número de build mientras que en el eje de las ordenadas el resultado del análisis. En este caso se configuró para que las gráficas muestren los warnings arreglados(celeste) y los nuevos(rojo).





A continuación se muestran la recopilación de datos del proyecto donde se destaca que el proyecto tiene una cantidad total de 3554 warnings y 9 de ellos son de alta prioridad, el software también nos muestra en donde se encuentran y los discrimina por categoría entre otras cosas.

Static Analysis Result

Warnings Trend

All Warnings	New Warnings	Fixed Warnings
3554	891	221

Summary

Total	High Priority	Normal Priority	Low Priority
3554	9	3543	2

Details

Folders	Files	People	Categories	Types	Warnings	Origin	Details	New	Fixed	High	Normal	Low
Source Folder				Total	Distribution							
./src/Beat				624								
./src/Heart				390								
./src/Menu				58								
./src/Strategy				180								
./src/UnitTest				902								
./src/arkanoid MVC				1177								
./src/arkanoid beat				194								
Beat				4								
Heart				5								
Menu				4								
arkanoid MVC				9								
arkanoid beat				7								
Total				3554								

Folders	Files	People	Categories	Types	Warnings	Origin	Details	New	Fixed	High	Normal	Low
Category				Total	Distribution							
(none)				27								
BAD PRACTICE				5								
Blocks				40								
CORRECTNESS				2								
Coding				18								
Imports				106								
Indentation				1355								
Javadoc				64								
MT_CORRECTNESS				2								
Naming				103								
PERFORMANCE				4								
STYLE				6								
Sizes				29								
Unused Code				10								
Whitespace				1783								
Total				3554								

Conclusión

En este trabajo se realizó un análisis del proyecto “Arkanoid” desarrollado en la materia Ingeniería de Software comprendiendo todos los componentes de la calidad de software como también los procedimientos para realizar el análisis.

Al final la materia Ingeniería de Software los desarrolladores encontraron solo dos defectos en el proyecto contrastado con los 79 errores que se espera encontrar en un software de este tamaño, por lo que se puede deducir que faltaron casos de testing. Siguiendo con el análisis también se concluye que es necesario volver a redactar los requerimientos y que los gráficos UML están bien aunque faltan algunos. Luego se utilizó el software kiuwan para la revisión de código y se encontraron 103 defectos aunque en esta etapa se esperaba encontrar 20, esta diferencia es debida a que los programadores eran inexpertos y no contaban con herramientas de este tipo para corregir estos defectos inmediatamente. Por último se implementó una herramienta de integración, Jenkins, que permite automatizar el control de errores, esta retroalimentación es muy importante tanto para el propio programador como para el equipo ya que promueve que se sigan reglas y estándares y asegura que el proyecto siempre sea estable.

Para finalizar podemos decir que el seguimientos de estandares de calidad a la hora de desarrollar el producto puede resultar mucho menos costoso que si se trata de obtener un proyecto de calidad una vez finalizado el mismo.

Referencias

Proyecto Arkanoid

https://github.com/estebantissot/arkanoid_MVC.git

JaCoCo

<http://www.jacoco.org/jacoco/trunk/doc/counters.html>

Kiuwan

<https://www.kiuwan.com/>

Jenkins

<https://www.digitalocean.com/community/tutorials/how-to-install-jenkins-on-ubuntu-16-04>

Compilar con javac

<http://www.codigorecursivo.com/compilar-y-ejecutar-ficheros-java-desde-la-consola.html>

PMD

<http://pmd.sourceforge.net/pmd-5.4.1/>

PMD - CPD

<http://pmd.sourceforge.net/pmd-5.4.1/usage/cpd-usage.html>

FINDBUGS

<http://findbugs.sourceforge.net/manual/running.html>

checkstyle

<http://checkstyle.sourceforge.net/>

Apendice

Jenkins

Instalación en Ubuntu 16.04
<pre>/* Se descargan los repositorios necesarios */ \$ wget -q -O - https://pkg.jenkins.io/debian/jenkins-ci.org.key sudo apt-key add - \$ echo deb http://pkg.jenkins.io/debian-stable binary/ sudo tee /etc/apt/sources.list.d/jenkins.list /* Se instala Jenkins */ \$ sudo apt-get update \$ sudo apt-get install jenkins /* Se inicializa el servicio */ \$ sudo systemctl start jenkins \$ sudo systemctl status jenkins /* Se abre el puerto 8080 */ \$ sudo ufw allow 8080 /* Se obtiene la clave temporal que Jenkins generó para poder configurarlo */ \$ sudo cat /var/lib/jenkins/secrets/initialAdminPassword /* Se debe configurar Jenkins ingresando en http://ip_address_or_domain_name:8080 y colocando la clave temporal */</pre>

Java

Instalación de Java
<pre>\$ sudo apt-get install software-properties-common \$ sudo apt-add-repository ppa:webupd8team/java \$ sudo apt-get update \$ sudo apt install oracle-java9-installer</pre>