

# Gestión de la Calidad del Software.

## Trabajo Práctico Final



### Alumnos:

- Kleiner Matías ([kleiner.matias@gmail.com](mailto:kleiner.matias@gmail.com)),
- López Gastón ([gopezlaston@gmail.com](mailto:gopezlaston@gmail.com)),
- Maero Facundo ([facundojmaero@gmail.com](mailto:facundojmaero@gmail.com)),
- Rivero Franco ([francorivero2012@gmail.com](mailto:francorivero2012@gmail.com))

**Nombre de grupo:** Hello World

**Docente:** Ing. Micelli Martín, Ing. Nonino Julián.

**Facultad de Ciencias Exactas, Físicas y Naturales**

**Universidad Nacional de Córdoba**

**Año 2018**

## Control de versiones del documento

<b>Versión</b>	<b>Resumen cambios</b>	<b>Autores de los cambios</b>	<b>Fecha</b>
1.0.0	Primera versión	Maero Facundo, López Gastón, Kleiner Matías, Rivero Franco	24/10/2018

# Índice

## Trabajo Práctico Final.

### 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.

### 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 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.

Conclusión.

Anexo.

# Trabajo Práctico Final.

## 1. Plan de Remoción de Defectos.

1.1. Utilice una herramienta como LocMetrics para contar la cantidad de líneas de código de su proyecto.

Para el desarrollo de este trabajo práctico, se utilizó el proyecto final de Ingeniería de Software, que consistió en tomar el código del libro Head First Design Patterns utilizado para ilustrar el patrón de arquitectura Model View Controller. Se creó una vista nueva, ligada a una nueva funcionalidad. En este caso, se codificó un reproductor de archivos mp3.

Como el proyecto está formado por archivos de diferente proveniencia, se analizó puntualmente el contenido de los codificados por los alumnos, dejando de lado el que provee el libro Head First.

Se utilizó la herramienta LocMetrics para realizar un conteo de líneas de código del proyecto. El gráfico siguiente muestra una visión global, donde se observa que las carpetas *controllers* y *views* concentran más del 75% de las líneas de código de todo el proyecto.

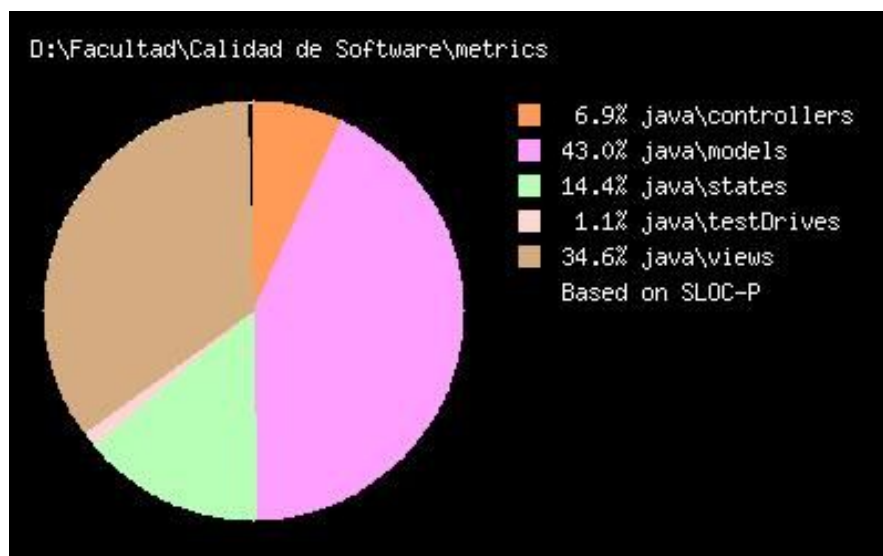


Figura 1 - Gráfico obtenido de LocMetrics.

En el recuento general, se observa que en 17 archivos hay un total de 1323 líneas de código, de las cuales 1065 son ejecutables (al no tener en cuenta comentarios y líneas en blanco).

Overall		
Symbol	Count	Definition
Source Files	17	Source Files
Directories	10	Directories
LOC	1323	Lines of Code
BLOC	203	Blank Lines of Code
SLOC-P	1065	Physical Executable Lines of Code
SLOC-L	844	Logical Executable Lines of Code
MVG	65	McCabe VG Complexity
C&SLOC	23	Code and Comment Lines of Code
CLOC	55	Comment Only Lines of Code
CWORD	478	Commentary Words
HCLOC	0	Header Comment Lines of Code
HCWORD	0	Header Commentary Words

Figura 2 - Recuento de líneas de código

1.2. Elabore un plan de remoción de defectos, estimando la cantidad de defectos que deberían encontrarse en cada etapa.

El plan de remoción de defectos elegido es la variante Integral que incluye, además de todo lo comprendido en el plan estándar, inspecciones de diseño y de código. Esta decisión se tomó ya que es sabido que el costo de remover un defecto aumenta a medida que avanzan las diferentes etapas del proyecto, por lo que es deseable detectar la mayor cantidad posible en cada etapa, y evitar que pasen a la siguiente.

### Plan de efectividad de remoción de defectos<sup>[1/2]</sup>

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%
<b>Inspecciones de Diseño</b>	-----	<b>70%</b>
Revisiones de Diseño	50%	60%
<b>Inspecciones de Código</b>	-----	<b>70%</b>
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%

Figura 3 - Comparación en la efectividad de remoción de defectos de ambos planes.

En la tabla se observa una comparativa entre la efectividad de remoción de defectos de ambos planes.

La cantidad de defectos que se espera encontrar en un proyecto de esta magnitud, si se tiene en cuenta que hay 1 error por cada 10 acciones ejecutadas (100 errores por cada 1000 LOC), es de 107 defectos (1065 LOC / 10). Se adoptó esta métrica teniendo en cuenta que el equipo es inexperto, y no tiene experiencia desarrollando proyectos similares.

Además se tuvo en cuenta la siguiente tabla, que muestra una aproximación de la cantidad de defectos originados en cada etapa, y el costo relativo de su corrección.

## Origen de los defectos y sus costos

Fase de Desarrollo	%Promedio de defectos originados por fase	Costo promedio relativo de remover un defecto
Especificación de Requerimientos	15%	1
Diseño	35%	2.5
Codificación Unitaria	30%	6.5
Codificación de la Integración	10%	16
Documentación	10%	16
Pruebas de Sistema	-----	40
Software en Operación	-----	110

Figura 4 - Origen de los defectos y sus costos.

El resultado obtenido a partir de estos datos arroja los siguientes valores:

Etapas	Defectos introducidos	Defectos Arrastrados	Defectos totales	Defectos corregidos
Especificación de Requerimientos	16	0,0	16,0	9,6
Inspecciones de Diseño	37	6,4	43,4	30,7
Revisiones de Diseño	0	13,2	13,2	7,9
Inspecciones de Código	32	5,3	37,3	26,2
Pruebas Unitarias	0	11,2	11,2	4,5
Pruebas de Integración	11	6,7	17,7	10,5
Revisión de Documentación	11	7,0	18,0	10,6
Pruebas de Sistema	0	7,1	7,1	4,2
Software en Operación	0	2,8	2,8	2,8
<b>Total</b>	<b>107</b>	<b>-</b>	<b>-</b>	<b>107,0</b>

Figura 5 - Tabla indicativa del plan de remoción de defectos.



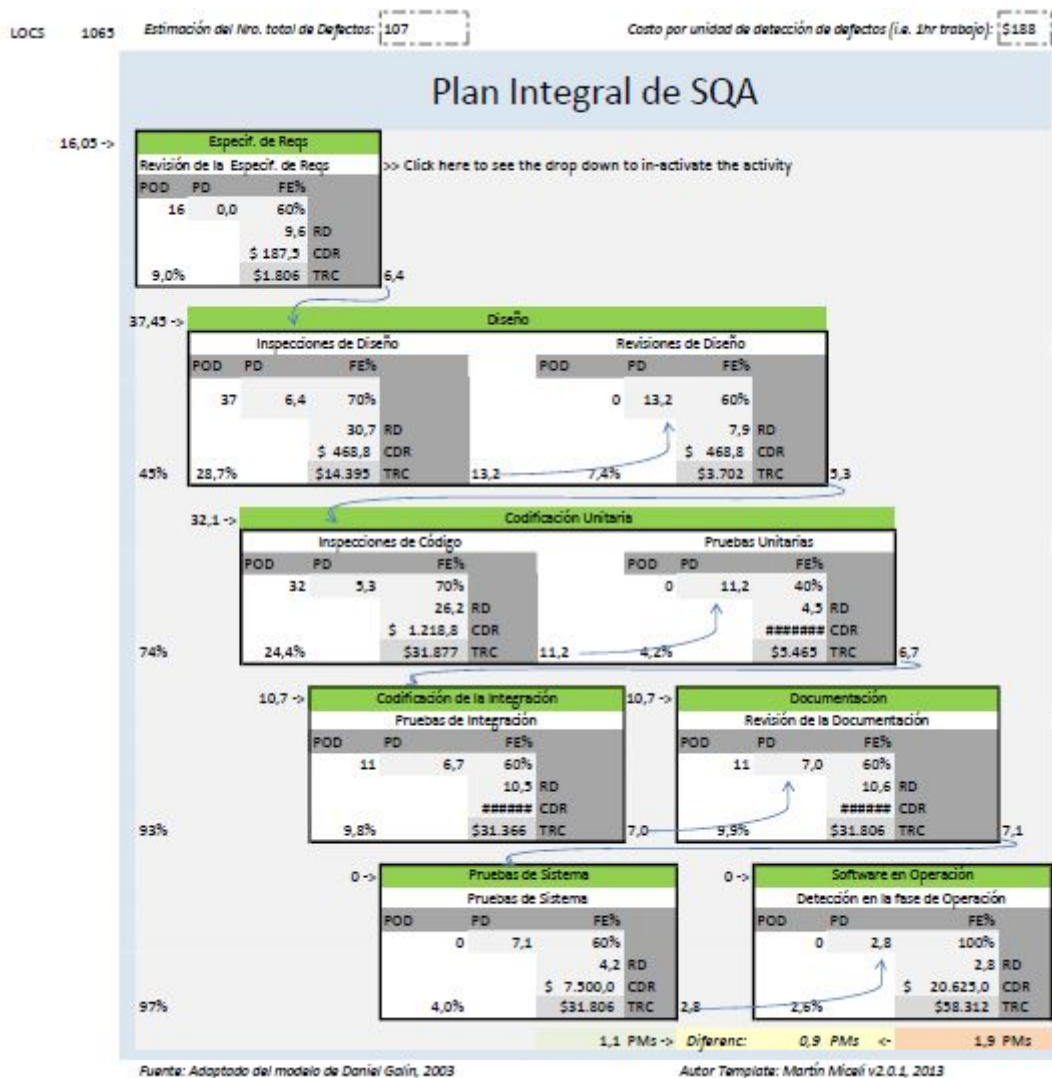


Figura 6 - Planilla de Excel utilizada para completar la tabla de la Figura 5.

Cabe destacar que el costo por hora de trabajo es de \$188 debido a que se considera un sueldo mensual de \$30.000 y 160 hs de trabajo por mes por persona (8 hs x 20 días).

Por otra parte, el costo total de remoción de los defectos encontrados sin tener en cuenta la etapa de release, es de \$152.223.

1.3. Compare la cantidad de defectos estimada con la cantidad de defectos que encontró al finalizar la materia Ingeniería de Software.

En el repositorio del proyecto de Ingeniería de Software se puede ver que se encontraron y documentaron 11 issues en GitHub. De los 107 que se estima que fueron introducidos en el proyecto, 104 defectos deberían haberse detectado y



corregido antes de entregar el producto final. (Ver los 11 issues en <https://github.com/castagno/IngSoft-2016-HelloWorld/issues>).

Esto contrasta mucho con los 11 encontrados en realidad. A pesar de ello, al no seguir una metodología dada (en Ingeniería de Software), cuando se encontraba un defecto muchas veces se lo corregía sin dejar ninguna documentación de ello, por lo que es muy probable que el número encontrado y corregido sea mucho mayor, acercándose al valor teórico estimado. Nuevamente, al no tener documentación es imposible saberlo con seguridad.

## **2. Revisión de los requerimientos.**

### **2.1. ¿Faltan requerimientos?**

#### Requerimientos funcionales que faltan:

- El sistema deberá permitir continuar la reproducción de la canción siguiente una vez finalizada la actual en forma automática.

#### Requerimientos no funcionales que faltan:

- No hay detalles acerca de la duración mínima asegurada de funcionamiento del programa. (Confiabilidad).
  - Test posible: dado un requerimiento de duración mínima de, por ejemplo, 8 horas, armar un set de canciones a reproducir y poner a funcionar el programa por 8 horas. Durante las mismas, se debe modificar el curso de reproducción, y ejecutar acciones sobre las canciones (pausa, stop, etc.). Realizar este test, por lo menos, 3 veces con distinto número o cantidad de canciones en el set (1, 10, 50).
- No hay detalles acerca de requisitos legislativos, como por ejemplo, las posibles licencias que se pueden utilizar para el desarrollo del código.

### **2.2. ¿Sobran requerimientos?**

No, no sobra ningún requerimiento.

### **2.3. ¿Está correctamente escritos los requerimientos?**

Salvo los requerimientos funcionales 11, 12 y 13, los demás sí. En estos requerimientos se debe modificar el verbo “debería” por “debe” o “deberá”. Esto igualmente se aplica a los requerimientos no funcionales 1, 2 y 3. Además, en el

requerimiento funcional número 13 se debe especificar que el volumen mínimo es igual a volumen cero.

2.4. Elabore los reportes de revisión necesarios.

Reporte de Revisión de Requerimientos			
Fecha: 10-oct-18		Preparado por: 1-2-3-4	
Nombre del Proyecto: Reproductor MP3			
Documento revizado: SRS.md (System Requirements Specification)		Versión: 1.3.0	
<b>1. Resumen de las discusiones</b>			
#	Temas de la discusión	Nro. acciones	
1	Requerimientos que sobran	No	
2	Requerimientos funcionales que faltan	1	
3	Requerimientos no funcionales que faltan	2,3	
4	Requerimientos funcionales correctamente escritos	4,5	
5	Requerimientos no funcionales correctamente escritos	6	
<b>2. Listado de Acciones</b>			
#	Acciones	Responsable	Fecha de Fin
1	Colocar requerimiento de reproducción automática.		
2	Colocar req. de duración mínima de funcionamiento del programa.		
3	Colocar requerimientos legislativos.		
4	En los req. 11, 12 y 13, modificar "debería" por "debe" o "deberá"		
5	En el req. 13, especificar cuánto es el volumen mínimo		
6	En los req. 1, 2 y 3, modificar "debería" por "debe" o "deberá"		
<b>3. Decisión sobre el documento revizado</b> (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:			
Secciones 1, 2, 3 (Definición de requerimientos de usuario), 6.1.			
Requerimientos funcionales de sistema del 1 al 10.			
<input type="checkbox"/>	Desaprobado		
<b>4. Participantes</b>			
#	Nombre y Apellido	Rol	Firma
1	López Gastón	Revisor	López G
2	Kleiner Matías	Revisor	Kleiner M
3	Rivero Franco	Revisor	Rivero F
4	Maero Facundo	Autor	Maero F

Figura 7 - Planilla de Excel utilizada para la revisión de los requerimientos.

2.5. ¿En ésta etapa, se encontró la cantidad de defectos esperada de acuerdo a su plan de remoción de defectos?

De acuerdo al plan de remoción de defectos diagramado, se esperaban encontrar en esta etapa 9,6 defectos, y se hallaron 10, por lo que la revisión de requerimientos arrojó resultados satisfactorios y acordes a lo estimado.

### 3. Revisión del diseño.

3.1. Revise todos los documentos de diseño elaborando los reportes de revisión.

Reporte de Revisión de Diseño			
Fecha: 17-oct-18		Preparado por: 1-2-3-4	
Nombre del Proyecto: Reproductor MP3			
Documento revisado: Informe Ingeniería de Software.pdf		Versión: 1.2.0	
<b>1. Resumen de las discusiones</b>			
#	Temas de la discusión	Nro. acciones	
1	Diagrama de Arquitectura	1	
2	Diagrama de Clases - Modelos	2	
3	Diagrama de Clases - Vistas	3	
4	Diagrama de Actividades	4	
5	Diagramas de Secuencia	5	
6	Diagrama de Deployment	6	
7	Librerías Externas	7	
<b>2. Listado de Acciones</b>			
#	Acciones	Responsable	Fecha de Fin
1	Rehacerlo para que refleje con más detalle el funcionam. del SW		
2	Reducir lista de métodos para que sea más legible		
3	Separar ambas GUIs en dos diagramas diferentes		
4	No se muestra como terminar el programa (nodo de fin)		
5	Agregar diag. que muestre mejor como funciona el patrón State		
6	Mostrar relación entre test drives y librerías de 3ros		
7	Detallar si son mantenidas, abandonadas, versión usada.		
<b>3. Decisión sobre el documento revizado (ponga una X en donde corresponda)</b>			
<input type="checkbox"/> Aprobado <input checked="" type="checkbox"/> Parcialmente Aprobado. Las partes aprobadas para proseguir a la siguiente fase son las siguientes: <div style="border: 1px solid black; padding: 5px; margin: 5px 0;">Secciones 1, 2, 3, 4, 7, 8, 9</div>			
<input type="checkbox"/> Desaprobado			
<b>4. Participantes</b>			
#	Nombre y Apellido	Rol	Firma
1	López Gastón	Revisor	López G
2	Kleiner Matias	Revisor	Kleiner M
3	Rivero Franco	Revisor	Rivero F
4	Maero Facundo	Autor	Maero F

Figura 8 - Planilla de Excel para revisión del *Informe Ingeniería de Software.pdf*.

### Reporte de Revisión de Diseño

Fecha: 17-oct-18 Preparado por: 1-2-3-4  
 Nombre del Proyecto: Reproductor MP3  
 Documento revisado: CM Plan.md Versión: 1.2.0

#### 1. Resumen de las discusiones

#	Temas de la discusión	Nro. acciones
1	Configuration Identification	1
2	Gestión de la Configuración del Código Fuente	2, 3, 4

#### 2. Listado de Acciones

#	Acciones	Responsable	Fecha de Fin
1	Agregar diagrama de árbol de directorios para más claridad		
2	Contemplar el caso en el que es necesario un hotfix desde una rama diferente a la master. Explicar política de merge.		
3	Actualizar diagrama explicativo de merge policy		
4	Actualizar explicación de hotfix tags		

#### 3. Decisión sobre el documento revizado (ponga una X en donde corresponda)

☐ Aprobado  
☒ Parcialmente Aprobado.

Las partes aprobadas para proseguir a la siguiente fase son las siguientes:

Secciones 1, 2, 3, 5, 7, 8 y 9
Secciones 6.1.5, 6.1.3

☐ Desaprobado

#### 4. Participantes

#	Nombre y Apellido	Rol	Firma
1	López Gastón	Revisor	López G
2	Kleiner Matías	Revisor	Kleiner M
3	Rivero Franco	Revisor	Rivero F
4	Maero Facundo	Autor	Maero F

Figura 9 - Planilla de Excel para revisión del CM plan.

3.2. ¿En ésta etapa, se encontró la cantidad de defectos esperada de acuerdo a su plan de remoción de defectos?

Nuestro plan de remoción de defectos espera encontrar 39 defectos, de los cuales se encontraron 7 en el informe final, y 4 en el configuration management plan. Como se tiene incertidumbre en cuanto a la cantidad de defectos filtrados en la materia Ingeniería de Software, suponemos que allí se han filtrado 28 defectos.

Cabe mencionar que al revisar el informe final del trabajo práctico se tuvieron en cuenta solamente las secciones de diseño, ya que el mismo abarca requerimientos, diseño, implementación, testing, etc.

## 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 ).

Se optó por utilizar la herramienta Crucible de Atlassian para efectuar la revisión del código fuente del proyecto. Se eligió dicha herramienta debido a que nos resultó más fácil de utilizar, además de tener una GUI muy amigable. Para comenzar, se debieron crear usuarios y enlazar la herramienta con el repositorio de GitHub utilizado para la materia Gestión de la Calidad de Software.

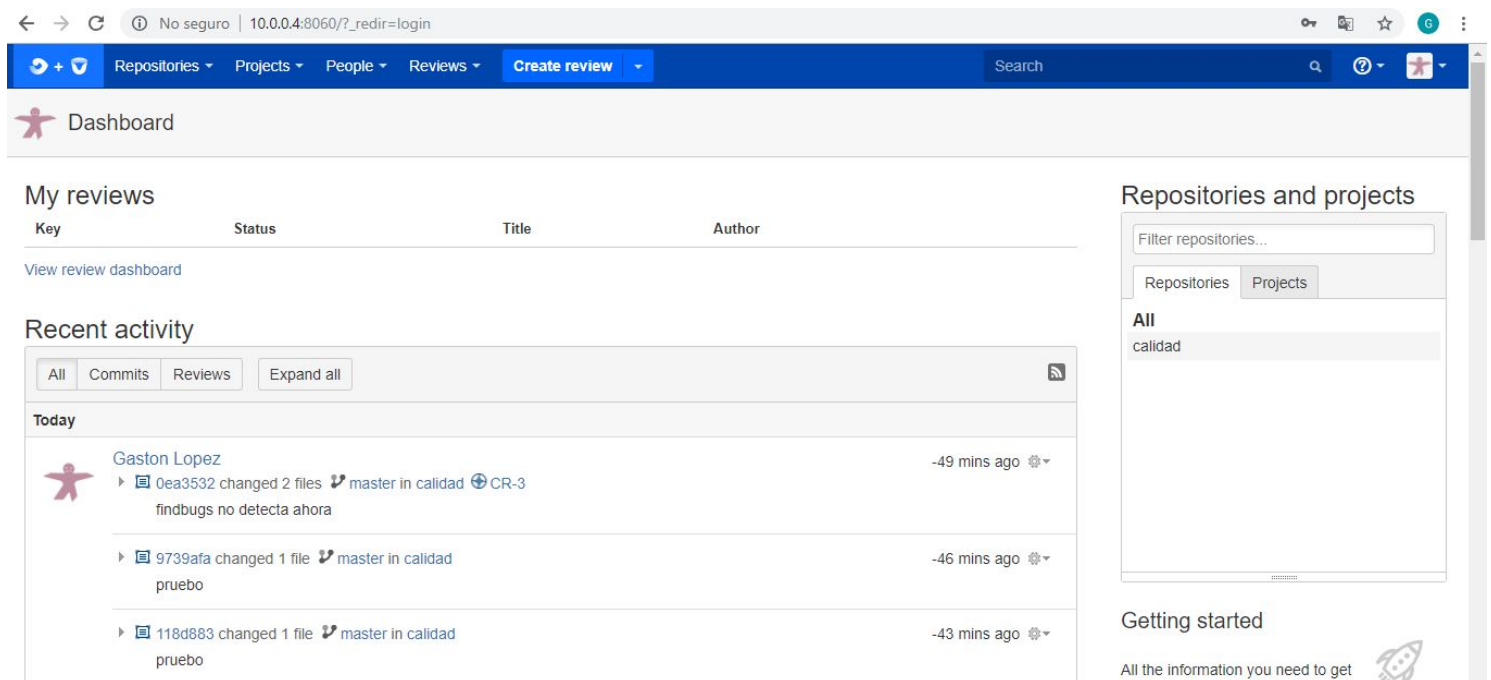


Figura 10 - Dashboard de la herramienta Crucible instanciada a partir de un Docker. Creación de usuario y enlace con el repositorio.

El paso siguiente fue comenzar a revisar el código fuente del proyecto de Ingeniería de Software. Para ello, se tildaron los archivos a revisar y se comenzó por cada archivo a encontrar defectos entre los 4 integrantes del equipo:



```

10
11     public void increase(){
12         setValue(getValue()+1);
13         repaint();
14     }
15
16     public void reset(){
17         setValue(0);
18         repaint();
19     }
20
21     public void setMax(int max){

```



**Matias Kleiner**

Agregar control sobre max.

**DEFECT**

Classification: Extra (superfluous)

Ranking: Minor

Reply · Mark as unread · Add to favourites · 17:28

Matias Kleiner marked as **UNRESOLVED** 17:28 [Resolve](#)

Figura 11 - Revisión de archivo en la herramienta Crucible.

[Unresolved](#) 24

[Resolved](#) 0

Number of files included: 17

calidad

- ▼ **src/main/java**
  - ▼ **controllers**
    - ControllerInterface.java
    - MP3Controller2.java 3
  - ▼ **models**
    - MP3Model.java 8
    - MP3ModelInterface.java
    - ProgressThread.java 3
  - ▼ **states**
    - EmptyState.java
    - MP3State.java
    - PausedState.java
    - PlayingState.java
    - StoppedState.java
  - ▼ **testDrives**
    - MyMP3TestDrive2.java
  - ▼ **views**
    - BPMObserver.java
    - BeatObserver.java
    - MP3View.java 9
    - ProgressBar.java 1
    - ProgressObserver.java
    - TrackObserver.java

Figura 12 - Listado de archivos y número de defectos encontrados en cada uno.

A continuación se muestra un reporte que arroja la herramienta, mostrando los defectos encontrados por archivo, y los comentarios en cada uno.

---

**File: src/main/java/controllers/MP3Controller2.java**

Agregar control sobre la variable "volumen". El máximo debe ser 1, no existe control sobre el limite minimo o maximo del mismo.

No existe control sobre el Path, agregar try y catch.

Agregar try y catch en caso de que el índice sea inválido.

---

**File: src/main/java/models/MP3Model.java**

No existe control sobre el parámetro path.

Siempre se espera al menos un archivo finalizado en ".mp3". No hay control en lo que sucede si no se encuentra uno.

Se debe permitir canciones superiores a los 60 minutos.

Agregar un control sobre la variable estática "index".

No debe retornar null, sino un tipo de dato byte[] vacío.

Agregar control sobre parámetro index.

Agregar cierto control sobre la variable index.

Agregar control sobre parametro.

---

**File: src/main/java/models/ProgressThread.java**

Se debe proteger la escritura concurrente de la variable value.

Se debe proteger la escritura concurrente de la variable playlist.

Se debe proteger la variable maximum de la escritura concurrente.

---

**File: src/main/java/views/MP3View.java**



No existe un control sobre los path utilizados.

Abstraer todas estas asignaciones en un método.

Todas las constantes de la GUI deben estar definidas en variables estáticas, constantes y parametrizables.

Definir los strings como constantes, para mayor parametrización.

Método comienza con minúscula.

Método debe comenzar con minúscula.

Método debe comenzar con minúscula.

No existe ningún control sobre los parámetros.

Agregar caso para más de 60 minutos.

-----  
**File: src/main/java/views/ProgressBar.java**

Agregar control sobre max.

#### 4.2. ¿Cuántos defectos encontraron? ¿Fueron encontrados todos los defectos que esperaba encontrar?

Se encontraron 24 defectos en la revisión de código, de los 26 que se estimaron en el plan de remoción de defectos. Esta diferencia es aceptable, teniendo en cuenta que algunos errores pueden haberse filtrado ya en la materia Ingeniería de Software.

## 5. Pruebas Unitarias

### 5.1. ¿Qué cantidad de defectos espera encontrar en ésta etapa?

En la etapa general de codificación unitaria deberían descubrirse aproximadamente 5 defectos según el plan de remoción previamente descrito.

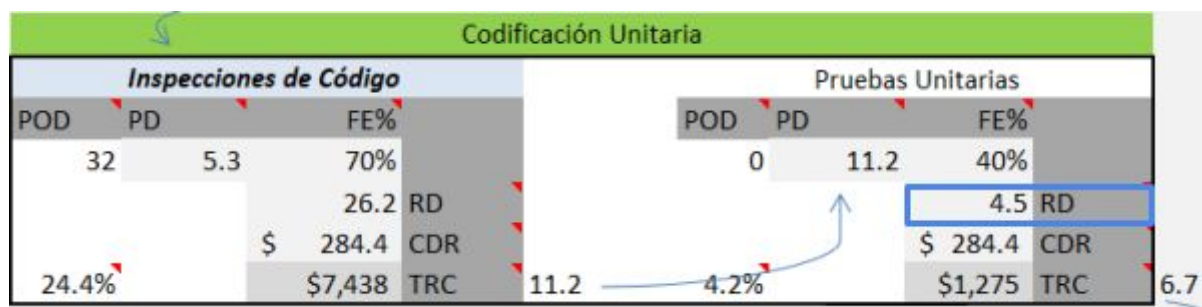


Figura 13 - Sección de codificación unitaria en el plan de remoción de defectos.

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?

Para medir la cobertura del código fuente al ejecutar los tests se utilizó la última versión de Eclipse Photon, que posee preinstalado el plugin Java Code Coverage 3.1.1.

Los resultados obtenidos se muestran a continuación. Cabe mencionar que no fue posible configurar el plugin para que omita en el conteo los archivos de test (paquetes **test.java.\***), por lo que el porcentaje visible en las capturas de pantalla es incorrecto. Su valor real se calcula tomando solamente los paquetes **main.java.\***.

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
✓ Mp3-Player-HelloWorld	56.6 %	2,260	1,736	3,996
✓ src	56.6 %	2,260	1,736	3,996
> main.java.views	0.0 %	0	1,250	1,250
✓ main.java.models	62.7 %	569	339	908
> MP3Model.java	60.4 %	499	327	826
> ProgressThread.java	85.4 %	70	12	82
> main.java.controllers	0.0 %	0	97	97
✓ main.java.states	92.6 %	261	21	282
> PausedState.java	92.0 %	92	8	100
> PlayingState.java	90.7 %	78	8	86
> EmptyState.java	77.8 %	14	4	18
> StoppedState.java	98.7 %	77	1	78
> main.java.testDrives	0.0 %	0	20	20
> test.java.controllers	0.0 %	0	6	6
> test.java.views	0.0 %	0	3	3
> test.java.models	100.0 %	568	0	568
> test.java.states	100.0 %	862	0	862

Figura 14 - Resultados obtenidos del Instruction Coverage en el Eclipse Photon con el plugin Java Code Coverage 3.1.1.

De 2557 instrucciones, fueron cubiertas 830 (569 + 261).

Instruction coverage del proyecto principal: **32,46%**

Element	Coverage	Covered Lines	Missed Lines	Total Lines
▼ Mp3-Player-HelloWorld	57.7 %	586	430	1,016
▼ src	57.7 %	586	430	1,016
> main.java.views	0.0 %	0	278	278
▼ main.java.models	63.9 %	161	91	252
> MP3Model.java	61.3 %	136	86	222
> ProgressThread.java	83.3 %	25	5	30
> main.java.controllers	0.0 %	0	39	39
▼ main.java.states	84.0 %	68	13	81
> EmptyState.java	60.0 %	6	4	10
> PausedState.java	84.6 %	22	4	26
> PlayingState.java	84.0 %	21	4	25
> StoppedState.java	95.0 %	19	1	20
> main.java.testDrives	0.0 %	0	6	6
> test.java.controllers	0.0 %	0	2	2
> test.java.views	0.0 %	0	1	1
> test.java.models	100.0 %	129	0	129
> test.java.states	100.0 %	228	0	228

Figura 15 - Resultados obtenidos del Line Coverage en el Eclipse Photon con el plugin Java Code Coverage 3.1.1.

De 656 líneas, fueron cubiertas 229 (161 + 68).

Line coverage del proyecto principal: **34,91%**

Element	Coverage	Covered Branches	Missed Branches	Total Branches
▼ Mp3-Player-HelloWorld	37.3 %	41	69	110
▼ src	37.3 %	41	69	110
> main.java.views	0.0 %	0	30	30
▼ main.java.models	54.8 %	34	28	62
> MP3Model.java	53.4 %	31	27	58
> ProgressThread.java	75.0 %	3	1	4
> main.java.controllers	0.0 %	0	8	8
▼ main.java.states	50.0 %	3	3	6
> PausedState.java	50.0 %	1	1	2
> PlayingState.java	50.0 %	1	1	2
> StoppedState.java	50.0 %	1	1	2
> EmptyState.java		0	0	0
> main.java.testDrives		0	0	0
> test.java.controllers		0	0	0
> test.java.models		0	0	0
> test.java.states	100.0 %	4	0	4
> test.java.views		0	0	0

Figura 16 - Resultados obtenidos del Branch Coverage en el Eclipse Photon con el plugin Java Code Coverage 3.1.1.

De un total de 106 branches, fueron cubiertos 37 (34 + 3).

Branch coverage del proyecto principal: **34,91%**

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?

**Line coverage:** expresa el porcentaje de líneas de código ejecutadas. Depende del formato de código.

```
int a = 0;
int b = 0;
int c = 0;

int d = 0; int e = 0; int f = 0;
```

Por ejemplo, las declaraciones anteriores son equivalentes. Java permite escribir múltiples instrucciones en la misma línea, usando el punto y coma como separador, pero la herramienta de conteo utilizada calcula en el primer caso 3 líneas, y en el segundo caso, solo una.

**Instruction coverage:** todas las instrucciones en Java se transforman en bytecode, que luego es ejecutado por la máquina virtual. Esta métrica indica el porcentaje de dicho bytecode que es ejecutado por un test. A diferencia del line coverage, es independiente del formato en el que esté escrito el código. A pesar de esto, la diferencia se suele mitigar, y ambas métricas son muy similares al utilizar estilos de código preestablecidos, que incentivan un estilo de codificación prolijo y predecible, priorizando la legibilidad del código por sobre una sintaxis más compacta.

**Branch coverage:** expresa el porcentaje de caminos ejecutados en un código fuente. Estos dependen de la cantidad de construcciones de control presentes (if, else, for, while, switch).

La métrica más útil para un proyecto de software es la de branch coverage, ya que por un lado engloba a las demás: si se tiene un 100% de branch coverage, se habrán cubierto todas las instrucciones en el proyecto. Por otro lado, es un mejor indicador sobre el nivel de cobertura de los distintos escenarios que pueden ocurrir al ejecutar un programa. Dicho de otro modo, puede suceder que se tenga un alto porcentaje de cobertura de instrucciones, pero estas corresponden solo a algunos branches, por lo que hay muchas situaciones no contempladas.

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.

El proyecto tiene un branch coverage del 34,91%, y su complejidad ciclomática tiene un valor de 65. Por otro lado, los tests unitarios codificados en Ingeniería de Software son 36, por lo que serían necesarias 29 pruebas adicionales para cubrir todos los caminos calculados con el índice de McCabe.

Overall		
Symbol	Count	Definition
Source Files	17	Source Files
Directories	10	Directories
LOC	1323	Lines of Code
BLOC	203	Blank Lines of Code
SLOC-P	1065	Physical Executable Lines of Code
SLOC-L	844	Logical Executable Lines of Code
MVG	65	McCabe VG Complexity
C&SLOC	23	Code and Comment Lines of Code
CLOC	55	Comment Only Lines of Code
CWORD	478	Commentary Words
HCLOC	0	Header Comment Lines of Code
HCWORD	0	Header Commentary Words

Figura 17 - Complejidad ciclomática de McCabe.

Si se observa la complejidad de cada archivo, puede verse que la mayoría de ellos son muy simples, con excepción de la clase MP3Model.java, que contiene la mayor parte de la lógica de nuestro modelo de un reproductor musical, y una complejidad de 48. A pesar de esto, su alto valor se encuentra distribuido entre sus 42 métodos, por lo que no se considera necesario refactorizar ninguno de ellos.



D:\Facultad\Calidad de Software\metrics - FILES										
File	LOC	SLOC Physical	SLOC Logical	MVG	BLOC	C&SLOC	CLOC	CWORD	HCLOC	HCWORD
D:\Facultad\IngSoft-2016-HelloWorld\src\main\java\controllers\ControllerInterface.java	14	8	7	0	6	0	0	0	0	0
D:\Facultad\IngSoft-2016-HelloWorld\src\main\java\controllers\MP3Controller2.java	82	66	44	4	14	3	2	41	0	0
D:\Facultad\IngSoft-2016-HelloWorld\src\main\java\models\MP3Model.java	437	371	278	48	50	6	16	165	0	0
D:\Facultad\IngSoft-2016-HelloWorld\src\main\java\models\MP3ModelInterface.java	65	35	34	0	30	0	0	0	0	0
D:\Facultad\IngSoft-2016-HelloWorld\src\main\java\models\ProgressThread.java	58	52	40	3	4	3	2	29	0	0
D:\Facultad\IngSoft-2016-HelloWorld\src\main\java\states\EmptyState.java	35	22	13	0	12	0	1	11	0	0
D:\Facultad\IngSoft-2016-HelloWorld\src\main\java\states\MP3State.java	12	9	8	0	3	0	0	0	0	0
D:\Facultad\IngSoft-2016-HelloWorld\src\main\java\states\PausedState.java	56	45	32	1	10	0	1	11	0	0
D:\Facultad\IngSoft-2016-HelloWorld\src\main\java\states\PlayingState.java	57	45	32	1	11	0	1	11	0	0
D:\Facultad\IngSoft-2016-HelloWorld\src\main\java\states\StoppedState.java	45	32	23	1	10	0	3	13	0	0
D:\Facultad\IngSoft-2016-HelloWorld\src\main\java\testDrives\MyMP3TestDrive2.java	20	12	10	0	4	0	4	47	0	0
D:\Facultad\IngSoft-2016-HelloWorld\src\main\java\views\BeatObserver.java	6	4	3	0	2	0	0	0	0	0
D:\Facultad\IngSoft-2016-HelloWorld\src\main\java\views\BPMObserver.java	6	4	3	0	2	0	0	0	0	0
D:\Facultad\IngSoft-2016-HelloWorld\src\main\java\views\MP3View.java	393	333	296	7	35	11	25	150	0	0
D:\Facultad\IngSoft-2016-HelloWorld\src\main\java\views\ProgressBar.java	25	18	14	0	7	0	0	0	0	0
D:\Facultad\IngSoft-2016-HelloWorld\src\main\java\views\ProgressObserver.java	6	4	3	0	2	0	0	0	0	0
D:\Facultad\IngSoft-2016-HelloWorld\src\main\java\views\TrackObserver.java	6	5	4	0	1	0	0	0	0	0

Figura 18 - Complejidad ciclomática de McCabe según los distintos archivos .java del proyecto.

Al codificar las pruebas unitarias faltantes se apuntó a lograr un 80% de instruction coverage en todo el proyecto, priorizando los archivos con más complejidad, y procurando no testear métodos simples o triviales (por ejemplo, getters y setters, o métodos vacíos en clases que implementan el patrón State). Por otro lado, se procuró testear la mayor cantidad de branches posible para cubrir diferentes caminos lógicos a la hora de ejecutar el programa.

En la clase Mp3Model (complejidad ciclomática igual a 48) fue priorizado el testing de métodos ligados al comportamiento del reproductor, y no se chequearon aquellos relacionados al patrón Observer, ya que son idénticos al código provisto por el libro Head First Design Patterns.

Se decidió no testear la clase Mp3View (complejidad ciclomática igual a 7), dado que la mayoría de sus métodos están ligados a funcionalidad directa de la

librería Swing de Java. El resto está ligado a pasar acciones recibidas por el usuario al controlador, lo cual es revisado en detalle en los tests de sistema.

Se adicionaron un total de 24 tests unitarios.

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.

Defectos encontrados en el producto:

A continuación se nombran los defectos encontrados en el producto luego de implementar nuevas pruebas unitarias. La escala de severidad utilizada respeta lo visto en clase, tomado del sitio web de Bugzilla:

[https://wiki.documentfoundation.org/QA/Bugzilla/Fields/Severity#Severity\\_Levels](https://wiki.documentfoundation.org/QA/Bugzilla/Fields/Severity#Severity_Levels)

1. Mp3Model: la clase tiene un comportamiento silencioso a la hora de intentar agregar archivos incorrectos o no soportados a una playlist. Debería informar (por ejemplo, por consola) que el formato no es soportado (archivo .flac, .aac, etc).

**Severidad: Menor**

2. Mp3Model: sucede algo similar a la hora de solicitar la portada del álbum, ya que si el archivo no tiene embebida la imagen jpg, el método retorna null (sería más útil un mensaje por consola).

**Severidad: Menor**

3. Mp3Controller2: se observa que las clases están muy acopladas, aún utilizando el patrón Model View Controller, por lo que se dificulta realizar Unit Tests, ya que cada método interactúa con las demás clases.

**Severidad: Mejora**

4. Mp3Controller2: para testear esta clase es necesario pasarle en el constructor un modelo y una vista. No soporta recibir parámetros nulos, lo cual es otro indicador del alto acoplamiento entre ellas. Para realizar el “unit test” es necesario entonces construir un modelo y una vista.

**Severidad: Mejora**

5. Mp3Controller2: los métodos start, stop no son robustos, ya que directamente avisan al modelo que comience o detenga la música, sin antes realizar ningún tipo de chequeo o construcción try/catch.

**Severidad: Mayor**



6. Mp3View: contrario a la implementación estándar del patrón MVC, la clase view en este proyecto conoce al modelo, lo cual dificulta separar qué hace cada clase. Sería deseable una capa más fina en la vista, sin lógica y que solamente avisa al controlador sobre los inputs recibidos.

**Severidad:** Mejora

7. Al realizar los tests del controlador se observará una suba de coverage importante, sobre todo en la vista y el modelo. Esto sucede porque, como se comentó anteriormente, al estar acoplados, es imposible testearlo sin inicializar las demás clases, lo que implica ejecutar su código. Esto podría haberse evitado con un mejor diseño de las clases y sus relaciones desde el principio. Otra herramienta que podría haber sido útil es construir un stub del modelo por ejemplo, y usarlo a la hora de testear el controlador. Esto no se hizo ya que se considera que es un paso previo a la codificación del modelo mismo, por lo que directamente se lo utilizó en las pruebas.

**Severidad:** Mejora

8. En el repositorio se encuentra una carpeta con canciones de demo, utilizadas para fines demostrativos y para ejecutar los tests. Es necesario agregar nuevos archivos para lograr un coverage mayor y probar otros casos. Por ejemplo, todas las canciones ya incluidas poseen los datos de artista, álbum, carátula y demás, por lo que ciertos branches no pueden ser testeados a menos que se utilice otro archivo que no tenga estos datos. (Se puede tomar la decisión de utilizar un archivo solo con el tag de nombre).

**Severidad:** Mejora

9. Mp3Model: si el archivo no tiene tag de título, el método que arroja la playlist para ser mostrada en la vista falla con una excepción. En la codificación original se esperaba que el nombre fuera null, pero ese no es el comportamiento de la librería usada, que directamente arroja una NullPointerException. Esto no permite testear el caso en el que no se tienen tags.

**Severidad:** Crítico

En resumen:

Defecto	Severidad	Agregar	Falta	Incorrecto
1	2		x	
2	2		x	
3	0	x		
4	0	x		
5	4		x	
6	0	x		
7	0	x		
8	0		x	
9	5			x

Luego de implementar los nuevos tests, los valores de coverage alcanzados son:

- Instruction coverage: **83,30%** (de 2564, 2136 cubiertas).
- Branch Coverage: **56,60%** (de 106, 60 cubiertos).
- Line coverage: **79,69%** (de 655, 522 cubiertas).

Habiendo sumado al test suite:

- 3 tests en el paquete **States**.
- 14 tests en el paquete **Models**.
- 7 tests en el paquete **Controllers**.

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
▼ Mp3-Player-HelloWorld	90,6 %	4.179	434	4.613
▼ src	90,6 %	4.179	434	4.613
> main.java.controllers	83,5 %	81	16	97
> main.java.models	84,8 %	775	139	914
> main.java.states	92,9 %	262	20	282
> main.java.testDrives	0,0 %	0	20	20
> main.java.views	81,4 %	1.018	233	1.251
> test.java.controllers	98,5 %	192	3	195
> test.java.models	100,0 %	913	0	913
> test.java.states	100,0 %	938	0	938
> test.java.views	0,0 %	0	3	3

Figura 19 - Resultados obtenidos del Instruction Coverage en el Eclipse Photon con el plugin Java Code Coverage 3.1.1.

Element	Coverage	Covered Branches	Missed Branches	Total Branches
▼ Mp3-Player-HelloWorld	58,2 %	64	46	110
▼ src	58,2 %	64	46	110
> main.java.controllers	62,5 %	5	3	8
> main.java.models	72,6 %	45	17	62
> main.java.states	100,0 %	6	0	6
> main.java.testDrives		0	0	0
> main.java.views	13,3 %	4	26	30
> test.java.controllers		0	0	0
> test.java.models		0	0	0
> test.java.states	100,0 %	4	0	4
> test.java.views		0	0	0

Figura 20 - Resultados obtenidos del Branch Coverage en el Eclipse Photon con el plugin Java Code Coverage 3.1.1.

Element	Coverage	Covered Lines	Missed Lines	Total Lines
▼ Mp3-Player-HelloWorld	88,5 %	1.035	135	1.170
▼ src	88,5 %	1.035	135	1.170
> main.java.controllers	83,3 %	30	6	36
> main.java.models	81,0 %	209	49	258
> main.java.states	81,2 %	69	16	85
> main.java.testDrives	0,0 %	0	6	6
> main.java.views	79,3 %	214	56	270
> test.java.controllers	98,1 %	52	1	53
> test.java.models	100,0 %	212	0	212
> test.java.states	100,0 %	249	0	249
> test.java.views	0,0 %	0	1	1

Figura 21 - Resultados obtenidos del Line Coverage en el Eclipse Photon con el plugin Java Code Coverage 3.1.1.

Como puede verse, se encontraron 9 defectos al codificar las nuevas pruebas unitarias, siendo que se esperaba encontrar un total de 5, lo que refleja la inexperiencia del equipo a la hora del desarrollo.

5.6. ¿Qué porcentaje de “ branch coverage ” es deseable para un proyecto de software? Justifique.

Se investigó en la web, y según la documentación del software Bullseye Code Coverage Analyzer ([www.bullseye.com/minimum.html](http://www.bullseye.com/minimum.html)), un porcentaje aceptable para tests de sistema oscila entre un 70% y un 80%, mientras que para unit testing el porcentaje asciende hasta un 80 - 90% (hablando de coverage de instrucciones).

Está claro que estos valores dependen del proyecto, las convenciones tomadas al inicio del trabajo, la dificultad y magnitud del mismo, entre otros. Es necesario tener en cuenta también que a menudo lograr un 100% de coverage

puede ser muy bueno, pero su costo demasiado elevado, y no compensar con la cantidad de errores descubiertos.

En nuestro proyecto, consideramos como un buen valor alcanzar un 80% de instruction coverage y un 60% de branch coverage, considerando también las explicaciones mencionados en las consignas anteriores sobre archivos simples en cuanto al código, métodos simples, métodos estándar, etc.

Cabe destacar además que existen diferentes estándares, con requerimientos más o menos elevados en función de qué tan crítico es el software ejecutado. Algunos de ellos son:

- **DO-178B:** estándar de aviación. Estipula que si un fallo en el sistema causaría lesiones en un pasajero, la cobertura requerida es del 100% de las instrucciones, mientras que si la falla puede causar la muerte, el coverage impuesto es del 100% de las instrucciones, y el 100% de los branches.
- **Requerimientos del FDA para dispositivos médicos:** recomienda testing estructural, pero no especifica un valor de coverage requerido, sino que indica que el valor de coverage debe ser proporcional al nivel de riesgo que acarrea el uso del software en particular.

## 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.

Los tests de sistema definidos en el documento correspondiente, durante la materia Ingeniería de Software son:

- Reproducir canción.
- Pausar.
- Reanudar.
- Detener.
- Play luego de un stop.
- Siguiente canción.
- Borrar canción (ninguna se encuentra en reproducción).
- Ver información de la canción.
- Ver portada del álbum.
- Barra de volumen.
- Botón mute.
- Añadir un archivo con extensión distinta a mp3.

- Tratar de reproducir cuando no hay canciones.
- Borrar la canción que se está reproduciendo.

En total son 14 tests. Se considera que abarcan una gran parte de la funcionalidad esperada a nivel de sistema, sin embargo pueden agregarse los casos de test siguientes, para completar funciones no abarcadas:

#### **15. Descripción del Test: Canción anterior**

##### **Ejecución (pasos):**

1. Abrir el programa.
2. Apretar el botón para añadir canción(+).
3. Elegir un archivo con extensión .mp3.
4. Añadir otra canción.
5. Comenzar a reproducir la primer canción presionando Play.
6. Hacer click en el botón anterior (<<).

**Resultado esperado:** La última canción en la lista debería comenzar a reproducirse.

#### **16. Descripción del Test: Realizar acciones cuando no hay canciones en la playlist**

##### **Ejecución (pasos):**

1. Abrir el programa.
2. Presionar el botón Stop.
3. Presionar el botón Siguiente (>>).
4. Presionar el botón Anterior (<<).
5. Presionar el botón de borrar canción.
6. Presionar el botón de ver portada.
7. Presionar el botón de ver información de la canción.

**Resultado esperado:** No debería haber ningún cambio en la UI.

#### **17. Descripción del Test: Borrar otras canciones que no sea la que se encuentra en reproducción**

##### **Ejecución (pasos):**

1. Abrir el programa.
2. Apretar el botón para añadir canción(+).
3. Elegir un archivo con extensión .mp3.
4. Añadir otra canción.
5. Comenzar a reproducir la primer canción presionando Play.
6. Seleccionar la segunda canción con el mouse.
7. Borrar la segunda canción presionando el botón de eliminar.

**Resultado esperado:** La primer canción debería seguir sonando, y la segunda debería borrarse de la playlist.

6.2. ¿En ésta etapa, se encontró la cantidad de defectos esperada de acuerdo a su plan de remoción de defectos?

En esta etapa se encontraron 3 defectos (tests de sistema faltantes), de los 4 que deberían haberse hallado según el plan de remoción de defectos planteado.

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?

Con la ayuda de la planilla de Excel brindada en las clases de la materia, se obtuvo la siguiente tabla, considerando que en Ingeniería de Software se utilizó un plan Estándar sin las etapas de:

- Revisión de requerimientos.
- Inspección de diseño.
- Revisión de diseño.
- Inspección de código.
- Revisión de la documentación.

	Plan Ing. de SW.	Plan Integral	Ahorro
<b>Costo pre-release</b>	\$145.520	\$36.606	\$108.914
<b>Costo total</b>	\$351.495	\$112.285	\$239.210

**Con etapa de release:**

$\$351.495 - \$112.285 = \$239.210$

**Durante la fase de desarrollo:**

$\$145.520 - \$36.606 = \$108.914$

## 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.

El servidor de integración continua que se instaló fue Jenkins. El mismo se configuró para que realice un build con cada commit como solicitaba la consigna.

El servidor Jenkins funciona como servidor local, por lo que para poder realizar un build con cada commit se debió abrir el puerto utilizado (8080 en este caso) y utilizar una IP pública.

Los pasos para instalar Jenkins (en Linux) fueron los siguientes:

1. Ejecutar el comando: `docker pull jenkins/jenkins:lts`.
2. Ejecutar el comando: `docker run -p 8080:8080 -p 50000:50000 jenkins/jenkins:lts`.
3. Abrir el navegador e introducir la url: *localhost:8080*.
4. Se requiere una clave, la cual se obtiene en la iniciación del container del docker.
5. Instalar plugins sugeridos.
6. Completar el formulario ingresando los datos de usuario admin, password, full name, email. Guardar y continuar.
7. Se abre un formulario en donde se solicita la jenkins URL. Introducir la dirección IP pública correspondiente de acuerdo a la configuración y apertura de puertos del router.
8. Click en FreeStyle Project. Asignar nombre del proyecto (en este caso “Hello World”) y click en OK.
9. En Source Code Management, click en Git. Poner URL del repo de git y la branch correspondiente (branch master en este caso). Para este paso, es necesario la instalación del Git Integration Plugin.
10. **Configuración de Build Triggers.** Marcar la opción *GitHub hook trigger for GitScm Polling*.
11. **Configurar Build.** Seleccionar la opción *Invoke Gradle script*. Marcar las casillas *Use Gradle Wrapper* y *Make gradlew executable*. En el text box con etiqueta *Tasks* escribir:

*check*

*test*

*build*



12. **Acciones para ejecutar después.** Introducir email en la sección *Notificación por correo.*
13. Click en Apply.
14. Click en Save.

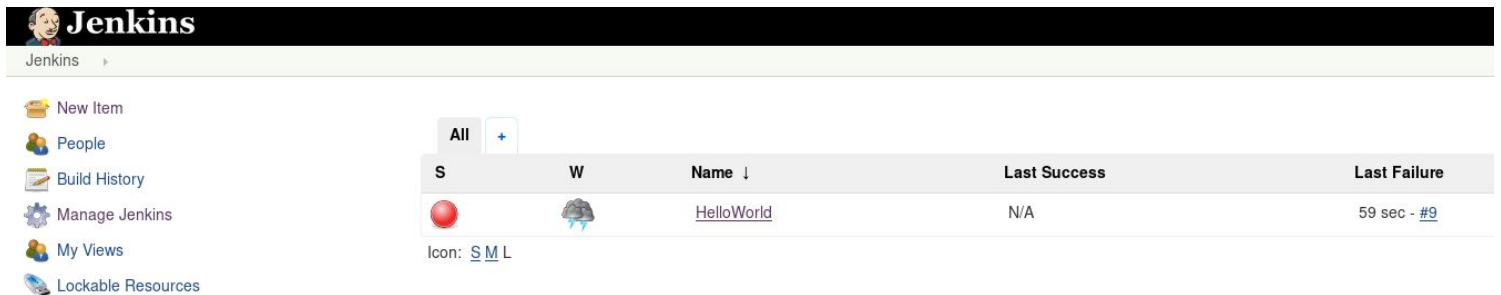


Figura 22 - Proyecto HelloWorld conectado al repositorio de Github en Jenkins.

Para que nuestro servidor de integración continua pueda reconocer el evento push, se debió también configurar el apartado “Webhooks” en las configuraciones del repositorio de GitHub, especificando la dirección del servidor de integración continua.

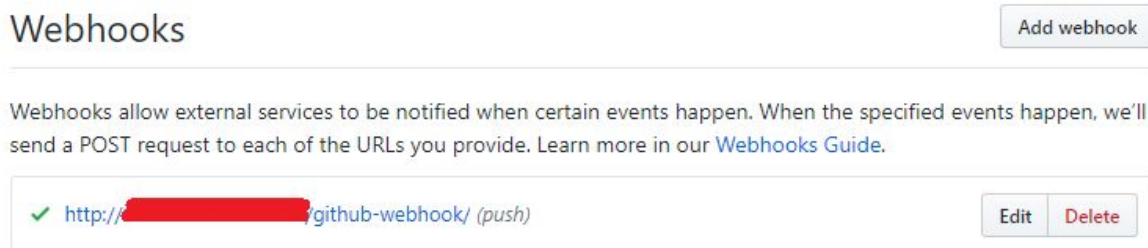


Figura 23 - Configuración del repositorio de GitHub para notificar al servidor Jenkins que se efectuó un PUSH.

Caso en el que el build no falla:

The screenshot shows the Jenkins web interface for a build named 'HelloWorld' (build #57). The left sidebar contains navigation links: 'Volver al proyecto', 'Estatus', 'Cambios', 'Console Output' (selected), 'Editar información de la ejecución', 'Borrar Proyecto', 'Git Build Data', 'No Tags', 'Checkstyle Warnings', 'Static Analysis Warnings', and 'Ejecución previa'. Below these is a list of 'Executed Gradle Tasks' including `compileJava`, `processResources`, `classes`, `checkstyleMain`, `compileTestJava`, `processTestResources`, `testClasses`, `checkstyleTest`, `findbugsMain`, `findbugsTest`, `pmdMain`, `pmdTest`, `test`, and `check`.

The main console output area, titled 'Salida de consola', shows the following log:

```

Started by user matias kleiner
Building in workspace /var/jenkins_home/workspace/HelloWorld
> git rev-parse --is-inside-work-tree # timeout=10
Fetching changes from the remote Git repository
> git config remote.origin.url https://github.com/CondorNegro/TPFinalGestionCalidadSU.git # timeout=10
Fetching upstream changes from https://github.com/CondorNegro/TPFinalGestionCalidadSU.git
> git --version # timeout=10
> git fetch --tags --progress https://github.com/CondorNegro/TPFinalGestionCalidadSU.git +refs/heads/*:refs/remotes/origin/* # timeout=10
> git rev-parse refs/remotes/origin/master^(commit) # timeout=10
> git rev-parse refs/remotes/origin/master^(commit) # timeout=10
Checking out Revision 47527b24e82961e6520b604f158ce4f5b34384be (refs/remotes/origin/master)
> git config core.sparsecheckout # timeout=10
> git checkout -f 47527b24e82961e6520b604f158ce4f5b34384be
Commit message: "prueba test"
> git rev-list --no-walk 47527b24e82961e6520b604f158ce4f5b34384be # timeout=10
[HelloWorld] $ /var/jenkins_home/workspace/HelloWorld/gradlew check test build findbugsMain
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:checkstyleMain UP-TO-DATE
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:checkstyleTest UP-TO-DATE
:findbugsMain UP-TO-DATE
:findbugsTest UP-TO-DATE
:pmdMain UP-TO-DATE
:pmdTest UP-TO-DATE
:test UP-TO-DATE
:check UP-TO-DATE

```

Figura 24 - Build exitoso en Jenkins.

## Compilar #57 (22-oct-2018 0:33:13)

The screenshot shows the Jenkins build results for 'Compilar #57' (22-oct-2018 0:33:13). The results are listed as follows:


- No changes.**
- Iniciado por el usuario [matias kleiner](#)**
- Revision: 47527b24e82961e6520b604f158ce4f5b34384be**
  - `refs/remotes/origin/master`
- Checkstyle: [3.795 warnings](#) from one analysis.**
  - [3.795 new warnings](#)
  - Plug-in Result:  - no threshold has been exceeded
  - New highscore: only successful builds since yesterday!
- FindBugs: 0 warnings.**
  - No warnings since build 57.
  - New zero warnings highscore: no warnings since yesterday!
  - During parsing an [error](#) has been reported.
- PMD: 0 warnings.**
  - No warnings since build 57.
  - New zero warnings highscore: no warnings since yesterday!
  - During parsing an [error](#) has been reported.
- Duplicate Code: 0 warnings.**
  - No warnings since build 57.
  - New zero warnings highscore: no warnings since yesterday!
  - During parsing an [error](#) has been reported.
- Static Analysis Warnings: [3.795 warnings](#).**
  - [3.795 new warnings](#)


Figura 25 - Resultados obtenidos luego del build exitoso en Jenkins.

Para que el build no falle, se debieron sacar aquellos tests que intentan instanciar una GUI en Docker (tests del controlador).

Cada “build” debe incluir las siguientes etapas:

a. Compilación del código fuente. Haga que el “ build ” falle cuando existan errores de compilación.

La configuración fue efectuada en el punto anterior. A continuación se demostrará que la consigna se cumple de una manera simple, por ejemplo, sacándole un punto y coma a alguna línea.



```
MP3View.java
93     ImageIcon nextIcon = new ImageIcon(getClass().getClassLoader().getResource(nextIconPath));
94     String addIconPath = "main/resources/images/add.png";
95     ImageIcon addIcon = new ImageIcon(getClass().getClassLoader().getResource(addIconPath));
96     String stopIconPath = "main/resources/images/stop.png";
97     ImageIcon stopIcon = new ImageIcon(getClass().getClassLoader().getResource(stopIconPath));
98     String deleteIconPath = "main/resources/images/delete.png";
99     ImageIcon deleteIcon = new ImageIcon(getClass().getClassLoader().getResource(deleteIconPath));
100    String infoIconPath = "main/resources/images/songinfo.png"
101    ImageIcon infoIcon = new ImageIcon(getClass().getClassLoader().getResource(infoIconPath));
102    String artIconPath = "main/resources/images/songart.png";
103    ImageIcon artIcon = new ImageIcon(getClass().getClassLoader().getResource(artIconPath));
```

Figura 26 - Introducción de error de sintaxis en línea 100.

The screenshot shows the Jenkins web interface for a project named 'HelloWorld'. The left sidebar contains navigation links: 'Volver al proyecto', 'Estatus', 'Cambios', 'Console Output', 'View as plain text', 'Editar información de la ejecución', 'Borrar Proyecto', 'Logs de polling', 'Git Build Data', 'No Tags', 'Checkstyle Warnings', and 'Ejecución previa'. The main area is titled 'Salida de consola' and displays the following console output:

```

Started by GitHub push by CondorNegro
Building in workspace /var/jenkins_home/workspace/HelloWorld
> git rev-parse --is-inside-work-tree # timeout=10
Fetching changes from the remote Git repository
> git config remote.origin.url https://github.com/CondorNegro/TPfinalGestionCalidadSW.git # timeout=10
Fetching upstream changes from https://github.com/CondorNegro/TPfinalGestionCalidadSW.git
> git --version # timeout=10
> git fetch --tags --progress https://github.com/CondorNegro/TPfinalGestionCalidadSW.git +refs/heads/*:refs/remotes/origin/*
> git rev-parse refs/remotes/origin/master^{commit} # timeout=10
> git rev-parse refs/remotes/origin/origin/master^{commit} # timeout=10
Checking out Revision f0ea20833baa5b8639413d99871533092e4b0f59 (refs/remotes/origin/master)
> git config core.sparsecheckout # timeout=10
> git checkout -f f0ea20833baa5b8639413d99871533092e4b0f59
Commit message: "Saco punto y coma para probar"
> git rev-list --no-walk 1cc8b04195324ec31202945ac21e5cfc3467a426 # timeout=10
[Gradle] - Launching build.
[HelloWorld] $ /var/jenkins_home/workspace/HelloWorld/gradlew check test build
:compileJava/var/jenkins_home/workspace/HelloWorld/src/main/java/views/MP3View.java:100: error: ';' expected
    String infoIconPath = "main/resources/images/songinfo.png"
                                ^
1 error
FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':compileJava'.
> Compilation failed; see the compiler error output for details.

* Try:
Run with --stacktrace option to get the stack trace. Run with --info or --debug option to get more log output.

BUILD FAILED

```

Below the console output, there is a section titled 'Executed Gradle Tasks' which lists the task `compileJava/var/jenkins_home/workspace/HelloWorld/src/main/java/views/MP3View.java:100`.

Figura 27 - Detección de error de sintaxis en línea 100 en Jenkins.

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.

Como primer paso, se debió instalar el plugin de CheckStyle en Jenkins. CheckStyle es una herramienta de análisis de código estático que verifica si un código fuente escrito en Java cumple con ciertas reglas de codificación. En Jenkins dicho plugin posee varias configuraciones:







Acciones para ejecutar después.

☒ 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" value="0"/>	<input type="text" value="0"/>	<input type="text" value="0"/>	<input type="text" value="0"/>
	<input type="text" value="0"/>	<input type="text" value="0"/>	<input type="text" value="0"/>	<input type="text" value="0"/>

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 ☒

If set the number of new warnings will always be computed based on the previous build, even if that build is unstable (due to a violated warning threshold). Otherwise the last build that did not violate any given threshold will be used as reference. It is recommended to uncheck this option if the plug-in should ensure that all new warnings will be finally fixed in subsequent builds.

Only use stable builds as reference ☐

Use the last stable build as the reference to compute the number of new warnings against. This allows to ignore interim unstable builds for which the number of warnings decreased. Note that the last stable build is evaluated only by inspecting the unit test failures. The static analysis results are not considered.

Figura 29 - Configuración del plugin de CheckStyle.

Para cumplir con la consigna de forma de hacer que el “ build ” falle cuando la cantidad de errores supere la cantidad que había en el “ build ” anterior, se debió tildar la opción *Compute new warnings (based on the last successful build unless another reference build is chosen below)* para que Jenkins se fije al hacer el build en la aparición de nuevos warnings con respecto a los que existían en un build previo de referencia. Al tildar dicha opción se despliega una serie de opciones. Una de ellas hay que tildarlas y es la que dice *Use previous build as reference*. Dicha opción coloca al build anterior (sin importar si fue un build exitoso o no) como referencia. Otra de las que se debe tildar es la que posee como etiqueta *Use delta for new warnings*. Dicha opción setea como número de new warnings la diferencia entre la cantidad de estos en el build actual con respecto a los del build previo. Luego, se debe colocar un umbral para determinar que el build falle cuando se supera ese umbral de nuevos warnings. Es por ello que se debe colocar en los recuadros de la sección *Status Thresholds* el valor de cero. Esto indica que el build va a fallar en caso de que la diferencia de warnings entre el build actual y el anterior sea igual a 1 o mayor. Por último, se debe hacer click en Apply y Guardar.

A continuación se mostrará que lo explicado anteriormente funciona. Uno de los errores que detecta CheckStyle, son los de indentación.



Checkstyle: [3,794 warnings](#) from one analysis.

- Plug-in Result: - no threshold has been exceeded (Reference build: [#70](#))
- Still one day before reaching the previous successful builds highscore.

Figura 30 - Resultados de CheckStyle previo a cambio de código fuente.

```
build.gradle  MP3View.java x
367
368  @Override
369  public void updatePlaylistInfo() {
370      String[] playlist = model.getCurrentPlaylist();
371      songList.clear(); //Borro la songList que habia y le pido al modelo que me de la nuev
372      for(int i=0;i<playlist.length;i++){
373          songList.addElement(playlist[i]);
374      }
375      jSongList.setSelectedIndex(model.getIndex());
376  }
377
378  @Override
379  public void updateTrackProgress(int progress, int size) {
380      if (progress==0){
381          progressBar.reset()
382          ;           You, a few seconds ago • Uncommitted changes
383      }
384      else {
385          progressBar.increase();
386      }
387      progressBar.setMax(size);
388      int minutes = progress/60;
389      int seconds = progress%60;
390      String a = String.format("%02d:%02d", minutes, seconds);
391      lblst.setText(a);
392  }
393  }
394
```

Figura 31 - Se agrega error de indentación en línea 381 al colocar el punto y coma en la línea 382.



## Compilar #76 (22-oct-2018 1:39:17)



### Changes

1. view para probar check ([detail](#) / [githubweb](#))



Started by [GitHub push](#) by [CondorNegro](#)



Revision: 06e1fb769132b909c3c642009a042e64d21765c8

- refs/remotes/origin/master



Checkstyle: [3,795 warnings](#) from one analysis.


- [18 new warnings](#)
- [17 fixed warnings](#)
- Plug-in Result:  - [1 new warning](#) exceeds the threshold of 0 by 1 (Reference build: [#75](#))

Figura 32 - Fallo de build por poseer 1 warning más que build anterior.

También es importante destacar la configuración que se colocó en el archivo build.gradle del proyecto.

```
checkstyle {
    ignoreFailures = false
    toolVersion = '6.6'
    configFile file("config/checkstyle/checkstyle.xml")
}

tasks.withType(Checkstyle) {
    reports {
        xml.enabled = true
        html.enabled = true
    }
}
```

Figura 33 - Archivo *build.gradle*.

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.

En primer lugar, se instaló el plugin de PMD en Jenkins. Los pasos de configuración de dicho plugin en Jenkins son los mismos que los de CheckStyle. Básicamente, se trata de un analizador de código estático que informa de problemas que detecta en el código fuente. Algunos de ellos son variables sin uso, objetos creados en vano, etc.

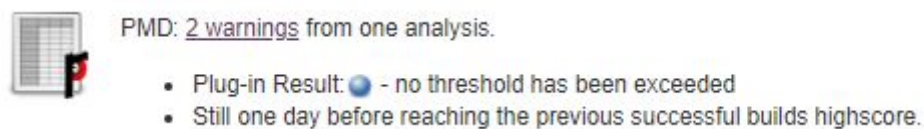


Figura 34 - Resultados de PMD previo a cambio de código fuente.

```
378  @Override
379  public void updateTrackProgress(int progress, int size) {
380      int variableSinUsarFindBugs;;
381      if (progress==0){
382          progressBar.reset();
383      }
384      else {
385          progressBar.increase();
386      }
387      progressBar.setMax(size);
388      int minutes = progress/60;
389      int seconds = progress%60;
390      String a = String.format("%02d:%02d", minutes, seconds);
391      lblst.setText(a);
392  }
393 }
```

Figura 35 - Se agrega error con doble coma en línea 380.

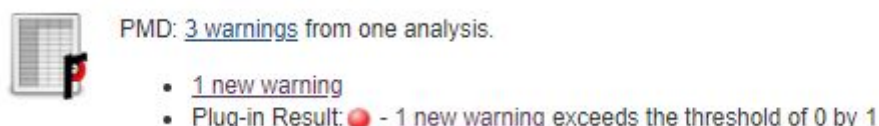


Figura 36 - Fallo de build por poseer un warning más que build anterior.

```

task pmd (type: Pmd) {

    ignoreFailures = false
    description 'Run pmd'
    group 'verification'
    rulePriority = 2

    ruleSets = ["java-android",
               "java-basic",
               "java-braces",
               "java-strings",
               "java-design",
               "java-unusedcode"]
    source = fileTree('src/main/java')

    reports {
        xml.enabled = true
        html.enabled = false
    }
}

check.doLast {
    project.tasks.getByName("pmd").execute()
}

```

Figura 37 - Configuración de archivo build.gradle del proyecto.

Link consultado para la configuración del archivo build.gradle:

<http://qaru.site/questions/2410758/jenkins-pmd-plugin-how-to-show-report-on-build-failure>

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.

Utilizando el plugin de PMD, se prosiguió a configurar la herramienta CPD, la cual permite encontrar duplicaciones de código. Se configuró el archivo build.gradle como indica el link que se muestra a continuación:

<https://gist.github.com/mychaelstyle/9826322>

```

check << {
    File outDir = new File('build/reports/pmd/')
    outDir.mkdirs()
    ant.taskdef(name: 'cpd', classname: 'net.sourceforge.pmd.cpd.CPDTask',
        classpath: configurations.pmd.asPath)
    ant.cpd(minimumTokenCount: '100', format: 'xml',
        outputFile: new File(outDir, 'cpd.xml')) {
        fileset(dir: "src/main/java") {**/
        include(name: '**/*.java')
        }
    }
}

```

Figura 38 - Configuración de archivo build.gradle del proyecto.

Se probó efectuar duplicaciones de líneas de código del proyecto pero las detectaba como errores de compilación de Java. Como el PMD no lanzaba warnings en base a código duplicado, se nos complicó poder finalizar este punto, ya que no encontramos casos de falla del CPD.

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.

En primer lugar, se instaló el plugin de FindBugs en Jenkins, el cual posee los mismos pasos de configuración que el de CheckStyle. Esta herramienta permite encontrar errores en códigos escritos en Java. Es de análisis estático también. Para probar el funcionamiento de lo solicitado en la consigna se efectúa lo expresado en las siguientes imágenes:



FindBugs: [37 warnings](#) from one analysis.

- Plug-in Result: ● - no threshold has been exceeded
- Still one day before reaching the previous successful builds highscore.

Figura 39 - Resultados de FindBugs previo a cambio de código fuente.

```
dependencies {
    testCompile 'junit:junit:4.10'
    compile fileTree(dir: 'lib', include: '*.jar')
    findbugs 'com.google.code.findbugs:findbugs:3.0.1'
    findbugs configurations.findbugsPlugins.dependencies
    findbugsPlugins 'com.h3xstream.findseccbugs:findseccbugs-plugin:1.4.4'
}
```

Figura 40 - Configuración de dependencias del archivo build.gradle.

```
tasks.withType(FindBugs) {
    reports {
        xml.enabled = true
        html.enabled = false
    }
}

task findSecurityBugs(type: FindBugs) {
    classes = fileTree(project.rootDir.absolutePath).include("**/*.class");
    source = fileTree(project.rootDir.absolutePath).include("**/*.java");
    classpath = files()
    pluginClasspath = project.configurations.findbugsPlugins
    findbugs {
        toolVersion = "3.0.1"
        //xml.enabled = true
        //html.enabled = false
        ignoreFailures = false
        effort = "max"
        reportLevel = "low"
    }
}
```

Figura 41 - Configuración del archivo build.gradle.

Links solicitados para la configuración del archivo build.gradle:

<https://github.com/finnishtransportagency/lippu-api-example/blob/master/build.gradle>  
<https://github.com/find-sec-bugs/find-sec-bugs/wiki/Gradle-configuration>

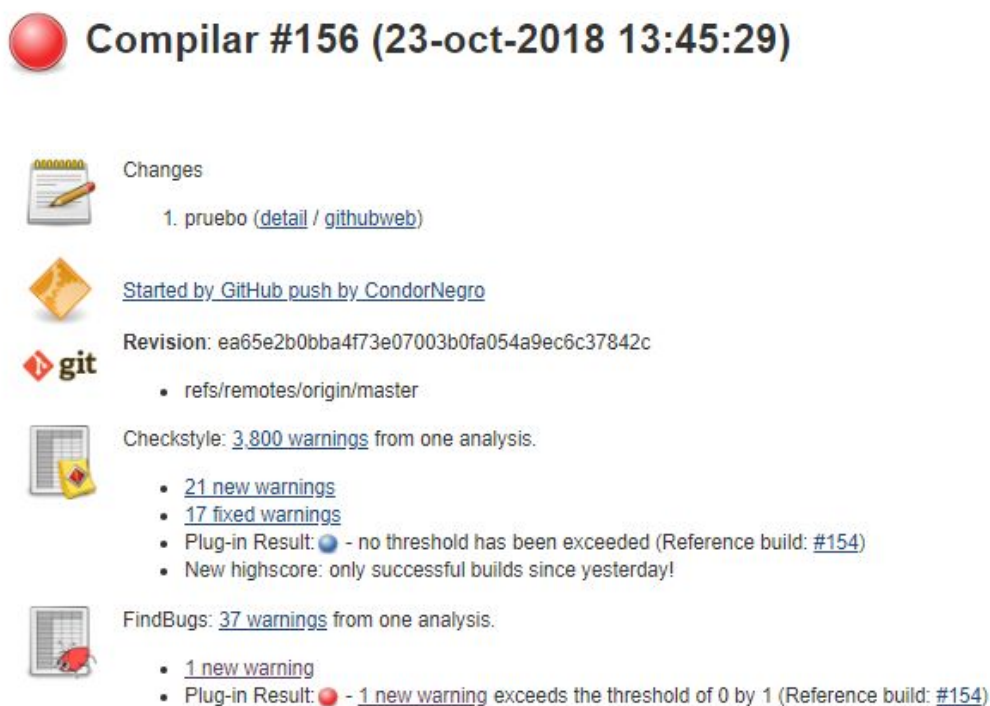


```


280 @Override
281 public byte[] getAlbumArt(){
282     Mp3File song = null;
283     try {
284         song = new Mp3File(playlist.get(index));
285     } catch (UnsupportedTagException | InvalidDataException | IOException e) {
286         e.printStackTrace();
287     }
288     if (song.hasId3v2Tag()) {
289         byte[] albumArt = song.getId3v2Tag().getAlbumImage();
290         return albumArt;
291     }
292     byte[] error = "Error".getBytes();
293     return error;
294 }

```


Figura 42 - Se agrega error en línea 292 para que lo detecte FindBugs.




**Compilar #156 (23-oct-2018 13:45:29)**


 Changes


- 1. pruebo ([detail](#) / [githubweb](#))


 Started by [GitHub](#) push by [CondorNegro](#)

 Revision: ea65e2b0bba4f73e07003b0fa054a9ec6c37842c

- refs/remotes/origin/master

 Checkstyle: [3,800 warnings](#) from one analysis.

- [21 new warnings](#)
- [17 fixed warnings](#)
- Plug-in Result:  - no threshold has been exceeded (Reference build: [#154](#))
- New highscore: only successful builds since yesterday!

 FindBugs: [37 warnings](#) from one analysis.


- [1 new warning](#)
- Plug-in Result:  - [1 new warning](#) exceeds the threshold of 0 by 1 (Reference build: [#154](#))

Figura 43 - Fallo de build por poseer un warning más que build anterior.


El error agregado para que lo detecte FindBugs consiste en que la conversión de String a bytes con el método `getBytes()` disminuye la portabilidad del proyecto.

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.

Para cumplir con la consigna, en primer lugar, se debió instalar el plugin de JaCoCo en Jenkins. A su vez, se debió instalar la versión 1.0.18 por problemas de compatibilidad que generaba con Jenkins. Para ello, se descargó el plugin desde el siguiente link:

<http://updates.jenkins-ci.org/download/plugins/jacoco/>

Una vez descargado, desde la configuración avanzada de plugins en Jenkins, se instaló manualmente la versión deseada.



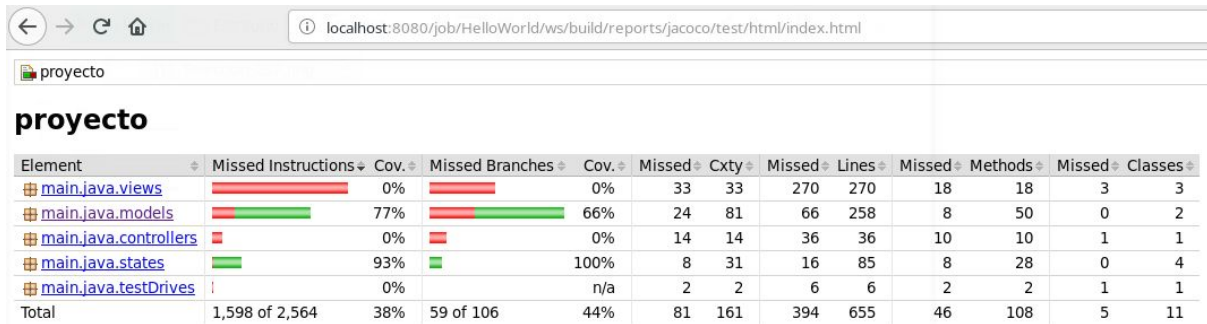
The screenshot shows the Jenkins configuration page for installing plugins manually. It is divided into two main sections: 'Upload Plugin' and 'Update Site'. The 'Upload Plugin' section has a 'Submit' button at the top left, followed by the title 'Upload Plugin'. Below the title is a description: 'You can upload a .hpi file to install a plugin from outside the central plugin repository.' There is a 'File:' label, a 'Browse...' button, and the text 'No file selected.' Below this is an 'Upload' button. The 'Update Site' section has the title 'Update Site', followed by a 'URL' label and a text input field containing 'https://updates.jenkins.io/update-center.json'. At the bottom of this section is a 'Submit' button.

Figura 44 - Instalación del plugin manual.

Con el plugin ya instalado y configurado el build.gradle tal como indica la Figura 49, los resultados fueron los siguientes:



Dentro del directorio *build/reports/jacoco/test/html* podemos encontrar el reporte *index.html* que genera la herramienta, donde se observa en la Figura 45 un build exitoso con la mayor cobertura conseguida mediante los test realizados.



The screenshot shows a web browser displaying the JaCoCo HTML report for a project named 'proyecto'. The report is titled 'proyecto' and shows a table of coverage data for various elements. The table includes columns for Element, Missed Instructions, Cov., Missed Branches, Cov., Missed Cxty, Missed Lines, Missed Methods, and Missed Classes. The data is as follows:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods	Missed Classes
main.java.views	33	0%	33	0%	270	270	18	3
main.java.models	24	77%	81	66%	66	258	8	2
main.java.controllers	14	0%	14	0%	36	36	10	1
main.java.states	8	93%	31	100%	16	85	8	4
main.java.testDrives	2	0%	2	n/a	6	6	2	1
Total	1,598 of 2,564	38%	59 of 106	44%	81	161	394	655

Figura 45 - Mediciones de JaCoCo para un build exitoso.

El plugin permite una configuración de forma tal de indicar cuando la cobertura es satisfactoria y cuando no, lanzando un build exitoso o un failure en cada caso. Para ello, se lo configuró tal como se observa en la Figura 46. Cabe destacar que la versión del plugin no permite la comparación del build actual con el anterior.



The screenshot shows the Jenkins plugin configuration for coverage thresholds. The table has columns for Instruction, % Branch, % Complexity, % Line, % Method, and % Class. The values are as follows:

	Instruction	% Branch	% Complexity	% Line	% Method	% Class
☀	60	60	60	60	60	60
☁	43	43	43	43	43	43

Figura 46 - Umbrales configurados en el plugin de Jenkins.

Lo que se hizo a continuación para probar los umbrales que se configuraron anteriormente fue comentar varios test que se realizaron sobre el unit test *MP3ModelTest.java*, con lo que se redujo la cobertura y se espera (debido al umbral configurado) que el build falle. Tal como indica la Figura 47, el build falla debido a la cobertura que alcanzó.

```
[JaCoCo plugin] Collecting JaCoCo coverage data...
[JaCoCo plugin] **/*.exec;/**/*.classes;/**/*.src/main/java; locations are configured
[JaCoCo plugin] Number of found exec files for pattern **/*.exec: 1
[JaCoCo plugin] Saving matched execfiles: /var/jenkins_home/workspace/HelloWorld/build/jacoco/test.exec
[JaCoCo plugin] Saving matched class directories for class-pattern: **/*.classes: /var/jenkins_home/workspace/HelloWorld/build/classes /var/jenkins_home/workspace/HelloWorld/build/reports
/testes/classes
[JaCoCo plugin] Saving matched source directories for source-pattern: **/*.src/main/java: /var/jenkins_home/workspace/HelloWorld/src/main/java
[JaCoCo plugin] Loading inclusions files..
[JaCoCo plugin] inclusions: []
[JaCoCo plugin] exclusions: []
[JaCoCo plugin] Thresholds: JacocoHealthReportThresholds [minClass=40, maxClass=40, minMethod=30, maxMethod=30, minLine=40, maxLine=40, minBranch=40, maxBranch=40, minInstruction=100,
maxInstruction=100, minComplexity=80, maxComplexity=80]
[JaCoCo plugin] Publishing the results..
[JaCoCo plugin] Loading packages..
[JaCoCo plugin] Done.
Build step 'Record JaCoCo coverage report' changed build result to FAILURE
Finished: FAILURE
```

Figura 47 - Failure de build por no superar los umbrales.

Por último, en la Figura 48 se observan los nuevos valores de cobertura alcanzados para contrastar con los alcanzados antes de comentar el unit test *MP3ModelTest.java*.

proyecto												
proyecto												
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods	Missed Classes				
main.java.views	<div><div></div></div>	0%	<div><div></div></div>	0%	33 33	270 270	18 18	3 3				
main.java.models	<div><div></div></div>	59%	<div><div></div></div>	52%	36 81	105 258	13 50	0 2				
main.java.controllers	<div><div></div></div>	0%	<div><div></div></div>	0%	14 14	36 36	10 10	1 1				
main.java.states	<div><div></div></div>	93%	<div><div></div></div>	100%	8 31	16 85	8 28	0 4				
main.java.testDrives	<div><div></div></div>	0%		n/a	2 2	6 6	2 2	1 1				
Total	1,767 of 2,566	31%	68 of 106	36%	93 161	433 655	51 108	5 11				

Figura 48 - Mediciones de JaCoCo para un build que falla.

```
apply plugin: "jacoco"

jacoco {
    toolVersion = "0.7.4+"
}

jacocoTestReport {
    reports {
        xml.enabled true
        html.enabled true
    }
}

check.dependsOn jacocoTestReport
```

Figura 49 - Configuración de archivo build.gradle.

Link consultado para la configuración del archivo build.gradle:

[https://docs.gradle.org/current/userguide/jacoco\\_plugin.html](https://docs.gradle.org/current/userguide/jacoco_plugin.html)

Nota: debido a que la versión de Jenkins presenta incompatibilidad con el plugin downgradeado que se instaló, los resultados no son mostrados directamente en la página principal, sino que tuvimos que acceder al directorio *build/reports/jacoco/test/html* para poder analizarlos.

## Conclusión.

Como conclusión de este proyecto se puede mencionar que se logró comprender la utilidad de las herramientas de integración continua y de las revisiones e inspecciones en cada etapa del desarrollo. Además, gracias a los conocimientos teóricos y prácticos adquiridos en la materia sobre herramientas, técnicas y metodologías que se utilizan para asegurar la calidad de un producto de software, logramos encontrar una cantidad de defectos en un proyecto de Ingeniería de Software similar a lo estimado. Por último, se logró calcular y estimar el gran ahorro en costos que implica la aplicación de las técnicas que se llevan a cabo a lo largo de este trabajo.

## Anexo.

En el presente trabajo se utilizó un repositorio de GitHub aparte al usado en Ingeniería de Software para efectuar el proyecto final.

Link del repositorio de Gestión de la Calidad de Software:

<https://github.com/CondorNegro/TPfinalGestionCalidadSW>

Usuarios de GitHub:

- Kleiner Matías: ragnar-l.
- López Gastón: CondorNegro.
- Maero Facundo: facundojmaero.
- Rivero Franco: Franco Rivero.

Link del repositorio de Ingeniería de Software:

<https://github.com/castagno/IngSoft-2016-HelloWorld>

Por último es importante mencionar que se utilizó Trello como una herramienta para listar y notificar los avances de las tareas en las que fue descompuesta la realización de este trabajo final de la materia Gestión de la Calidad de Software.