Caso de estudio deuda técnica en proyectos de software (mayo 2022)

Sergio Andrés Rodríguez Torres, estudiante de magíster

Escuela Colombiana de Ingeniería Julio Garavito

***Abstract*— Al desarrollar proyectos de software complejos y de larga duración nos afrontamos a la complejidad de mantener y expandir el proyecto, para lograrlo la calidad del software se vuelve un factor crucial. En este artículo se presenta un caso de estudio en el cual se hace el análisis desde distintos puntos de vista sobre la deuda técnica y calidad de software de un proyecto de software, con el fin de presentar un ejemplo sobre un proyecto sencillo para ver los diferentes conceptos y componentes que se pueden tener en cuenta al revisar la deuda técnica de un proyecto.**

***Index Terms*— deuda técnica, calidad de software, CI, ATAM.**

I. Introducción

Este documento busca presentar un caso de estudio sobre el análisis de deuda técnica en proyectos de software en 6 etapas.

En la primera se realizará una revisión preliminar intentando detectar de forma manual *code smells* o problemas se pueden detectar sin seguir un método específico, luego revisaremos de forma detallada el código fuente validando si cumple con las características *clean code*, algunos principios de programación y por último las practicas XP. Luego revisaremos cómo están las pruebas del código basándonos en prácticas para identificar la deuda en las pruebas, analizaremos el uso de pruebas unitarias le daremos algunas sugerencias. Después se hará una revisión sobre herramientas para el análisis automático de calidad de código e integración continua, luego se analizará la deuda técnica que haya en el diseño a nivel de arquitectura del proyecto y por último se complementará con análisis corto ATAM para entender y priorizar los atributos de calidad en el proyecto.

Uglytrivia es el proyecto corto consiste en un juego de trivial que se puede jugar de 2 a 6 jugadores, es una Kata para reforzar técnicas de corrección de errores o *bugs* y refactorizar código, para ello en el proyecto se dejaron múltiples errores y debilidades de diseño de forma intencional. [1]

El objetivo es identificar y explicar cuál es el problema, como decisiones de diseño contribuyen o facilitan el problema, proponer mejores soluciones y argumentar porque son mejor. El análisis se realizó sobre la versión en Kotlin del proyecto.

Fuentes del proyecto y análisis en: https://github.com/SergioRt1/BugsZero-Kata

II. Revisión preliminar

Cuando pensamos en realizar una revisión sobre la calidad del software, la primera idea es revisar el proyecto el código fuente e intentar detectar algún defecto o algo extraño en el código fuente, pero terminamos en una búsqueda poco estructurada y sin objetivos claros por lo que puede que pasemos muchas cosas por alto. Aun así, me es interesante realizar el ejercicio para luego contrastar los resultados con las otras estrategias.

Código duplicado: En varios métodos se repite código como:

```
if (currentPlayer == players.size) currentPlayer = 0
```

Código difícil de probar: Es difícil de aislar las pruebas validando una sola funcionalidad, la UI y la lógica están mezcladas, y hay métodos que realizan más de una funcionalidad y llaman a otros métodos privados que son difíciles de probar de forma individual, como roll llama *movePlayerAndAskQuestion* que a su vez llama a *askQuestion* que llama a *currentCategory*.

Acceso a índices de arreglos sin validar se accede a índices con valores fijos sin validar el tamaño del arreglo, generando posibles *Index out of bounds exception*.

```
fun MutableList<String>.removeFirst(): String {
  return this.removeAt(0)
}
```

Variables definidas como mutables cuando pueden ser inmutables *val* y definirse en el constructor. El constructor no se usa para inicializar los parámetros de la clase, se usan los valores default y se dejan literales mágicos en el código.

```
var players = mutableListOf<String>()
var places = IntArray(6)
var purses = IntArray(6)
var inPenaltyBox = BooleanArray(6)
```

Acoplamiento de la cantidad de jugadores: Del bloque de

código anterior se ve que el número máximo de jugadores está explícito en el código (6) como un literal sin valor semántico, estas variables se pueden inicializar en el constructor para darle más flexibilidad. Hay poca cohesión entre la lógica y los datos.

Posibles técnicas de refactorización identificadas:

- Extract Method
- Extract variable
- Encapsulate Record
- Change Function Declaration
- Inline Function
- Replace Magic Literal
- Remove Flag Argument

III. Análisis del código

A. *Características Clean code*

Código enfocado: No hay métodos excesivamente largos, sin embargo, se puede mejorar hay métodos tiene varias responsabilidades, por ejemplo, *movePlayerAndAskQuestion* hace dos cosas mueve al jugador y realiza una pregunta, esto se puede separar.

```
private fun movePlayerAndAskQuestion(roll: Int) {
  places[currentPlayer] = places[currentPlayer] + roll
  if (places[currentPlayer] > 11)
     places[currentPlayer] = places[currentPlayer] - 12
  println(players.get(currentPlayer)
          + "'s new location is "
          + places[currentPlayer])
  println("The category is " + currentCategory())
  askQuestion()
}
```

Regla del Boy Scout: Esta característica aplica en el historial de Git se observan cambios al código fuente y la documentación, correcciones de bugs y actualización de librerías ejemplo: commit *9dd55ac2*

Entendible: Esta característica se cumple en su mayoría, los nombres de variables y métodos tiene un sentido semántico, pero hay algunas literales mágicos como:

```
private fun didPlayerWin(): Boolean {
  return purses[currentPlayer] != 6
}
```

Escalable: La aplicación no es muy escalable, por ejemplo, agregar la funcionalidad para jugar de a más de dos jugadores requiere realizar múltiples cambios porque parte de la lógica está acoplada a 6 jugadores

```
var inPenaltyBox = BooleanArray(6)
```

Duplicidad: Hay segmentos de código que se repiten en múltiples métodos como:

```
  if (currentPlayer == players.size) currentPlayer = 0
```

Abstracción: Se puede mejorar la abstracción hay métodos que hacen más de una cosa y la lógica y la paca de vista que interactúa con el usuario están muy acopladas, si se desea cambiar la UI no se puede reusar la logia del juego.

```
private fun didPlayerWin(): Boolean {
  fun add(playerName: String): Boolean {
  players.add(playerName)
  places[howManyPlayers()] = 0
  purses[howManyPlayers()] = 0
  inPenaltyBox[howManyPlayers()] = false

  println(playerName + " was added")
  println("They are player number " + players.size)
  return true
}
```

Testeable: No es sencillo de testear, al menos con pruebas unitarias aisladas que validen cada método.

Principio menor asombro: Este principio se cumple los métodos hacen lo que su nombre índica.

B. *Principios de programación*

YAGNI: Este principio se cumple, no hay funcionalidades o métodos sin usar.

KISS: El proyecto es corto y por ende tiene poca complejidad, sin embargo, hay métodos innecesarios como:

```
fun MutableList<String>.addLast(element: String) {
  this.add(element)
}
```

DRY: Hay segmentos de código repetidos en varias partes del código como:

```
  if (currentPlayer == players.size) currentPlayer = 0
```

SOLID: Se pueden aplicar varias mejoras siguiendo los principios SOLID, como ya se ha mencionado anteriormente, por ejemplo, refactorizar lo acoplado que se encuentra que el juego solo sea para dos jugadores o como la UI y la lógica están juntas permitiría mejorar el principio Open/Close.

C. *Practicas XP*

Refactoring: Se pueden aplicar técnicas de refactoring para extraer variables, métodos, darle significado semántico a algunos literales mágicos.

```
private fun currentCategory(): String {
  if (places[currentPlayer] == 0) return "Pop"
  if (places[currentPlayer] == 4) return "Pop"
  if (places[currentPlayer] == 8) return "Pop"
  if (places[currentPlayer] == 1) return "Science"
  if (places[currentPlayer] == 5) return "Science"
  if (places[currentPlayer] == 9) return "Science"
  if (places[currentPlayer] == 2) return "Sports"
  if (places[currentPlayer] == 6) return "Sports"
  return if (places[currentPlayer] == 10) "Sports"
  else "Rock"
}
```

Además, la lógica y capa de la vista que interactúa con el usuario están muy acopladas.

Continuous integration: El proyecto no cuenta con CI.

Simple design: Se puede refactorizar la aplicación, en un diseño más sencillo menos acoplado y extensible.

Test-driven development: Se puede aplicar esta práctica para construir una aplicación fácil de testear y menos acoplada.

IV. Deuda técnica en las pruebas

A. *Prácticas para deuda en las pruebas*

No hay pruebas unitarias, hay una sola prueba en el proyecto y válida toda la funcionalidad en diversos casos, se podría considerar pruebas de usuario final. Las pruebas están muy acopladas al estado actual del proyecto, si se agregan nuevas funcionalidades se debería cambiar el archivo *GameTest.itsLockedDown.approved.txt* que contiene todos los escenarios que son válidos.

Cubrimiento de las pruebas: Las pruebas cubren casi todo el código (96% de las líneas), el único método que no se prueba es:

```
val isPlayable: Boolean
    get() = howManyPlayers() >= 2
```

porque es un método que no se usa, aun así, se podría realizar una prueba unitaria para validar su funcionamiento.

No se usa el patrón AAA en las pruebas, no se cumple Divide y vencerás, las pruebas se componen de múltiples escenarios que se corren en una única prueba.

Se usan estándares de nombramiento y prácticas *de Clean Code*, el nombre de la única prueba es *itsLockedDown* el cual no es muy semántico de lo que se desea validar.

Nuevos defectos nuevas pruebas unitarias: Como está si se detecta un defecto se modificaría el archivo *GameTest.itsLockedDown.approved.txt* que contiene los resultados esperados, no se realizaría una nueva prueba unitaria.

B. *Pruebas unitarias*

Se Agregaron nuevas pruebas unitarias para la clase *Game* en el archivo *GameTest.kt* que se encuentra en el mismo paquete de la clase aprobar y se renombró la prueba original a *GameRunnerTest*. Para garantizar que no haya dependencias entre pruebas se agregaron bloques *@Before* y *@After* para limpiar la ejecución de cada caso.

```
private lateinit var out: PrintStream

@Before
fun setUp() {
    out = System.out
}

@After
fun tearDown() {
    System.setOut(out)
}
```

Cada prueba tiene el buffer de salida default, por lo que, si se valida el output sobrescribiendo el buffer no va a afectar a otras pruebas, esto es útil para pruebas como *roll message should work* y *add player message should work* entre otras que validan el output de la consola.

C. *Sugerencias para las pruebas*

Se debería refactorizar el código para reducir las dependencias entre métodos, por ejemplo, hay métodos que realizan múltiples acciones y llaman a otros métodos privados que son difíciles de probar de forma individual, como *roll* llama *movePlayerAndAskQuestion* que a su vez llama a *askQuestion* que llama a *currentCategory*.

Se debería separar la capa de UI y de lógica, se deben hacer pruebas que cubran ambas funcionalidades sobre el mismo método, por ejemplo, las pruebas *add player message should work* y *add player should work*, este acoplamiento dificulta las pruebas y la extensibilidad de la aplicación.

V. CI y herramientas de análisis de calidad

A. *SonarCloud*

Para configurar *SonarCloud* debemos conceder permisos para acceder al repositorio Git, luego establecer en el repositorio un *secret* en la configuración de *GitHub Actions* con el token de Sonar, por último, debemos agregar el archivo de *build.yml* para el pipeline en *Github Actions* y agregar las properties de sonar a la de Maven en el *pom.xml*. Estos cambios se pueden ver en el *commit* 3aa238a.

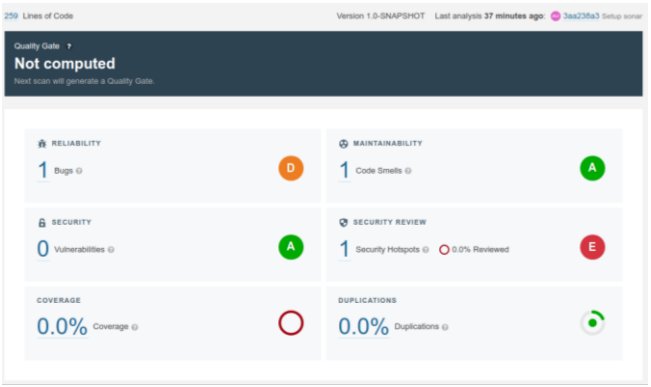

Figura 1 Dashboard de resumen del proyecto en SonarCloud

B. *Cobertura con JaCoCo*

La cobertura de código es el porcentaje de código que está cubierto por pruebas automatizadas. La medición de cobertura de código determina declaraciones en el código se han ejecutado a través de la ejecución de pruebas y qué declaraciones no.

SonarCloud nos permite llevar un historial de las métricas de calidad del código de nuestro proyecto, pero no se encarga de hacer el análisis de cobertura, usa el informe generado por herramientas externas para seguir la cobertura.

En este caso se usará JaCoCo, una de las herramientas más populares para realizar el análisis de cobertura en la JVM, el proceso para configurar la herramienta se puede encontrar en la documentación, modificamos el pom.xml para incluir el plugin

de JaCoCo, estos cambios lo podemos ver en el *commit* 47ab1d8.

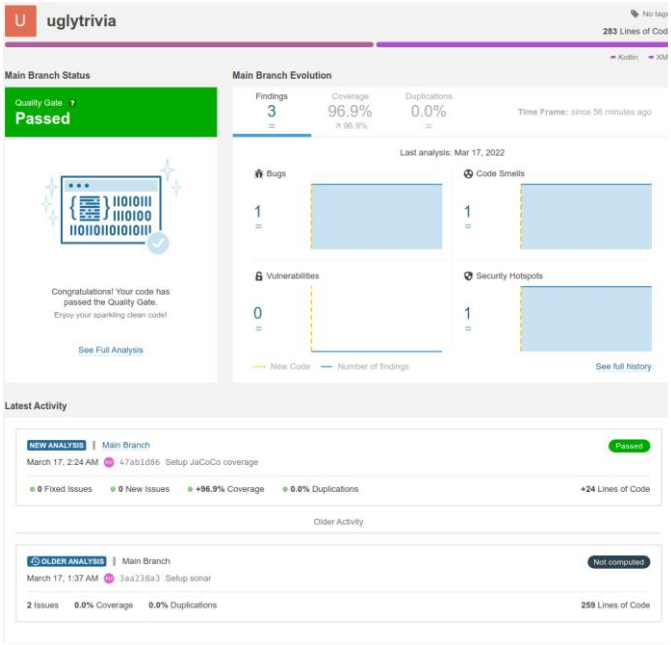

Figura 2 Dashboard de SonarCloud del resultado de la ejecución de las pruebas con la cobertura del código.

C. *SonarLint*

SonarLint es una herramienta que permite realizar análisis de código de forma local en el IDE para identificar y corregir problemas de calidad y seguridad en el código de forma temprana. Para instalar la herramienta solo debemos ir al IDE y agregar el plugin, en la nueva pestaña de SonarLint podemos correr un análisis sobre el proyecto para identificar los problemas de forma local, estos errores son los mismos que vemos en SonarCloud.

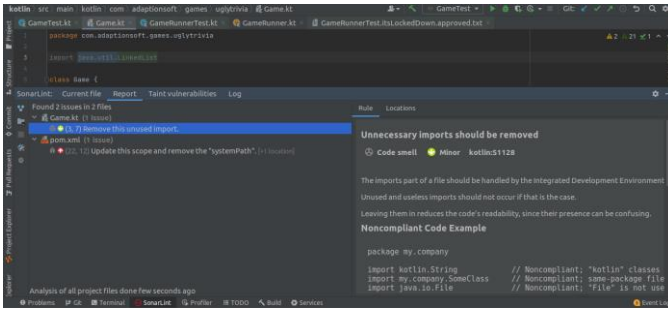

Figura 3 Análisis realizado sobre el proyecto con la herramienta Sonarlint.

D. *Pasos personalizados en Github workflow*

Podemos separar en diferentes pasos el *build*, *tests*, el análisis estático de código y agregar pasos extra cómo enviar notificaciones a servicios externos con el resultado de la ejecución del *workflow*, para eso debemos modificar el archivo *build.yml* en .github/workflows que especifica el funcionamiento de nuestro workflow, podemos crear varios Jobs cada uno para cada una de las etapas del flujo.

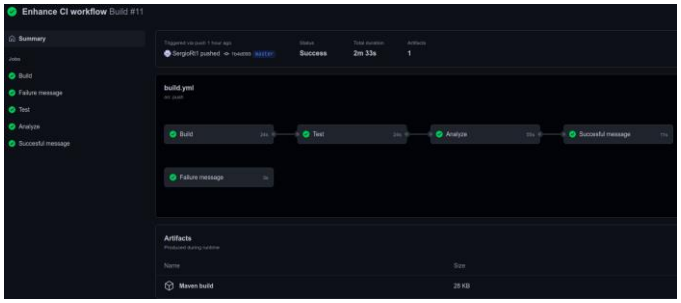

Figura 4 Flujo personalizado del proceso de integración continua con Github actions

El mensaje resultado puede ser exitoso o fallido, crearemos un Job para cada caso, para el caso exitoso lo dejaremos sin condición, pero la acción tiene como prerrequisito del ejecutar los pasos anteriores de forma satisfactoria, el caso fallido tendrá otro Job que siempre se ejecuta, pero este está condicionado a solo enviar el mensaje si el *workflow* falla.

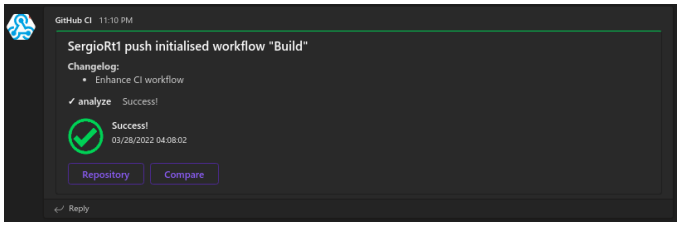

Figura 5 Ejemplo mensaje producto de un build existo enviado a Microsoft Teams.

E. *Análisis sobre el proyecto*

Con las herramientas de análisis de código podemos detectar algunos problemas y llevar trazabilidad de la evolución del proyecto, la cobertura de las pruebas (97%~) es amplia debido a las pruebas unitarios agregados, además el proyecto no es muy extenso y no hay demasiados problemas que sean evidentes.

VI. Deuda técnica en la arquitectura

Para identificar las causas de los problemas de arquitectura que pueda haber en el proyecto es necesario preguntarnos ¿Quién es el equipo en involucrado? En este caso el proyecto es un Dojo de aprendizaje con algunos errores hechos a propósito, por lo que podemos suponer que la mayoría de la deuda en la arquitectura es intencional.

El proyecto en sí tiene una funcionalidad sencilla por lo que no hay muchos drivers y restricciones de negocio, por lo cual vamos a centrarlos en los problemas de arquitectura que se puedan identificar analizando el código.

Dependencias externas sin soporte: Haciendo uso de herramientas de análisis de código como *Sonarlint* configurado previamente, podemos identificar *Architectural Smells* en el archivo pom.xml, con una de las dependencias usadas en el proyecto *ApprovalTests*, la cual no se importa de Maven central u otro repositorio unificado dependencias, sino que el archivo Jar se encuentra directamente en el proyecto y esto compromete la portabilidad del proyecto y es una dependencia inestable. [2]

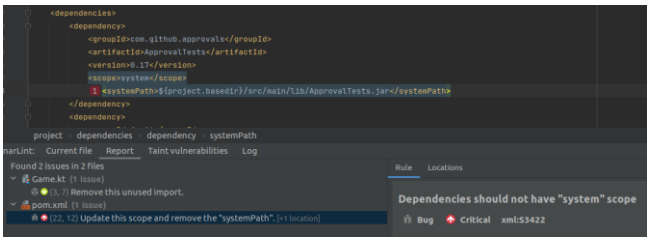

Figura 6 Detección de problema con la importación de la dependencia ApprovalTests usando Sonarlint.

Obsolescencia tecnológica: Uno de los cambios necesarios para poder correr el proyecto actualizar la versión de Kotlin, ya que versiones menores a la 1.3 no son compatibles con el JDK 11, ni la última versión del plugin de compatibilidad Kotlin-JVM. La mayoría de las librerías que se encontraban originalmente en el proyecto están desactualizadas, por ejemplo, *JUnit 4.8.2*, tiene fecha lanzamiento octubre del 2010 y presenta algunas vulnerabilidades de seguridad CVE-2020-15250. [3]

Otro caso es *ApprovalTests* de la cual no se sabe exactamente que versión se usa en el *pom.xml* indica la versión 0.17, en Maven central la versión actual es la 15.0.0 la más antigua es la 2.0.0 de 2018, que cuenta con múltiples vulnerabilidades de seguridad como: CVE-2022-23305, CVE-2022-23302 entre otras. [4]

Problemas explícitos en el código: En el compone de código que se encarga de la lógica del proyecto también podemos encontrar deuda técnica en arquitectura, algunos de estos puntos se explican previamente en más detalle en secciones anteriores, pero por ejemplo el uso de constantes mágicas y los problemas de acoplamiento de la cantidad de jugadores en el juego, afectan el atributo de calidad de modificabilidad del proyecto.

El proyecto no cuenta con capas que podrían separar la vista (UI) y la lógica, están integrados los *prints* en los métodos del funcionamiento del juego *No layer*. [5]

Diseños no actualizados o no existentes: No existe diseños ni documentación de la arquitectura del proyecto, para trabajar de forma efectiva como un equipo es necesario ser capaz de expresar el funcionamiento de la arquitectura de forma rápida entre el equipo, en esto juega un rol importante los diagramas y otras formas de documentación, que nos permiten visualizar y documentar los artefactos y sus interacciones en un sistema de software, facilitando así la comunicación de ideas.

El proyecto en sí tiene una funcionalidad sencilla por lo que no hay muchos drivers y restricciones de negocio, por lo cual vamos a centrarlos en los problemas de arquitectura que se puedan identificar analizando el código.

VII. ATAM y QAW

Al momento de plantarnos una arquitectura compleja es necesario analizar los atributos de calidad del sistema y priorizarlos, debido a que nos encontraremos con escenarios en los que debemos realizar concesiones entre estos atributos para priorizar algunos sobre otros.

A continuación, plantearemos dos escenarios qué buscan poner a prueba nuestras decisiones de arquitectura para validar si se cumple la prioridad que le dimos a los atributos de calidad y plantearnos posibles mejoras o modificaciones a la arquitectura qué nos permitan satisfacer los escenarios planteados.

A. Refinamiento de escenarios

Debido al reducido el tamaño del proyecto sobre el cual se está realizando el análisis, se plantearon los escenarios pensando en cuáles son los atributos de calidad qué son se pueden ver afectados por el artefacto del que disponemos (el código fuente) y serian relevantes para el owner del producto.

Scenario Refinement for Scenario 1	
Scenario(s):	Cuando se decide agregar la capacidad al juego para poder jugar con un número arbitrario de jugadores
Business Goals:	Nueva característica
Relevant Quality Attributes:	Modificabilidad
Stimulus:	Se determinó que los usuarios del producto desean a jugar con más de 6 jugadores
Stimulus Source:	Feedback provisto por los usuarios a través de encuestas, valoraciones en línea o sugerencias
Environment:	El producto es exitoso y se detecta la necesidad de nuevas características
Artifact:	Código fuente
Response:	Es posible jugar con cualquier número de jugadores
Response Measure:	Tiempo qué se demora en salir el feature a producción y esfuerzo medido en puntos de historia para realizar la tarea
Questions:	Cuánto esfuerzo nos costaría desarrollar esta característica
Issues:	Puede ser necesario refactorizar y realizar modificaciones en el

Scenario Refinement for Scenario 1	
	código para poder tener esta característica

Este primer escenario planteó porque se detectó en el código que había cierto acoplamiento que impedía la modificabilidad de la cantidad de jugadores que pueden jugar el juego.

Scenario Refinement for Scenario 2	
Scenario(s):	Se desea expandir el mercado del juego haciéndolo disponible en múltiples plataformas con diferentes sistemas operativos
Business Goals:	Expansión del producto
Relevant Quality Attributes:	Portabilidad
Stimulus:	El producto es exitoso y se desea expandir a nuevas plataformas
Stimulus Source:	Estadísticas de consumo del producto
Environment:	El producto está rindiendo satisfactoriamente en la plataforma actual
Artifact:	Código fuente
Response:	El producto funciona sin problemas en todas las plataformas deseadas
Response Measure:	Número de plataformas en las que función correctamente
Questions:	Cuál sería el número mínimo adecuado de plataformas en las que debería funcionar el producto y cuáles serían las plataformas a las que se desea expandir el producto
Issues:	Durante el desarrollo del componente de software es necesario garantizar la portabilidad del sistema

El segundo escenario se planteó porque identificó que se importa una librería de pruebas (*ApprovalTests*) de forma inadecuada lo cual afecta significativamente la portabilidad del sistema.

VIII. Referencias

[1] M. Son, «Github,» [En línea]. Available: https://github.com/martinsson/BugsZero-Kata.

[2] T. Sharma, «tusharma.in,» [En línea]. Available: https://www.tusharma.in/smells/AUD.html.

[3] «Maven repository,» [En línea]. Available: https://mvnrepository.com/artifact/junit/junit/4.8.2.

[4] «Maven repository,» [En línea]. Available: https://mvnrepository.com/artifact/com.approvaltests/approvaltests/2.0.0.

[5] T. Sharma, «tusharma.in,» [En línea]. Available: https://www.tusharma.in/smells/NL.html.