TESIS UNA - PUNO

Universidad Nacional del Altiplano

UNIVERSIDAD NACIONAL DEL ALTIPLANO

FACULTAD DE INGENIERÍA MECÁNICA ELÉCTRICA, ELECTRÓNICA Y SISTEMAS

ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS

"USO DE LOS PRINCIPIOS DRY Y S.O.L.I.D. EN UN PROCESO DE REFACTORING DE UNA APLICACIÓN WEB JAVA PARA MEJORAR SU CALIDAD INTERNA"

TESIS

PRESENTADA POR:

WILLINGTON SUCASACA SURCO

PARA OPTAR EL TÍTULO PROFESIONAL DE:

INGENIERO DE SISTEMAS

PUNO – PERÚ

2018

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

TESIS UNA - PUNO

Universidad Nacional del Altiplano

UNIVERSIDAD NACIONAL DEL ALTIPLANO

FACULTAD DE INGENIERÍA MECÁNICA ELÉCTRICA, ELECTRÓNICA Y SISTEMAS

ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS

“USO DE LOS PRINCIPIOS DRY Y S.O.L.I.D. EN UN PROCESO DE REFACTORING DE UNA APLICACIÓN WEB JAVA PARA MEJORAR SU CALIDAD INTERNA”

TESIS PRESENTADA POR:

WILLINGTON SUCASACA SURCO

PARA OPTAR EL TÍTULO PROFESIONAL DE:

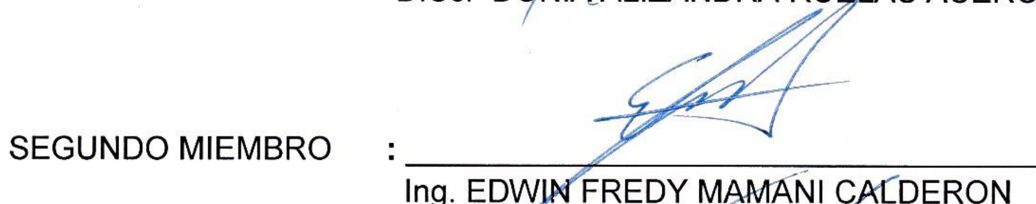
INGENIERO DE SISTEMAS

FECHA DE SUSTENTACIÓN: 20/11/2018

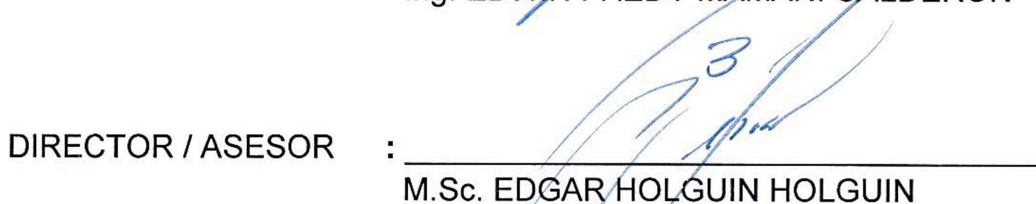
APROBADA POR EL JURADO REVISOR CONFORMADO POR:

PRESIDENTE : ______________________
M.Sc. HUGO YOSEF GÓMEZ QUISPE

PRIMER MIEMBRO : ______________________
D.Sc. DONIA ALIZANDRA RUELAS ACERO



SEGUNDO MIEMBRO : ______________________
Ing. EDWIN FREDY MAMANI CALDERON



DIRECTOR / ASESOR : ______________________
M.Sc. EDGAR HOLGUIN HOLGUIN

Área : Ingeniería de Software.
Tema : Calidad de Software.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

TESIS UNA - PUNO

Universidad Nacional del Altiplano

DEDICATORIA

A mis queridos padres

Sr. Sabino Sucasaca y Sra. Juana Surco,

y a mi familia.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

TESIS UNA - PUNO

Universidad Nacional del Altiplano

AGRADECIMIENTOS

Al M.Sc. Edgar Holguin Holguin, por su orientación para el desarrollo del presente trabajo de investigación.

A los miembros del jurado: M.Sc. Hugo Yosef Gómez Quispe, D.Sc. Donia Alizandra Ruelas Acero e Ing. Edwin Fredy Mamani Calderon; por sus sugerencias y recomendaciones para mejorar la presente investigación.

A mi familia, por su gran e invalorable apoyo, compresión y paciencia.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

TESIS UNA - PUNO

Universidad Nacional del Altiplano

ÍNDICE GENERAL

ÍNDICE DE FIGURAS ... 8
ÍNDICE DE TABLAS ... 12
ÍNDICE DE ACRÓNIMOS ... 14
RESUMEN ... 15
ABSTRACT ... 16
CAPÍTULO I ... 17
1.1. Introducción ... 17
1.2. Planteamiento del problema ... 19
1.3. Formulación del problema ... 21
1.4. Justificación ... 21
1.5. Limitaciones y restricciones de la investigación ... 23
1.6. Objetivos de la investigación ... 24
1.6.1. Objetivo general ... 24
1.6.2. Objetivos específicos ... 24
1.7. Hipótesis general ... 24
CAPÍTULO II REVISIÓN DE LITERATURA ... 25
2.1. Antecedentes de la investigación ... 25
2.2. Marco teórico ... 27
2.2.1. Software y Aplicaciones Web ... 27
2.2.2. Calidad de Software ... 28
2.2.3. Principio DRY ... 35
2.2.4. Principios S.O.L.I.D. ... 35
2.2.5. Cambiando el Software ... 41
2.2.6. Refactoring ... 43
2.2.7. El ciclo del refactoring ... 45
2.2.8. Proceso de refactoring ... 46
2.2.9. Pruebas del Software ... 47
CAPÍTULO III MATERIALES Y MÉTODOS ... 51
3.1. Lugar de estudio ... 51
3.2. Población ... 51
3.3. Muestra ... 51
3.4. Método de investigación ... 51
3.5. Técnicas e instrumentos de recolección de datos ... 52
3.5.1. Técnicas e instrumentos de recolección de datos ... 52
3.5.2. Procedimientos de recolección de datos ... 52
3.5.3. Métodos de tratamiento de datos ... 52

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

3.6. Operacionalización de variables 53
3.7. Métodos empleados... 53
3.7.1. Evaluación de la calidad interna del caso de estudio 53
3.7.2. Definición del proceso de refactoring para mejorar la calidad interna de una Aplicación Web Java 54
3.7.3. Desarrollo del proceso de refactoring sobre el caso de estudio..................... 54
3.7.4. Evaluación de la medida en que el uso de los principios DRY y S.O.L.I.D. en el proceso de refactoring desarrollado afectó la calidad interna del caso de estudio….. 55
3.8. Materiales empleados.. 55
3.8.1. Eclipse IDE.. 55
3.8.2. Maven... 55
3.8.3. SonarQube 56
3.8.4. DBEaver 56
3.8.5. MySQL.. 56
3.8.6. Modelio... 57
3.8.7. ObjectAid UML.. 57
3.8.8. Bizagi Process Modeler.. 57
3.8.9. JavaServer Faces 58
3.8.10. Spring Framework.. 58
3.8.11. Java Persistence API (JPA) 58
3.8.12. Hibernate 59
3.8.13. JUnit Framework.. 59
3.8.14. Mockito Framework.. 60
CAPÍTULO IV RESULTADOS Y DISCUSIÓN 61
4.1. Evaluación de la calidad interna del caso de estudio 61
4.1.1. Descripción del caso de estudio 61
4.1.2. Análisis de la calidad interna del caso de estudio 66
4.2. Definición del proceso de refactoring para mejorar la calidad interna de una aplicación Web Java... 67
4.2.1. Fase 1: Analizar e identificar incidencias.. 68
4.2.2. Fase 2: Elaborar un plan de refactoring.. 68
4.2.3. Fase 3: Desarrollar el refactoring 69
4.2.4. Fase 4: Medir los resultados de la calidad interna.. 69
4.2.5. Fase 5: Evaluar la calidad interna final 69
4.3. Desarrollo del proceso de refactoring sobre el caso de estudio 70
4.3.1. Fase 1: Analizar e identificar incidencias.. 70
4.3.2. Fase 2: Elaborar un plan de refactoring.. 72

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

4.3.3. Fase 3: Desarrollar el refactoring ... 74
4.3.4. Fase 4: Medir los resultados de la calidad interna ... 114
4.3.5. Fase 5: Evaluar la calidad interna final ... 116
4.4. Evaluación de la medida en que el uso de los principios DRY y S.O.L.I.D. en el proceso de refactoring desarrollado afectó la calidad interna del caso de estudio ... 117
4.4.1. En cuanto al número de Bugs ... 119
4.4.2. En cuanto al número de Vulnerabilidades ... 119
4.4.3. En cuanto al número de Code Smells ... 120
4.4.4. En cuanto al Código Duplicado ... 122
4.4.5. En cuanto al Tamaño de la Aplicación ... 122
4.4.6. En cuanto a la Cobertura de Código ... 123
4.5. Prueba de hipótesis ... 124
CONCLUSIONES ... 127
RECOMENDACIONES ... 129
REFERENCIAS ... 130
ANEXOS ... 133
Anexo A. Análisis de aplicaciones Web Java ... 134
Anexo B. Archivo pom.xml del proyecto Maven del caso de estudio obtenido después del proceso de refactoring ... 135
Anexo C. Archivo de propiedades para el análisis con SonarQube ... 136
Anexo D. Pruebas unitarias y de integración en el caso de estudio ... 137
Anexo E. Análisis de clases Java en las iteraciones del proceso de refactoring ... 142
ANEXO F. Reportes de SonarQube del análisis efectuado al caso de estudio antes del proceso de refactoring ... 146
ANEXO G. Reportes de SonarQube del análisis efectuado al caso de estudio después del proceso de refactoring ... 152
ANEXO H. Evolución de la calidad interna del caso de estudio durante el desarrollo de su proceso de refactoring ... 158
ANEXO I. Perfil / Java de SonarQube usado para el análisis del caso de estudio ... 160

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

TESIS UNA - PUNO

Universidad Nacional del Altiplano

ÍNDICE DE FIGURAS

Figura N° 2.1: ISO/IEC 25000. 30

Figura N° 2.2: ISO/IEC 25010 Características de calidad.......... 32

Figura N° 2.3: Principios de diseño S.O.L.I.D. 36

Figura N° 2.4: Una clase con una única responsabilidad. 38

Figura N° 2.5: Ciclo del refactoring. 46

Figura N° 2.6: Proceso de refactorización resumido. 47

Figura N° 4.1: Diagrama de límite de sistema del caso de estudio. 62

Figura N° 4.2: Funcionalidades del caso de estudio. 63

Figura N° 4.3: Arquitectura física del caso de estudio. 64

Figura N° 4.4: Diagrama de paquetes del caso de estudio.......... 64

Figura N° 4.5: Proceso de refactoring definido. 68

Figura N° 4.6: Estructura inicial del proyecto del caso de estudio en Eclipse IDE.......... 70

Figura N° 4.7: Medición de la calidad interna inicial del caso de estudio, obtenida con el software SonarQube antes del proceso de refactoring. 71

Figura N° 4.8: Porcentaje inicial de incidencias del caso de estudio. 72

Figura N° 4.9: División del proyecto Web en múltiples módulos Maven.......... 78

Figura N° 4.10: Proyecto y módulos Maven después del refactoring en Eclipse IDE. 78

Figura N° 4.11: Arquitectura del caso de estudio refactorizado. 79

Figura N° 4.12: Componentes generados del caso de estudio después del refactoring. 79

Figura N° 4.13: Ejemplo de clases duplicadas en la capa de lógica de negocios antes del refactoring. 84

Figura N° 4.14: Ejemplo de herencia aplicado en el refactoring de la capa de lógica de negocios. 86

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

TESIS UNA - PUNO

Universidad Nacional del Altiplano

Figura N° 4.15: Uso del patrón de diseño builder en el refactoring de la capa de lógica de negocios. 88

Figura N° 4.16: Fragmento de diagrama de clases obtenido después del refactoring de la capa de lógica de negocios.......... 89

Figura N° 4.17: Ejemplo de interface y clase con varias responsabilidades antes de su refactoring en la capa de lógica de negocios. 92

Figura N° 4.18: Ejemplo de comportamiento duplicado en la capa de lógica de negocios antes de su refactoring. 93

Figura N° 4.19: Ejemplo de un método que infringe los principios SRP y OCP en la capa de lógica de negocios.......... 94

Figura N° 4.20: Ejemplo de método después de su refactoring en la capa de lógica de negocio.......... 95

Figura N° 4.21: Ejemplo de inyección de dependencias en la capa de persistencia....... 96

Figura N° 4.22: Ejemplo de interface que cumple con ISP en la capa de persistencia. . 97

Figura N° 4.23: Implementación del patrón de diseño cadena de responsabilidad en la capa de lógica de negocios.......... 97

Figura N° 4.24: Ejemplo de clase con código duplicado en la capa de persistencia. ... 100

Figura N° 4.25: Ejemplo de métodos con código duplicado que infringen el principio DRY en la capa de persistencia. 101

Figura N° 4.26: Ejemplo de interface y clase que cumplen con ISP en la capa de persistencia.......... 102

Figura N° 4.27: Ejemplo de clases que incumplen con SRP y OCP en la capa de persistencia.......... 103

Figura N° 4.28: Clases GenericDtos, obtenidos después del refactoring de la capa de persistencia.......... 104

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

TESIS UNA - PUNO

Universidad Nacional del Altiplano

Figura N° 4.29: Clases DTOs y Parámetros, obtenidos después del refactoring de la capa de persistencia. 105

Figura N° 4.30: Clases que corresponden a criterios para consultas SQL obtenidos después del refactoring de la capa de persistencia. 106

Figura N° 4.31: Ejemplo de una clase obtenida después del refactoring de la capa de persistencia. 106

Figura N° 4.32: Fragmento de código fuente obtenido después del refactoring de la capa de persistencia. 107

Figura N° 4.33: Fragmento de código que muestra el uso de la inyección de dependencia en la capa de presentación. 109

Figura N° 4.34: Clase que presenta diversas incidencias en la capa de presentación del caso de estudio antes de su refactoring. 111

Figura N° 4.35: Uso del patrón Strategy después del refactoring de la capa de presentación. 112

Figura N° 4.36: Fragmento de diseño de clases obtenido después del refactoring de la capa de presentación. 113

Figura N° 4.37: Implementación del patrón de diseño Strategy después del refactoring de la capa de presentación. 114

Figura N° 4.38: Medición de la calidad interna final del caso de estudio, obtenida con el software SonarQube después del proceso de refactoring. 115

Figura N° 4.39: Porcentaje final de incidencias del caso de estudio. 115

Figura N° 4.40: Comparación de la calidad interna inicial y final del caso de estudio.116

Figura N° 4.41: Comparación del porcentaje de código duplicado inicial y final del caso de estudio. 116

Figura N° 4.42: Porcentaje bugs eliminado después del proceso de refactoring.......... 119

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

TESIS UNA - PUNO

Universidad Nacional del Altiplano

Figura N° 4.43: Porcentaje de vulnerabilidades eliminado después del proceso de refactoring. 120

Figura N° 4.44: Porcentaje de code smells eliminado después del proceso de refactoring. 121

Figura N° 4.45: Porcentaje de código duplicado eliminado después del proceso de refactoring. 122

Figura N° 4.46: Comparación del tamaño antes y después del proceso de refactoring. 123

Figura N° 4.47: Comparación de la cobertura de código antes y después del proceso de refactoring. 124

Figura N° 4.48: Resultados de la prueba de U efectuado con SPSS. 125

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

TESIS UNA - PUNO

Universidad Nacional del Altiplano

ÍNDICE DE TABLAS

Tabla N° 1.1: Análisis inicial del código fuente del caso de estudio............................ 21

Tabla N° 2.1: Principios SOLID.. 36

Tabla N° 3.1: Operacionalización de variables. 53

Tabla N° 4.1: Agrupación de paquetes por funcionalidad del caso de estudio. 65

Tabla N° 4.2: Tamaño inicial del caso de estudio (Aplicación Web)............................ 66

Tabla N° 4.3: Análisis del código fuente del caso de estudio antes del proceso de refactoring. 67

Tabla N° 4.4: Resumen del análisis del caso de estudio.. 71

Tabla N° 4.5: Plan de las etapas para desarrollar el refactoring. 73

Tabla N° 4.6: Estructura de paquetes del caso de estudio después del proceso de refactoring. 77

Tabla N° 4.7: Análisis de la capa de lógica de negocios del caso de estudio antes de su refactoring (módulo 1). 81

Tabla N° 4.8: Análisis de la capa de lógica de negocios del caso de estudio antes de su refactoring (módulo 2). 90

Tabla N° 4.9: Análisis de la capa de persistencia del caso de estudio antes de su refactoring. 98

Tabla N° 4.10: Análisis de la capa de presentación del caso de estudio antes de su refactoring. 108

Tabla N° 4.11: Comparación de los atributos de calidad interna del caso de estudio antes y después del proceso de refactoring. 117

Tabla N° 4.12: Análisis de los paquetes Java después del proceso de refactoring del caso de estudio. 118

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

TESIS UNA - PUNO

Universidad Nacional del Altiplano

Tabla N° 4.13: Comparación del número de bugs antes y después del proceso de refactoring. ... 119

Tabla N° 4.14: Comparación del número de vulnerabilidades antes y después del proceso de refactoring. ... 120

Tabla N° 4.15: Comparación del número de code smells antes y después del proceso de refactoring. ... 121

Tabla N° 4.16: Comparación del porcentaje de código duplicado antes y después del proceso de refactoring. ... 122

Tabla N° 4.17: Comparación del tamaño antes y después del proceso de refactoring. 123

Tabla N° 4.18: Comparación de la cobertura de código antes y después del proceso de refactoring. ... 123

Tabla N° 4.19: Número de incidencias de los paquetes Java del caso de estudio antes y después del proceso de refactoring desarrollado. ... 124

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

TESIS UNA - PUNO

Universidad Nacional del Altiplano

ÍNDICE DE ACRÓNIMOS

BPMN : Business Process Model and Notation / Modelo y Notación de Procesos de Negocio.

DAO : Data Access Object / Objeto de Acceso a Datos.

DIP : Dependency-Inversion Principle / Principio de Inversión de Dependencia.

DRY : Don´t Repeat Yourself / No te repitas.

DTO : Data Transfer Object / Objeto de Transferencia de Datos.

IDE : Integrated Development Environment / Entorno de Desarrollo Integrado.

ISP : Interface Segregation Principle / Principio de Segregación de Interfaces.

JPA : Java Persistence API / API de Persistencia de Java.

JSF : Java Server Faces.

LSP : Liskov Substution Principle / Principio de Sustitución de Liskov.

OCP : Open/Closed Principle / Principio Abierto/Cerrado.

ORM : Object-Relational Mapping / Mapeo Objeto/Relacional.

POJO : Plain Old Java Object / Antiguo Objeto Plano de Java.

SOLID : Conjunto de Principios SRP, OCP, LSP, ISP y DIP.

SQL : Structured Query Language / Lenguaje Estructurado de Consultas.

SRP : Single-Responsibility Principle / Principio de Responsabilidad Única.

SUT : Subject Under Test / Sujeto Bajo Prueba.

TDD : Test-Driven Development / Desarrollo Guiado por Pruebas.

UML : Unified Modeling Language / Lenguaje Unificado de Modelado.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

TESIS UNA - PUNO

Universidad Nacional del Altiplano

RESUMEN

Las Aplicaciones Web Java, que se encuentran en producción en instituciones o empresas, presentan la necesidad de evolucionar con el paso del tiempo según las necesidades y exigencias del negocio; el problema surge cuando su código fuente o diseño es de baja calidad, lo que ocasiona que el trabajo de adaptación a nuevos requerimientos sea difícil y consuma un tiempo importante; toda vez que su calidad interna tiene un efecto profundo en su mantenibilidad. El objetivo de esta investigación fue mostrar el efecto que tendrá en la calidad interna de una aplicación Web Java el uso de los principios DRY y S.O.L.I.D. en un proceso de refactoring. La mantenibilidad, seguridad y fiabilidad fueron los atributos de calidad que se consideraron en esta investigación. El caso de estudio -Aplicación Web Java-, corresponde a una aplicación paralela de una institución pública ubicada en la ciudad de Puno-Perú. La evaluación de la calidad interna inicial y final del caso de estudio se realizó con la herramienta analítica SonarQube, a través del cual se obtuvieron mediciones respecto al número de vulnerabilidades, número de bugs, número de code smells, porcentaje de código duplicado, cobertura de código y número de líneas de código. El proceso de refactoring definido consistió en cinco etapas, y su desarrollo permitió mitigar un porcentaje notable de las anomalías halladas inicialmente en el caso de estudio. Al evaluar en qué medida el uso de los principios DRY y S.O.L.I.D. en el proceso de refactoring desarrollado afectó la calidad interna del caso de estudio, se concluyó que el adecuado cumplimiento de estos principios repercute positivamente en su calidad interna.

Palabras Clave: DRY, SOLID, Refactoring, Calidad de Software, Aplicación Web Java.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

ABSTRACT

Java Web Applications, that are in production in institutions or companies, present the need to evolve over time according to the needs and demands of the business; the problem arises when its source code or design is of low quality, which causes that the work of adaptation to new requirements is difficult and consumes an important time; Considering that its internal quality has a profound effect on its maintainability. The objective of this research was to show the effect that the use of DRY and S.O.L.I.D. principles will have on the Java Web application internal quality in a refactoring process. Maintainability, security and reliability were the attributes of quality that were considered in this investigation. The case study - Java Web Application - corresponds to a parallel application of a public institution located in the city of Puno-Perú. The evaluation of the initial and final internal quality of the case study was carried out with the analytical tool SonarQube, through which measurements were obtained regarding the number of vulnerabilities, number of bugs, number of code smells, percentage of duplicate code, code coverage and number of lines of code. The refactoring process defined consisted of five stages, and its development allowed to mitigate a significant percentage of the anomalies found initially in the case study. When assessing the extent to which the use of the principles DRY and S.O.L.I.D. in the developed refactoring process, it affected the internal quality of the case study, it was concluded that the adequate compliance with these principles has a positive effect on its internal quality.

Key Words: DRY, SOLID, Refactoring, Software Quality, Java Web Application.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

CAPÍTULO I

1.1. Introducción

Las empresas e instituciones con áreas de desarrollo de software a menudo se enfrentan a labores de mantenimiento y extensión de las aplicaciones que tienen en producción. Estas labores podrían tomar más tiempo del necesario si la calidad interna del producto software no es adecuada, es decir, que su código fuente, tenga anomalías como: code smells, bugs o vulnerabilidades. En el área de la Ingeniería de Software, se tienen principios, prácticas y patrones, que ayudan a conseguir un código limpio, adaptable y de fácil mantenimiento. Los principios DRY y S.O.L.I.D., permiten que el código fuente sea de fácil lectura, adaptable a los cambios y evolución posterior. Asimismo, la tarea de mejorar el diseño interno del software se logra mediante el refactoring, el cual es una técnica valiosa a la hora de mejorar la estructura interna de los sistemas software. La presente investigación, tiene como objetivo, evaluar el efecto que tendrá en la calidad interna de una aplicación Web Java el uso de los principios DRY y S.O.L.I.D. en un proceso de refactoring, para lo cual se utilizó el software SonarQube con la finalidad de evaluar la calidad interna de la aplicación Web Java seleccionada como caso de estudio, los atributos de calidad del producto software considerados son: mantenibilidad, seguridad y fiabilidad.

La presente investigación, se encuentra dividido en cuatro capítulos, los cuales son:

En el **Capítulo I,** se menciona la introducción, se detalla el planteamiento del problema y su formulación, la justificación, las limitaciones y restricciones de la investigación, los objetivos e hipótesis de la investigación.

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

En el **Capítulo II.** Revisión de la literatura: se desarrolla la revisión de literatura, donde se mencionan los antecedentes de la investigación y el marco teórico de los principales conceptos.

En el **Capítulo III.** Materiales y métodos: se presenta el lugar de estudio, población y muestra. Asimismo, se describe la metodología utilizada, las técnicas e instrumentos de recolección de datos, la operacionalización de variables, los métodos empleados, por último, se describen los materiales empleados en la investigación.

En el **Capítulo IV.** Resultados y discusión: se exponen los resultados de la investigación, donde se realiza la evaluación de la calidad interna del caso de estudio – aplicación Web Java-, se define el proceso de refactoring para mejorar la calidad interna de una aplicación Web Java, se presenta el desarrollo del proceso de refactoring sobre el caso de estudio y se realiza la evaluación de la medida en que el uso de los principios DRY y S.O.L.I.D. en el proceso de refactoring desarrollado afectó la calidad interna del caso de estudio, para lo cual, se comparó los resultados del análisis de la calidad interna del caso de estudio antes y después del proceso de refactoring desarrollado en base al número de bugs, número de vulnerabilidades, número de code smells, porcentaje de código duplicado, tamaño de la aplicación y porcentaje de cobertura del código fuente.

Finalmente, se presentan las conclusiones a las cuales se ha llegado con la investigación, las recomendaciones y los anexos.

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

1.2. Planteamiento del problema

Los problemas de calidad del software se descubrieron inicialmente en la década de 1960 con el desarrollo de los primeros grandes sistemas de software, y han continuado invadiendo la ingeniería de software a partir de esa década. El software entregado era lento y poco fiable, difícil de mantener y de reutilizar. El descontento con esta situación condujo a la adopción de técnicas formales de gestión de calidad del software, desarrolladas a partir de métodos usados en la industria manufacturera. Estas técnicas de gestión de calidad, en conjunto con nuevas tecnologías y mejores pruebas de software, llevaron a progresos significativos en el nivel general de calidad del software (Sommerville, 2011).

La calidad de un sistema, aplicación o producto sólo es tan buena como los requerimientos que describen el problema, el diseño que modela la solución, el código que conduce a un programa ejecutable y las pruebas que ejercitan el software para descubrir errores (Pressman, 2010). Pero ¿qué sucede con el diseño de una aplicación que fue evolucionando continuamente con la implementación de nuevos requisitos producto de los cambios constantes del negocio?, estos cambios conllevan muchas veces al deterioro del diseño inicial del software, que finalmente afecta la legibilidad del código y dificulta su mantenibilidad y extensibilidad, ocasionando que su calidad se deteriore.

La calidad del software es un factor importante que las organizaciones que poseen áreas de desarrollo de software enfrentan. Las aplicaciones que desarrollan y pasan a producción, son usadas por un periodo de tiempo prolongado (por lo menos es lo que se espera), y producto de los cambios constantes del negocio, estas aplicaciones tienen la necesidad de evolucionar adecuadamente con el paso del tiempo. El problema surge cuando su estructura interna es de baja calidad, ya sea porque su arquitectura no

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

es la adecuada, o no cumple con los principios de diseño orientado a objetos o porque su código fuente presenta un número elevado code smells[1], bugs[2] u otras anomalías. La consecuencia de tener un mal diseño o un código de baja calidad, sin pruebas de cobertura, sumado al cambio de desarrolladores, ocasionará que el trabajo de añadir nuevas funcionalidades al software sea difícil y consuma un tiempo considerable. El software debe funcionar como se espera, con un nivel adecuado de usabilidad, fiabilidad y seguridad; debe poder evolucionar según las necesidades del negocio, su estructura interna debe permitir su extensión por otros programadores, su código debe ser de fácil lectura y comprensión; por último, debe tener una adecuada cobertura de pruebas que garanticen su buen funcionamiento.

El caso de estudio elegido para esta investigación no escapa a los problemas mencionados. En dos años, esta aplicación Web fue modificada en diferentes oportunidades, generalmente con el fin de agregar nuevas funcionalidades; sin embargo, en cada actualización se pagó un costo considerable de tiempo por los problemas con su calidad interna. La Tabla N° 1.1, evidencia que la calidad interna del caso de estudio presenta problemas, en total se aprecian 918 incidencias, 14.5% de código duplicado y 0.0% de cobertura de código.

Adicionalmente, se efectuó el análisis de cuatro aplicaciones Web Java, que son usadas en diferentes instituciones o empresas de la región de Puno, los resultados obtenidos se muestran en la Tabla N° A.1 del Anexo A, la cual muestra que las aplicaciones analizadas, carecen de pruebas unitarias, presentan código duplicado, y

[1] Olores del código, son cualquier síntoma en el código de un programa que posiblemente indica un problema más profundo.
[2] Un problema que representa algo mal en el código (SonarQube, 2018).

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

tienen cantidades considerables de code smells, bugs y vulnerabilidades[3], por consiguiente, se aprecian problemas en su calidad interna.

Tabla N° 1.1: Análisis inicial del código fuente del caso de estudio.

Aplicación	Líneas de código	Número de bugs	Número de vulnerabilidades	Número de code smells	Cobertura de código	Código Duplicado
Aplicación Web Java (Caso de Estudio)	15,571	192	57	669	0.00%	14.50%

Elaboración: Propia.

1.3. Formulación del problema

¿Qué efecto tendrá en la calidad interna de una Aplicación Web Java el uso de los principios DRY y S.O.L.I.D. en un proceso de refactoring[4]?

1.4. Justificación

En entornos empresariales, es habitual que las aplicaciones en producción sean extendidas con el propósito de que se adapten a los cambios que sufre el negocio, lo que conlleva a efectuar actualizaciones que muchas veces deben realizarse en periodos de tiempo cortos. Uno de los problemas de tener una baja calidad interna, es precisamente el tiempo excesivo que lleva realizar las modificaciones del producto software; considerando que en muchos casos no es posible construir la aplicación entera por el tiempo que involucra. En estos escenarios es preferible llevar a cabo un proceso de refactorización en busca de mejorar su calidad interna, con el objetivo de prevenir

[3] Un problema relacionado con la seguridad que representa una puerta trasera para los atacantes (SonarQube, 2018).

[4] Proceso de cambiar un sistema software de tal manera que no se altere su comportamiento externo, pero se mejore su estructura interna (Fowler, 1999).

TESIS UNA - PUNO

Universidad Nacional del Altiplano

futuras modificaciones. El objetivo de tener un buen diseño de software es abarcar la fase de mantenimiento de una manera más legible y sencilla; así como, conseguir crear nuevas funcionalidades sin tener que modificar en gran medida código antiguo. Los principios de Ingeniería de Software proporcionan los conceptos base que ayudan a prevenir y/o mitigar las anomalías en el código o en el diseño del producto software; sin embargo, estos principios muchas veces no son ampliamente conocidos ni aplicados por los profesionales que desarrollan software.

Como señala (Pressman, 2010), hay una característica presente en el software heredado: mala calidad. Hay veces en las que los sistemas heredados tienen diseños que no son susceptibles de extenderse, código confuso, documentación mala o inexistente, casos y resultados de pruebas que nunca se archivaron, una historia de los cambios mal administrada… la lista es muy larga. A pesar de esto, dichos sistemas dan apoyo a las "funciones básicas del negocio y son indispensables para éste".

(Fowler, 1999), define la refactorización como un proceso de cambiar un sistema software de tal manera que no se altere su comportamiento externo, pero se mejore su estructura interna.

Por su lado (Feathers, 2004), explica cuatro razones principales para cambiar el software: 1) Adicionar una funcionalidad, 2) Corregir un error, 3) Mejorar el diseño y 4) Optimización del uso de los recursos. En el caso de la mejora del diseño, es un tipo diferente de cambio de software. Cuando queremos alterar la estructura del software para hacerlo más fácil de mantener, generalmente queremos mantener su comportamiento intacto también. Cuando eliminamos comportamiento en el proceso, obtenemos un error. Una de las principales razones por la que muchos programadores no intentan mejorar el diseño a menudo es porque es relativamente fácil perder el comportamiento o crear un mal comportamiento en el proceso de hacerlo.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Respecto a SOLID, es el acrónimo de un conjunto de prácticas que, cuando se implementan juntas, hacen que el código sea adaptable al cambio. Las practicas SOLID fueron introducidas por Bob Martin hace 15 años. Aun así, estas prácticas no son tan ampliamente conocidas (McLean Hall, 2014). Por su lado, el principio DRY[5] (no te repitas) indica que toda pieza de código no debería ser duplicada debido a que la duplicación incrementa la dificultad en los cambios y evolución posterior.

Respecto al caso de estudio, la Tabla N° 1.1, evidencia problemas en su calidad interna, lo que justifica que se efectué un proceso de refactoring sobre esta aplicación, esto sumado a la importancia que tiene conseguir una adecuada calidad de software.

Por los motivos mencionados, en el presente trabajo, se usó los principios DRY y S.O.L.I.D. con el fin de mejorar la calidad interna de una aplicación Web Java por medio de un proceso de refactoring, apoyado en el uso de herramientas de análisis de código fuente que permitan obtener mediciones de la calidad interna del producto software.

1.5. Limitaciones y restricciones de la investigación

El presente trabajo, buscó incrementar la calidad interna de una aplicación web Java, centrándose en los atributos de calidad del producto que están especificados en el estándar ISO/IEC 25000[6] conocida como SQuaRE (Software Product Quality Requirements and Evaluation) los que son: mantenibilidad, fiabilidad y seguridad, dejando de lado los atributos de funcionalidad, rendimiento, compatibilidad, usabilidad y portabilidad. De forma complementaria, se consideró las mediciones de código duplicado, cobertura de código y el número de líneas de código. Asimismo, se buscó

[5] DRY—Don’t Repeat Yourself
[6] Sustituye a las anteriores ISO/IEC 9126 e ISO/IEC 14598

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

lograr una diferencia significativa en la calidad interna del caso de estudio antes y después del desarrollo del proceso de refactoring, no siendo el objetivo conseguir que el código fuente cumpla por completo con los principios DRY y SOLID. Por último, no fue parte de esta investigación efectuar el refactoring de la arquitectura monolítica del caso de estudio a otra arquitectura, por ejemplo, a una arquitectura basada en Microservicios.

1.6. Objetivos de la investigación

1.6.1. Objetivo general

Evaluar el efecto que tendrá en la calidad interna de una Aplicación Web Java el uso de los principios DRY y S.O.L.I.D. en un proceso de refactoring.

1.6.2. Objetivos específicos

- Evaluar la calidad interna del caso de estudio.
- Definir un proceso de refactoring para mejorar la calidad interna de una Aplicación Web Java.
- Desarrollar el proceso de refactoring definido sobre el caso de estudio.
- Evaluar en qué medida el uso de los principios DRY y S.O.L.I.D. en el proceso de refactoring desarrollado afectó la calidad interna del caso de estudio.

1.7. Hipótesis general

El uso de los principios DRY y S.O.L.I.D. en un proceso de refactoring de una Aplicación Web Java repercute positivamente en su calidad interna.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

CAPÍTULO II

REVISIÓN DE LITERATURA

2.1. Antecedentes de la investigación

- Manuel Alonso y Facundo H. Klaver, realizaron un trabajo de investigación titulado: "Aplicación de un Proceso de Refactoring guiado por Escenarios de Modificabilidad y Code Smells", con el objetivo de brindar un enfoque de refactorización que garantice una mejora en la modificabilidad del software. Concluyeron que la instanciación de este proceso de refactorización guiado por escenarios de calidad y code smells podría realizarse en cualquier otro sistema de software. Sin embargo, los resultados podrán variar de acuerdo a las características de la arquitectura del sistema. En los casos de sistemas cuyo proyecto de desarrollo cuente con documentación actualizada, requerimientos funcionales claros, atributos de calidad definidos, buena cobertura de tests y desarrolladores con conocimiento de la arquitectura; el proceso será más eficiente, ya que estos artefactos facilitan el análisis, la detección de problemas y la implementación de soluciones efectivas. En sistemas con menor grado de formalidad en su metodología de desarrollo o arquitectura o escasa cantidad de artefactos, el proceso de refactorización será más lento ya que habrá que conocer la arquitectura, documentar el proyecto para estudiar sus debilidades, elicitar requerimientos y objetivos del refactoring, implementar casos de tests, etc. De todas maneras, en esos casos el proceso también es aplicable y como resultado se logrará, además de una mejora en la calidad del sistema, una mejora en la calidad del proceso de desarrollo (Alonso & Klaver, 2015).
- Francisco Javier Pérez García, realizó una tesis doctoral titulada: "Refactoring Planning for Design Smell Correction in Object-Oriented Software", investigación orientada a mejorar la automatización de las actividades de refactorización

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Universidad Nacional del Altiplano

enfocadas a la corrección de design smell. La propuesta desarrollada se basa en la definición de las estrategias de refactorización y en la instanciación de planes de refactorización a partir de estas. Las estrategias de refactorización son especificaciones automatizables de secuencias de refactorización complejas dirigidas a un objetivo particular, como la corrección de *design smells*. Los planes de refactorización son secuencias de refactorización instanciadas a partir de estrategias de refactorización, que pueden ser aplicadas sobre un sistema para conseguir de forma efectiva ese objetivo. Las estrategias y los planes de refactorización permiten computar las refactorizaciones preparatorias requeridas, ayudando al desarrollador a resolver el incumplimiento de las precondiciones. El estudio demuestra que es una técnica apropiada para instanciar planes de refactorización y, por tanto, para respaldar procesos de refactorización complejos (Pérez García, 2011).

- Alejandra Garrido, Gustavo Rossi y Damiano Distante, realizaron un trabajo de investigación denominada: “Model Refactoring in Web Applications”, en su trabajo proponen refactorizaciones que se aplican a los modelos de diseño de una aplicación web. En particular, definen refactorizaciones en la navegación y modelos de presentación. La intención de las refactorizaciones del modelo Web es mejorar las cualidades externas como la usabilidad. También ayudan a introducir patrones web en una aplicación Web. Las conclusiones a las que llegan son: 1) Demuestran como las refactorizaciones pueden ayudar a las aplicaciones web a evolucionar, aplicando patrones web bien conocidos en su diseño, en mejorar las propiedades de calidad de uso como usabilidad. Además, demuestran como las refactorizaciones pueden ser combinadas para lograr una transformación más compleja, como una refactorización

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

desencadena a otros en la misma u otros modelos de diseño (Garrido, Rossi, & Distante, 2007).

- Edú James Baldeón Villanes, realizó una tesis titulada: "Método para la evaluación de calidad de software basada en ISO/IEC 25000", en el cual se plantea como objetivo general: "Mejorar la calidad del software a través de la aplicación de un método para la evaluación de calidad basada en ISO/IEC 25000". Al finalizar su investigación concluye en los siguientes puntos: 1) Se logró mejorar la calidad del software como resultado de la aplicación del método de evaluación de calidad de software basado en ISO/IEC 25000, 2) La cantidad de errores relacionados a requisitos funcionales disminuyó luego de la aplicación del método de evaluación de calidad, 3) La aplicación del método para la evaluación de calidad permitió asegurar que el equipo de desarrollo plasme adecuadamente lo que el usuario necesita (Badeón Villanes, 2015).

2.2. Marco teórico

2.2.1. Software y Aplicaciones Web

En (Sommerville, 2011) encontramos la definición de software como: programas de cómputo y documentación asociada. Los productos de software se desarrollan para un cliente en particular o para un mercado en general.

En el caso de las aplicaciones web, en (Pressman, 2010) hallamos una definición que indica: las aplicaciones web, llamadas "webapps", esta categoría de software centrado en redes agrupa una amplia gama de aplicaciones. En su forma más sencilla, las *webapps* son poco más que un conjunto de archivos de hipertexto vinculados que presentan información con uso de texto y gráficas limitadas. Sin embargo, desde que surgió Web 2.0, las *webapps* están evolucionando hacia ambientes de cómputo sofisticados que no sólo proveen características aisladas, funciones de cómputo y

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

contenido para el usuario final, sino que también están integradas con bases de datos corporativas y aplicaciones de negocio. Adicionalmente, (Dai, Mandel, & Ryman, 2007), mencionan que una aplicación Web simple tiene tres capas básicas: Presentación, Lógica de Negocios y Datos. Más capas se pueden definir en diferentes partes de la arquitectura.

2.2.2. Calidad de Software

(Pressman, 2010), define la calidad del software como: Proceso eficaz de software que se aplica de manera que crea un producto útil que proporciona valor medible a quienes lo producen y a quienes lo utilizan.

Por su lado (Kniberg, 2015), distingue la calidad interna y calidad externa de la siguiente forma:

- **Calidad externa:** Es lo que perciben los usuarios del sistema. Un interfaz de usuario lento y poco intuitivo es un ejemplo de baja calidad externa.
- **Calidad interna:** Se refiere a aquellos aspectos que normalmente no son visibles al usuario, pero que tienen un profundo efecto en la mantenibilidad del sistema. Cosas como consistencia del diseño del sistema, cobertura de pruebas, legibilidad del código, refactorización, etc.

Generalizando, un sistema con alta calidad interna puede, aun así, tener una baja calidad externa. Pero un sistema con baja calidad interna rara vez tendrá buena calidad externa.

Por otro lado, la ISO/IEC 25000, es una familia de normas que tiene por objetivo la creación de un marco de trabajo común para evaluar la calidad del producto software. En (ISO/IEC 25000:2014, 2017), se encuentra la definición de la medida interna de la calidad del software como: medida del grado en que un conjunto de atributos estáticos de un producto de software satisface las necesidades declaradas e implícitas para el

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

producto de software que se va a utilizar bajo condiciones especificadas; adicionalmente coloca dos anotaciones las que son: a) Los atributos estáticos incluyen aquellos que se relacionan con la arquitectura del software, la estructura y sus componentes, b) Los atributos estáticos se pueden verificar mediante revisión, inspección, simulación y/o herramientas automatizadas.

2.2.2.1. La calidad del proceso

La calidad vista desde el mundo de los procesos nos dice que la calidad del producto software está determinada por la calidad del proceso. Por proceso se entienden las actividades, tareas, entrada, salida, procedimientos, etc., para desarrollar y mantener software (Galicia, 2014).

Modelos, normas y metodologías típicas aquí son CMMI, ISO 15504 / ISO 12207, el ciclo de vida usado; incluso las metodologías ágiles entran aquí (Galicia, 2014).

2.2.2.2. La calidad del producto

Existen modelos de calidad de producto, destacando entre ellos la ISO 9126, o la nueva serie ISO 25000, que especifica diferentes dimensiones de la calidad de producto. Aunque aquí la dura tarea de evaluación recae en el uso de métricas software (Galicia, 2014).

2.2.2.3. La mala calidad del producto siempre tiene un coste

Porque la mala calidad del producto software (es decir, código espagueti, código repetido, diseño acoplado, etc.) siempre, siempre, alguien la paga, tiene un coste (lo que llamamos deuda técnica). Y sólo pueden pagarla uno de dos: el cliente o la empresa que desarrolló el software (Galicia, 2014).

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

2.2.2.4. La deuda técnica

La deuda técnica es el coste y los intereses a pagar por hacer mal las cosas. El sobre esfuerzo a pagar para mantener un producto software mal hecho, y lo que conlleva, como el coste de la mala imagen frente a los clientes, etc. Hay quien no es ni siquiera consciente de que está pagando intereses por hacer mal el software, y continúa así hasta el "default" (Galicia, 2014).

La deuda técnica al final siempre alguien la paga. O la paga el proveedor que desarrolla el software o la paga el cliente que lo usa o compra (Galicia, 2014).

2.2.2.5. La familia de normas ISO/IEC 25000

ISO/IEC 25000, conocida como SQuaRE (System and Software Quality Requirements and Evaluation), es una familia de normas que tiene por objetivo la creación de un marco de trabajo común para evaluar la calidad del producto software.

La familia ISO/IEC 25000 es el resultado de la evolución de otras normas anteriores, especialmente de las normas ISO/IEC 9126, que describe las particularidades de un modelo de calidad del producto software, e ISO/IEC 14598, que abordaba el proceso de evaluación de productos software. Esta familia de normas ISO/IEC 25000 se encuentra compuesta por cinco divisiones (Portal ISO 25000, 2017).

Figura N° 2.1: ISO/IEC 25000.

Fuente: (Portal ISO 25000, 2017).

a. ISO/IEC 2501n – División de modelo de calidad

Las normas de este apartado presentan modelos de calidad detallados incluyendo características para calidad interna, externa y en uso del producto software. Actualmente esta división se encuentra formada por (Portal ISO 25000, 2017):

- **ISO/IEC 25010 - System and software quality models**: describe el modelo de calidad para el producto software y para la calidad en uso. Esta Norma presenta las características y sub-características de calidad frente a las cuales evaluar el producto software.
- **ISO/IEC 25012 - Data quality model**: define un modelo general para la calidad de los datos, aplicable a aquellos datos que se encuentran almacenados de manera estructurada y forman parte de un Sistema de Información.

2.2.2.6. ISO/IEC 25010

El modelo de calidad representa la piedra angular en torno a la cual se establece el sistema para la evaluación de la calidad del producto. En este modelo se determinan las características de calidad que se van a tener en cuenta a la hora de evaluar las propiedades de un producto software determinado (Portal ISO 25000, 2017).

La calidad del producto software se puede interpretar como el grado en que dicho producto satisface los requisitos de sus usuarios aportando de esta manera un valor. Son precisamente estos requisitos (funcionalidad, rendimiento, seguridad, mantenibilidad, etc.) los que se encuentran representados en el modelo de calidad, el cual categoriza la calidad del producto en características y sub-características (Portal ISO 25000, 2017).

El modelo de calidad del producto definido por la ISO/IEC 25010 se encuentra compuesto por las ocho características de calidad mostradas en la Figura N° 2.2.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Figura N° 2.2: ISO/IEC 25010 Características de calidad.

Fuente: (Portal ISO 25000, 2017).

a. Fiabilidad

Capacidad de un sistema o componente para desempeñar las funciones especificadas, cuando se usa bajo unas condiciones y periodo de tiempo determinados. Esta característica se subdivide a su vez en las siguientes subcaracterísticas (Portal ISO 25000, 2017):

- **Madurez**. Capacidad del sistema para satisfacer las necesidades de fiabilidad en condiciones normales.
- **Disponibilidad**. Capacidad del sistema o componente de estar operativo y accesible para su uso cuando se requiere.
- **Tolerancia a fallos**. Capacidad del sistema o componente para operar según lo previsto en presencia de fallos hardware o software.
- **Capacidad de recuperación**. Capacidad del producto software para recuperar los datos directamente afectados y restablecer el estado deseado del sistema en caso de interrupción o fallo.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

b. Seguridad

Capacidad de protección de la información y los datos de manera que personas o sistemas no autorizados no puedan leerlos o modificarlos. Esta característica se subdivide a su vez en las siguientes sub-características (Portal ISO 25000, 2017):

- **Confidencialidad**. Capacidad de protección contra el acceso de datos e información no autorizados, ya sea accidental o deliberadamente.
- **Integridad**. Capacidad del sistema o componente para prevenir accesos o modificaciones no autorizados a datos o programas de ordenador.
- **No repudio**. Capacidad de demostrar las acciones o eventos que han tenido lugar, de manera que dichas acciones o eventos no puedan ser repudiados posteriormente.
- **Responsabilidad**. Capacidad de rastrear de forma inequívoca las acciones de una entidad.
- **Autenticidad**. Capacidad de demostrar la identidad de un sujeto o un recurso.

c. Mantenibilidad

Esta característica representa la capacidad del producto software para ser modificado efectiva y eficientemente, debido a necesidades evolutivas, correctivas o perfectivas. Esta característica se subdivide a su vez en las siguientes sub-características (Portal ISO 25000, 2017):

- **Modularidad:** Capacidad de un sistema o programa de ordenador (compuesto de componentes discretos) que permite que un cambio en un componente tenga un impacto mínimo en los demás.
- **Reusabilidad:** Capacidad de un activo que permite que sea utilizado en más de un sistema software o en la construcción de otros activos.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

- **Analizabilidad:** Facilidad con la que se puede evaluar el impacto de un determinado cambio sobre el resto del software, diagnosticar las deficiencias o causas de fallos en el software, o identificar las partes a modificar.
- **Capacidad para ser modificado:** Capacidad del producto que permite que sea modificado de forma efectiva y eficiente sin introducir defectos o degradar el desempeño.
- **Capacidad para ser probado:** Facilidad con la que se pueden establecer criterios de prueba para un sistema o componente y con la que se pueden llevar a cabo las pruebas para determinar si se cumplen dichos criterios.

2.2.2.7. Bug

(SonarQube, 2018), lo define como un problema que representa algo mal en el código. Si esto no se ha roto todavía, lo hará, y probablemente en el peor momento posible.

2.2.2.8. Code Smell

Según (SonarQube, 2018), un code smell, es un problema relacionado con la mantenibilidad en el código. Dejarlo como está significa que, en el mejor de los casos, los mantenedores tendrán más dificultades en hacer cambios en el código de lo que deberían. En el peor de los casos, estarán tan confundidos por el estado del código que introducirán errores adicionales a medida que realizan cambios.

2.2.2.9. Vulnerabilidad

Un problema relacionado con la seguridad que representa una puerta trasera para los atacantes (SonarQube, 2018).

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

2.2.3. Principio DRY

DRY - Don´t Repeat Yourself.

No te repitas

(Hunt & Thomas, 2000)

DRY o Don't Repeat YourSelf implica que, cualquier funcionalidad existente en un programa debe existir de forma única en él, o lo que es lo mismo, no debemos tener bloques de código repetidos (Álvarez Caules, 2012).

(Hunt & Thomas, 2000), puntualizan que la única manera de desarrollar software confiablemente y hacer desarrollos más fáciles de entender y mantener, es seguir lo que llaman el principio DRY: "cada pieza de conocimiento debe tener una representación sencilla, individual, autoritativa dentro de un sistema".

2.2.3.1. ¿Cómo surge la duplicación?

Según (Hunt & Thomas, 2000), la mayor parte de la duplicación que vemos cae en una de las siguientes categorías:

- **Duplicación impuesta:** los desarrolladores sienten que no tienen elección, el ambiente parece requerir duplicación.
- **Duplicación inadvertida:** los desarrolladores no se dan cuenta que están duplicando la información.
- **Duplicación impaciente:** los desarrolladores se ponen perezosos y duplican porque parece más fácil.
- **Duplicación de entre desarrolladores:** múltiples personas en un equipo (o en diferentes equipos) duplican una pieza de información.

2.2.4. Principios S.O.L.I.D.

SOLID es el acrónimo para un conjunto de prácticas que, cuando se implementan juntos, hacen que el código sea adaptable al cambio. Las practicas SOLID

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

fueron introducidas por Bob Martin hace casi 15 años. Aun así, estas prácticas no son tan ampliamente conocidas como podría ser y quizás debería serlo (McLean Hall, 2014). La Figura N° 2.3, muestra los cinco principios que conforman SOLID.

Figura N° 2.3: Principios de diseño S.O.L.I.D.

Fuente: (Martin & Kriens, 2012).

La Tabla N° 2.1, presenta una breve descripción de cada uno de estos principios.

Tabla N° 2.1: Principios SOLID.

SRP	El Principio de Responsabilidad Única	Una clase debe tener una, sólo una, razón para cambiar.
OCP	El Principio Abierto/Cerrado	Debe ser capaz de extender un comportamiento de clases, sin modificarlo.
LSP	El Principio de Sustitución de Liskov	Las clases derivadas deben ser sustituibles por sus clases base.
ISP	El Principio de Segregación de Interfaces	Hacer interfaces de grano fino que son específicos del cliente.
DIP	El Principio de Inversión de Dependencia.	Depende de abstracciones, no de concreciones.

Fuente: (UncleBob, 2017).

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

2.2.4.1. Principio de Responsabilidad Única (SRP)

The Single-Responsibility Principle (SRP).
Una clase debe tener sólo una razón para cambiar.
(Martin & Martin, 2006).

El principio de responsabilidad única (SRP) instruye a los desarrolladores a escribir código que tiene una y sólo una razón para cambiar. Si una clase tiene más de una razón para cambiar, tiene más de una responsabilidad. Las clases con más de una sola responsabilidad deben dividirse en clases más pequeñas que debe tener solo una responsabilidad y razón para cambiar (McLean Hall, 2014).

Según (Martin, 2012), este principio nos indica la definición de responsabilidad y una directriz para el tamaño de la clase.

Todos los sistemas tienen una gran lógica y complejidad. El objetivo principal para gestionar dicha complejidad es organizarla para que un programador sepa dónde buscar y comprenda la complejidad directamente afectada en cada momento concreto. Por el contrario, un sistema con clases multipropósito de mayor tamaño nos obliga a buscar entre numerosos elementos que no siempre necesitamos conocer (Martin, 2012).

(Martin, 2012), puntualiza que los sistemas deben estar formados por muchas clases reducidas, no por algunas de gran tamaño. Cada clase reducida encapsula una única responsabilidad, tiene un solo motivo para cambiar y colabora con algunas otras para obtener los comportamientos deseados del sistema.

La Figura N° 2.4, presenta un ejemplo de una clase con una única responsabilidad.

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

Figura N° 2.4: Una clase con una única responsabilidad.

```
public class Version {
        public int getMajorVersionNumber();
        public int getMinorVersionNumber();
        public int getBuildNumber();
}
```

Fuente: (Martin, 2012).

2.2.4.2. Principio Abierto/Cerrado (OCP)

The Open/Closed Principle (OCP).

Las entidades de software (Clases, módulos, funciones, etc.) deben estar abiertas para extensión, pero cerradas para modificación.

(Martin & Martin, 2006).

Según (McLean Hall, 2014) hay dos definiciones del principio abierto/cerrado que deben ser examinadas.

- **La definición de Meyer:** Las entidades de software deben estar abiertas para la extensión, pero cerradas para su modificación.
- **La definición de Martin:**
  - *"Abierto para extensión"*. Esto significa que el comportamiento del módulo puede ampliarse. A medida que cambian los requisitos de la aplicación, podemos ampliar el módulo con nuevos comportamientos que satisfagan esos cambios. En otras palabras, somos capaces de cambiar lo que hace el módulo.
  - *"Cerrado para modificación"*. Extender el comportamiento de un módulo no resulta en los cambios en el código fuente o binario del módulo. La versión binaria ejecutable del módulo, ya sea en una librería enlazable, una DLL, o un .jar de Java permanece intacto.

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

2.2.4.3. Principio de Sustitución de Liskov (LSP)

The Liskov Substution Principle (LSP).

Los subtipos deben ser sustituibles por sus tipos base.

(Martin & Martin, 2006).

Barbara Liskov escribió este principio en 1988. Ella dijo: "Lo que se busca aquí es algo así como la siguiente propiedad de sustitución: si para cada objeto o_1 de tipo *S* hay un objeto o_2 de tipo *T* tal que para todos los programas *P* definidos en términos de *T*, el comportamiento de *P* no cambia cuando se sustituye o_1 por o_2 entonces S es un subtipo de *T*."

La importancia de este principio se hace evidente cuando se consideran las consecuencias de violarla. Supongamos que tenemos una función *f* que toma como argumento una referencia a alguna clase base *B*. presume también que cuando se pasa *f* en la forma de *B*, algún derivado *D* de *B* hace que *f* se comporte mal. Entonces *D* viola LSP. Claramente, *D* es frágil en presencia de *f*.

Los autores de *f* estarán tentados a poner en algún tipo de prueba para *D* para que *f* pueda comportarse correctamente cuando se le pasa una *D*. esta prueba viola OCP porque ahora, *f* no está cerrado a todos los diversos derivados de *B*. estas pruebas son un olor de código que son el resultado de desarrolladores inexpertos o, lo que es peor, desarrolladores en una prisa de reaccionar a las violaciones de LSP (Martin & Martin, 2006).

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

2.2.4.4. Principio de Segregación de Interfaces (ISP)

The Interface Segregation Principle (ISP).
Los clientes no deben verse obligados a depender de métodos que no utilizan.
(Martin & Martin, 2006).

Este principio se ocupa de las desventajas de las interfaces "gordas". Clases cuyas interfaces no son cohesivas tienen interfaces "gordas". En otras palabras, las interfaces de la clase pueden dividirse en grupos de métodos. Cada grupo sirve a un conjunto diferente de clientes. Así, algunos clientes utilizan un grupo de métodos y otros clientes utilizan los otros grupos.

ISP reconoce que hay objetos que requieren interfaces no cohesivas; sin embargo, sugiere que los clientes no deben saber acerca de ellos como una sola clase. En cambio, los clientes deben saber de clases bases abstractas que tienen interfaces cohesivas (Martin & Martin, 2006).

Cuando los clientes se ven obligados a depender de métodos que no utilizan, esos clientes están sujetos a cambios de esos métodos. Esto resulta en un acoplamiento inadvertido entre todos los clientes. Dicho de otra manera, cuando un cliente depende de una clase que contiene métodos que el cliente no utiliza pero que otros clientes utilizan, ese cliente se verá afectado por los cambios que esos otros clientes fuerzan en la clase. Nos gustaría evitar tales acoplamientos donde sea posible, y por eso queremos separar las interfaces (Martin & Martin, 2006).

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

2.2.4.5. Principio de Inversión de Dependencia (DIP)

The Dependency-Inversion Principle (DIP).

A. *Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.*

B. *Las abstracciones no deben depender de detalles. Los detalles deben depender de abstracciones.*

(Martin & Martin, 2006).

Para (Blé Jurado, 2010), la inversión de dependencias da origen a la conocida inyección de dependencias, una de las mejores técnicas para lidiar con las colaboraciones entre clases, produciendo un código reutilizable, sobrio y preparado para cambiar sin producir efectos “bola de nieve”. DIP explica que un módulo concreto *A*, no debe depender directamente de otro módulo concreto *B*, sino de una abstracción de *B*. tal abstracción es una interfaz o una clase (que podría ser abstracta) que sirve de base para un conjunto de clases hijas.

2.2.5. Cambiando el Software

(Feathers, 2004), explica las razones principales para cambiar el software, estas se describen a continuación.

2.2.5.1. Adicionar una funcionalidad y corregir un error

El comportamiento es lo más importante del software. Es de lo que los usuarios dependen. A los usuarios les gusta cuando añadimos comportamiento (siempre que sea lo que realmente querían), pero si cambiamos o eliminamos el comportamiento del que dependen (se introducen errores), dejan de confiar en nosotros (Feathers, 2004).

2.2.5.2. Mejorar el diseño

La mejora del diseño es un tipo diferente de cambio de software. Cuando queremos alterar la estructura del software para hacerlo más fácil de mantener,

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

generalmente queremos mantener su comportamiento intacto también. Cuando eliminamos comportamiento en el proceso, obtenemos un error. Una de las principales razones por la que muchos programadores no intentan mejorar el diseño a menudo es porque es relativamente fácil perder el comportamiento o crear un mal comportamiento en el proceso de hacerlo (Feathers, 2004).

El acto de mejorar el diseño sin cambiar su comportamiento se llama refactoring. La idea detrás del refactoring es que podemos hacer el software más fácil de mantener sin cambiar su comportamiento si escribimos pruebas para asegurarnos de que el comportamiento existente no cambie y tomar pequeños pasos para verificarlo a lo largo del proceso. El refactoring difiere de la limpieza en general en que no sólo estamos haciendo cosas de bajo riesgo, como el reformateo del código fuente, o cosas invasivas y arriesgadas como reescribir fragmentos de la misma. En cambio, estamos haciendo una serie de pequeñas modificaciones estructurales, soportado por pruebas para hacer el código más fácil de cambiar. La clave del refactoring desde el punto de vista del cambio es que no se supone que haya cambios funcionales cuando refactorizas (aunque el comportamiento puede cambiar algo porque los cambios estructurales que hace pueden alterar el rendimiento, para bien o para mal) (Feathers, 2004).

2.2.5.3. Optimización

La optimización es como la refactorización, pero cuando lo hacemos, tenemos un objetivo diferente. Con el refactoring y la optimización, decimos: “Vamos a mantener la funcionalidad exactamente igual cuando hacemos cambios, pero vamos a cambiar algo más”. En el refactoring, la “otra cosa” es la estructura del programa; queremos que sea más fácil de mantener. En la optimización, el “algo más” es algún recurso utilizado por el programa, generalmente el tiempo o la memoria (Feathers, 2004).

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

2.2.5.4. Riesgo al cambio

Preservar el comportamiento es un gran desafío. Cuando necesitamos hacer cambio y preservar el comportamiento, puede implicar un riesgo considerable (Feathers, 2004).

Para mitigar el riesgo, tenemos que hacer tres preguntas:

1) ¿Qué cambios tenemos que hacer?
2) ¿Cómo sabremos que las hemos hecho correctamente?
3) ¿Cómo sabremos que no hemos roto nada?

2.2.5.5. El algoritmo de cambio de código heredado

Los pasos descritos en (Feathers, 2004) de este algoritmo son:

1) **Identificar los puntos de cambio.** Los lugares donde necesita hacer sus cambios dependen de manera sensible de su arquitectura.
2) **Encontrar los puntos de prueba**. En algunos casos, encontrar lugares para escribir pruebas es fácil, pero en código heredado a menudo puede ser difícil.
3) **Romper dependencias**. Las dependencias son a menudo el impedimento más obvio para las pruebas. Las dos formas en que este problema se manifiesta son la dificultad de instanciar objetos en las pruebas y la dificultad de ejecutar los métodos en las pruebas.
4) **Escribir pruebas**.
5) **Realizar cambios y refactorizar**.

2.2.6. Refactoring

Refactoring es el proceso de cambio de un sistema de software de manera tal que no altera el comportamiento externo del código, pero mejora su estructura interna. Es una manera disciplinada de limpiar código que minimiza las posibilidades de introducir

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

errores. En esencia cuando se refactoriza se está mejorando el diseño del código después de que ha sido escrito (Fowler, 1999).

Por su lado (Feathers, 2004), precisa que el refactoring es una técnica básica para mejorar el código.

(Fowler, 1999), explica las razones para realizar una refactorización, estas son explicadas a continuación.

2.2.6.1. Refactoring mejora el diseño del Software

Sin refactoring, el diseño del programa decaerá. A medida que las personas cambian los objetivos a corto plazo o cambios realizados sin una comprensión completa del diseño del código, el código pierde su estructura. Se hace más difícil ver el diseño leyendo el código. Refactoring es más bien como ordenar el código. Se trabaja para eliminar los bits que no están realmente en el lugar correcto. La pérdida de la estructura del código tiene un efecto acumulativo. Cuando más difícil es ver el diseño del código, más difícil es preservarlo, y más rápidamente se descompone. Refactorizaciones regulares ayudan a que el código retenga su forma (Fowler, 1999).

2.2.6.2. Refactoring hace que el software sea más fácil de entender

(Fowler, 1999), menciona que la programación es en muchos sentidos una conversación con una computadora. Escribe un código que le indica a la computadora qué hacer, y responde haciendo exactamente lo que usted le dice. Con el tiempo, cierras la brecha entre lo que quieres que haga y lo que le dices que haga. La programación en este modo tiene que ver con decirle exactamente lo que quieres. Pero hay otro usuario de su código fuente. Alguien intentará leer su código en unos meses para hacer algunos cambios. Fácilmente olvidamos el usuario extra del código, pero ese usuario es realmente el más importante. ¿A quién le importa si la computadora toma algunos ciclos

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

más para compilar algo?, no importa si se necesita un programador por una semana para hacer un cambio que habría tardado sólo una hora si hubiera entendido su código.

El refactoring le ayuda a hacer su código más legible. Cuando refactoriza tiene código que funciona, pero no está estructurado de forma ideal. Invertir un poco de tiempo al refactoring puede hacer que el código comunique mejor su propósito. La programación en este modo se trata de decir exactamente lo que quieres decir (Fowler, 1999).

2.2.6.3. Refactoring le ayuda a encontrar errores

(Fowler, 1999), menciona que la ayuda en la comprensión del código también ayuda a detectar errores. Si se refactoriza el código, se trabaja profundamente en comprender lo que el código hace, y se puede volcar ese nuevo entendimiento de nuevo en el código. Al aclarar la estructura del programa, se aclara ciertas suposiciones que se hicieron, hasta el punto en que ni siquiera se puede evitar detectar los errores.

2.2.6.4. Refactoring le ayuda a programar más rápido

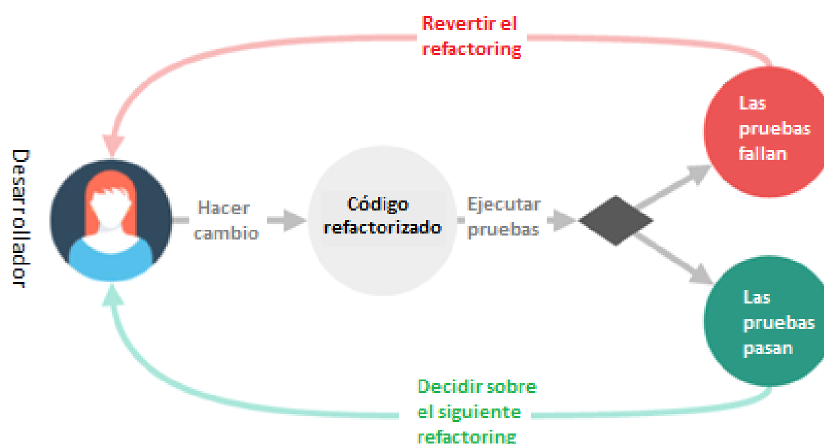
Un buen diseño es esencial para mantener la velocidad en el desarrollo de software. Refactoring le ayuda a desarrollar software más rápidamente, ya que detiene el diseño del sistema en decaimiento. Puede incluso mejorar un diseño (Fowler, 1999).

2.2.7. El ciclo del refactoring

Este ciclo comienza con nosotros haciendo un cambio, ejecutando las pruebas y viendo si las pruebas pasan. Si lo hacen, podemos estar seguros de que nuestra refactorización no afectó adversamente el sistema y continuar con otra refactorización. Si las pruebas fallan, sabemos que hemos alterado el sistema de alguna manera notoriamente externa y debemos deshacer nuestros cambios (que en realidad no fue una refactorización, ya que cambió el comportamiento externo del sistema) (DZone, 2018). Este proceso se ilustra en la Figura N° 2.5.

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

Figura N° 2.5: Ciclo del refactoring.



Fuente: (DZone, 2018).

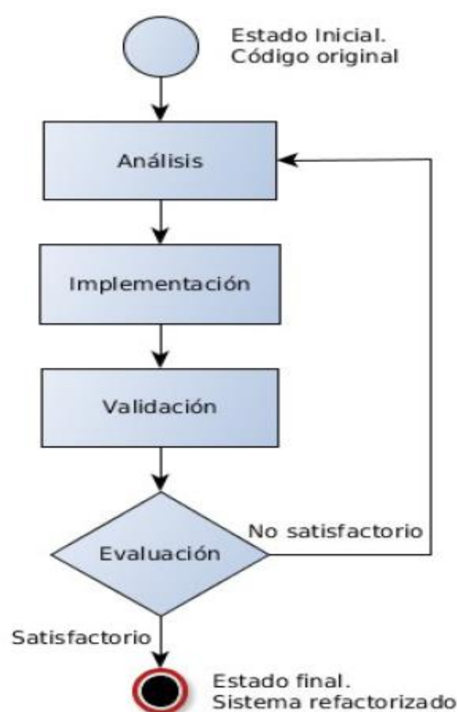
2.2.8. Proceso de refactoring

(Alonso & Klaver, 2015), definen un proceso de refactorización en el cual se pueden identificar un conjunto de etapas iterables de manera incremental, que conforman un proceso guiado por escenarios de modificabilidad y code smells. Las etapas que definen son las siguientes:

- **Etapas de Análisis:** al comienzo del proceso es necesario el análisis del código del sistema para detectar errores y síntomas de problemas asociados al desarrollo y las malas prácticas del diseño.
- **Etapas de Implementación:** con el diagnóstico del sistema, se pasa a la proposición y realización de modificación con el objetivo de solucionar esos problemas detectados.
- **Etapas de Validación:** finalmente se realiza una evaluación de los cambios desarrollados y su impacto en la calidad del sistema. Una vez realizada la evaluación se decide si conviene llevar adelante una nueva iteración sobre el proceso para refinar la refactorización.

La Figura N° 2.6, muestra el resumen del proceso.

Figura N° 2.6: Proceso de refactorización resumido.



Fuente: (Alonso & Klaver, 2015).

2.2.9. Pruebas del Software

2.2.9.1. Tipos de test

(Blé Jurado, 2010), menciona y define los siguientes tipos de test:

a. Test de aceptación

Es un test que permite comprobar que el software cumple con un requisito de negocio.

b. Test funcionales

Todos los tests son en realidad funcionales, puesto que todos ejercitan alguna función del SUT[7], aunque en el nivel más elemental sea un método de una clase. No obstante, cuando se habla del aspecto funcional, se distingue entre test funcional y test

[7] Subject Under Test; el código que estamos probando.

no funcional. Un test funcional es un subconjunto de los test de aceptación. Es decir, comprueban alguna funcionalidad con valor de negocio. Un test funcional es un test de aceptación, pero, uno de aceptación, no tiene por qué ser funcional.

c. Test de sistema

Es el mayor de los tests de integración, ya que integra varias partes del sistema. Se trata de un test que puede ir, incluso, de extremo a extremo de la aplicación o del sistema. Se habla de sistema porque es un término más general que aplicación, pero no se refiere a administración de sistemas, no es que estemos probando el servidor web o el servidor SMTP, aunque, tales servicios podrían ser una parte de nuestro sistema. Así pues, un test del sistema se ejercita tal cual lo haría el usuario humano, usando los mismos puntos de entrada (aquí si es la interfaz gráfica) y llegando a modificar la base de datos o lo que haya en el otro extremo.

Los tests de sistema son muy frágiles en el sentido de que cualquier cambio en cualquiera de las partes que componen el sistema, puede romperlos. No es recomendable escribir un gran número de ellos por su fragilidad.

d. Tests unitarios

Son los tests más importantes para el practicante TDD[8], los ineludibles. Cada test unitario o test de unidad (unit test en inglés) es un paso que andamos en el camino de la implementación del software. Todo test unitario debe ser:

- **Atómico:** significa que el test prueba la mínima cantidad de funcionalidad posible. Esto es, probará un solo comportamiento de un método de una clase. El mismo método puede presentar distintas respuestas ante distintas entradas o

[8] Desarrollo guiado por pruebas.

distinto contexto. El test unitario se ocupará exclusivamente de uno de esos comportamientos, es decir, de un único camino de ejecución.

- **Independiente:** significa que un test no puede depender de otros para producir un resultado satisfactorio. No puede ser parte de una secuencia de tests que se deba ejecutar en un determinado orden. Debe funcionar siempre igual independientemente de que se ejecuten otros tests o no.
- **Inocuo:** significa que no altera el estado del sistema. Al ejecutarlo una vez, produce exactamente el mismo resultado que al ejecutarlo veinte veces. No altera la base de datos, ni envía emails ni crea ficheros, ni los borra. Es como si no se hubiera ejecutado.
- **Rápido:** porque ejecutamos un gran número de tests cada pocos minutos y se ha demostrado que tener que esperar unos cuantos segundos cada rato, resulta muy improductivo. Un solo test tendría que ejecutarse en una pequeña fracción de segundo.

e. Tests de integración

Los tests de integración se pueden ver como tests de sistema pequeños. Típicamente, también se escriben usando herramientas xUnit y tienen un aspecto parecido a los tests unitarios, sólo que estos pueden romper las reglas. Como su nombre indica, integración significa que ayuda a unir distintas partes del sistema. Un test de integración puede escribir y leer de base de datos para comprobar que, efectivamente, la lógica de negocio entiende datos reales. Es el complemento a los tests unitarios, donde habíamos "falseado" el acceso a datos para limitarnos a trabajar con la lógica de manera aislada. Un test de integración podría ser aquel que ejecuta la capa de negocio y después consulta la base de datos para afirmar que todo el proceso, desde negocio hacia abajo, fue bien. Son, por tanto, de granularidad más gruesa y más frágiles que los tests

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

unitarios, con lo que el número de tests de integración tiende a ser menor que el número de test unitarios. Una vez que se ha probado que dos módulos, objetos o capas se integran bien, no es necesario repetir el test para otra variante de la lógica de negocio; para eso habrá varios tests unitarios.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

TESIS UNA - PUNO

Universidad Nacional del Altiplano

CAPÍTULO III

MATERIALES Y MÉTODOS

3.1. Lugar de estudio

La presente investigación se realizó en la región de Puno – Perú. El caso de estudio elegido (Aplicación Web Java) se encuentra en uso en una institución pública de la ciudad de Puno.

3.2. Población

La población de estudio está constituida por los paquetes Java que conforman el caso de estudio. Considerando que las clases como piezas de código permiten la determinación de incidencias (code smells, bugs y vulnerabilidades), la suma de estas incidencias a nivel de clases de un paquete, dan como resultado el total de incidencias del paquete, y la suma de incidencias de los paquetes da el total de incidencias de la aplicación.

3.3. Muestra

Debido a que la población es de un tamaño pequeño (18 paquetes), se consideró la muestra igual a la población. Al finalizar el refactoring del caso de estudio, se obtuvo 56 paquetes Java, el cual se usó para realizar la prueba de hipótesis con muestras independientes.

3.4. Método de investigación

La presente investigación es de tipo cuantitativo correlacional que tiene como propósito conocer la relación que exista entre dos o más conceptos, categorías o variables en un contexto en particular; se realizó a través de un diseño cuasi-experimental.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

3.5. Técnicas e instrumentos de recolección de datos

3.5.1. Técnicas e instrumentos de recolección de datos

Para obtener los datos de la calidad interna del caso de estudio, se utilizó la herramienta SonarQube. Los resultados obtenidos corresponden a los siguientes atributos de calidad:

- Mantenibilidad: medido por el número de code smells.
- Fiabilidad: medido por el número bugs.
- Seguridad: medido por el número de vulnerabilidades.

Adicionalmente, se obtuvo las siguientes mediciones:

- Porcentaje de código duplicado.
- Cobertura de código.
- Tamaño total del producto (medido por líneas de código).

3.5.2. Procedimientos de recolección de datos

Los datos utilizados para la prueba de esta investigación fueron recopilados por medio del análisis del código fuente del caso de estudio (archivos con extensión *.java*) con el software SonarQube. Se dejó de lado los archivos con extensión: *xhtml, xml, properties, js, css,* u otros que no formen parte del desarrollo backend. El análisis de los paquetes Java se efectuó antes y después del proceso de refactoring desarrollado.

3.5.3. Métodos de tratamiento de datos

El tratamiento de datos se realizó de la siguiente manera:

- Análisis del código fuente del caso de estudio antes del proceso de refactoring.
- Análisis del código fuente del caso de estudio después del proceso de refactoring.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

- Validación de la hipótesis estadística mediante la prueba U de Mann-Whitney[9].

3.6. Operacionalización de variables

Tabla N° 3.1: Operacionalización de variables.

	Variable	Dimensión	Escala	Indicador
Independiente	Uso de los principios DRY y SOLID en el proceso de refactoring de la Aplicación Web Java	Usó el principio DRY	Nominal	0 = no se usa en el refactoring 1 = se usa en el refactoring
		Usó algún principio S.O.L.I.D.	Nominal	0 = no se usa en el refactoring 1 = se usa en el refactoring
Dependiente	Calidad interna de la Aplicación Web Java	Mantenibilidad	Numérica	Número de Code Smells
		Fiabilidad	Numérica	Número de Bugs
		Seguridad	Numérica	Número de Vulnerabilidades.

Elaboración: Propia.

3.7. Métodos empleados

3.7.1. Evaluación de la calidad interna del caso de estudio

La evaluación de la calidad interna del caso de estudio se efectuó con la herramienta de análisis SonarQube. Los resultados del número de bugs, número de vulnerabilidades y número de code smells, se hallaron con las reglas[10] que esta herramienta analítica posee. La Figura N° I.1 del Anexo I, muestra el perfil/Java de

---

[9] Es una prueba no paramétrica que contrasta dos muestras independientes, dando a conocer si existen o no diferencias significativas entre ambas.

[10] Un estándar de codificación o práctica que debe seguirse. El incumplimiento de las reglas de codificación conlleva errores, vulnerabilidades, zonas activas de seguridad y olores de código. Las reglas pueden verificar la calidad en archivos de código o pruebas unitarias (SonarQube, 2018).

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

SonarQube usado, el cual tiene 276 reglas en total (108 reglas activas de bugs, 18 reglas activas de vulnerabilidades y 150 reglas activas de code smells).

3.7.2. Definición del proceso de refactoring para mejorar la calidad interna de una Aplicación Web Java

Se usó BPMN[11] para representar gráficamente el proceso de refactoring definido, el que consta de cinco etapas. En la etapa de desarrollo se empleó el algoritmo de cambio de código heredado descrito por (Feathers, 2004).

3.7.3. Desarrollo del proceso de refactoring sobre el caso de estudio

El desarrollo del proceso de refactoring se realizó en dos etapas. Primero se llevó a cabo la restructuración y modularización del caso de estudio, para esta labor se usó la herramienta Maven; la segunda etapa consistió en realizar refactorizaciones a cada capa de la arquitectura del caso de estudio. En cada iteración se siguió los pasos indicados por el algoritmo de cambio de código heredado, efectuándose diversas refactorizaciones con la finalidad de que el código fuente no infrinja alguno de los principios DRY o SOLID.

Durante este proceso, se utilizó la herramienta ObjectAid para obtener los diagramas de clases de UML[12] del código original y del código refactorizado. Además, el uso de Eclipse IDE[13] fue indispensable para realizar las diferentes refactorizaciones.

Al término de cada iteración del refactoring, se realizó un nuevo análisis del código fuente con SonarQube, con la finalidad de obtener los nuevos resultados de la calidad interna, y proseguir con las refactorizaciones de las capas siguientes.

---

[11] Modelo y notación de procesos de negocio.
[12] El Lenguaje Unificado de Modelado es un lenguaje para especificar, visualizar, construir y documentar los artefactos de los sistemas software (Larman, 2003).
[13] Entorno de Desarrollo Integrado.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

3.7.4. Evaluación de la medida en que el uso de los principios DRY y S.O.L.I.D. en el proceso de refactoring desarrollado afectó la calidad interna del caso de estudio

Para evaluar en qué medida el proceso de refactoring desarrollado afectó la calidad interna del caso de estudio, se comparó los resultados obtenidos de los atributos de calidad: seguridad (N° de vulnerabilidades), fiabilidad (N° de bugs) y mantenibilidad (N° de code smells), antes y después del proceso ejecutado; también, se comparó los resultados sobre el código duplicado, tamaño de la aplicación y cobertura de código.

3.8. Materiales empleados

A continuación, se enumeran y describen las herramientas de software y frameworks[14] que fueron utilizados en esta investigación.

3.8.1. Eclipse IDE

Eclipse es una plataforma universal de herramientas, un entorno de desarrollo integrado (IDE) extensible y abierto para cualquier cosa y nada en particular (Dai, Mandel, & Ryman, 2007). El desarrollo de todo el proceso de refactoring se realizó usando Eclipse IDE, aprovechando las herramientas de formateo de código y de refactoring que posee.

3.8.2. Maven

Es una herramienta que se puede utilizar para construir y administrar cualquier proyecto basado en Java (The Apache Software Foundation, 2018). En la presente investigación se utilizó Maven para la administración del proyecto del caso de estudio.

---

[14] Un framework, entorno de trabajo o marco de trabajo, es un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar.

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

3.8.3. SonarQube

El software SonarQube (anteriormente llamado Sonar) es una plataforma de gestión de calidad de código abierto, dedicada a analizar y medir continuamente la calidad técnica, desde la cartera de proyectos hasta un método. Sus características son (SonarSource S.A, 2017):

- Inspección continúa.
- Detecta problemas difíciles.
- Multi-Lenguaje.
- Integración DevOps.
- Calidad centralizada.

El uso de SonarQube –versión 6.2- en la presente investigación fue esencial para evaluar la calidad interna del caso de estudio. El análisis del código fuente del caso de estudio durante el desarrollo del proceso de refactoring se basó en las reglas que esta herramienta proporciona. Así mismo, el resultado de la calidad interna inicial y final del caso de estudio se obtuvo de los reportes de análisis que proporciona.

3.8.4. DBEaver

Herramienta de base de datos multiplataforma gratuita para desarrolladores, programadores SQL, administradores de bases de datos y analistas. Admite las bases de datos: MySQL, PostgreSQL, MariaDB, SQLite, Oracle, DB2, SQL Server, Sybase, MS Access, Teradata, Firebird, Derby, etc (DBeaver, 2018). En la presente investigación se utilizó DBEaver para tener acceso a la base de datos del caso de estudio, con la finalidad de conocer su estructura y los objetos que posee (tablas, vistas, triggers, etc).

3.8.5. MySQL

Es un sistema de administración de bases de datos relacional (RDBMS). Se trata de un programa capaz de almacenar una enorme cantidad de datos de gran variedad y de

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

distribuirlos para cubrir las necesidades de cualquier tipo de organización, desde pequeños establecimientos comerciales a grandes empresas y organismos administrativos. MySQL compite con sistemas RDBMS propietarios conocidos, como Oracle, SQL Server y DB2 (Gilfillan). El uso de MySQL fue necesario porque el caso de estudio implementa su base de datos en este gestor.

3.8.6. Modelio

Modelio es un entorno de modelado de código abierto que soporta los estándares principales: UML, BPMN, MDA, SysML (modelio, 2018). En esta investigación se usó Modelio para graficar los diagramas UML que permitieron mostrar el diagrama de casos de uso[15] y de arquitectura del caso de estudio.

3.8.7. ObjectAid UML

El ObjectAid UML Explorer es una herramienta de visualización de código ágil y liviano para Eclipse IDE. Utiliza la notación UML para mostrar una representación gráfica del código existente (ObjectAid, 2018). En la presente investigación se utilizó ObjectAid para obtener el diagrama de clases desde el código Java de las clases implementadas en el caso de estudio, con la finalidad de mostrar su diseño inicial y final en el proceso de refactoring.

3.8.8. Bizagi Process Modeler

Es un software utilizado para diagramar, documentar y simular procesos usando la notación estándar BPMN. En la presente investigación se utilizó este software para el modelado del proceso de refactoring de la aplicación Web Java.

[15] Un diagrama de casos de uso explica gráficamente un conjunto de casos de uso de un sistema, los actores y la relación entre éstos y los casos de uso (Larman, 2003).

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

3.8.9. JavaServer Faces

JSF es un estándar de JCP, JSR 127 [JSR127], que define un conjunto de etiquetas JSP y clases de Java para simplificar el desarrollo de la interfaz de usuario Web. Se espera que JSF estandarice las herramientas y los componentes al proporcionar un marco de un solo componente para JSP y servlets. JSF proporciona un marco para la capa de presentación y también se basa en los conceptos de MVC. JSF forma parte de la especificación Java EE 5. (Dai, Mandel, & Ryman, 2007). El caso de estudio usa el framework JSF, por tanto, su uso fue necesario en la presente investigación.

3.8.10. Spring Framework

Spring es un framework de código abierto, creado originalmente por Rod Johnson. Spring se creó para abordar la complejidad del desarrollo de aplicaciones empresariales y hace posible utilizar JavaBeans simples para lograr cosas que anteriormente solo eran posibles con EJB. Pero la utilidad de Spring no se limita al desarrollo del lado del servidor. Cualquier aplicación Java puede beneficiarse con Spring en términos de simplicidad, capacidad de prueba y acoplamiento flexible (Walls, 2015). Sus características principales son la inyección de dependencia (DI) y la programación orientada a aspectos (AOP) (Walls, 2015). El caso de estudio usa el Framework Spring para aprovechar la inyección de dependencias, y en esta investigación se amplió el uso de este framework para el desarrollo de las pruebas de integración con la base de datos.

3.8.11. Java Persistence API (JPA)

La API de persistencia de Java (JPA) es una solución basada en estándares Java para la persistencia. Persistencia utiliza un enfoque de mapeo objeto/relacional para cerrar la brecha entre un modelo orientado a objetos y una base de datos relacional. La API Java Persistence también se puede usar en aplicaciones Java SE fuera del entorno Java EE

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

(Jendrock, Cervera-Navarro, Evans, Haase, & Markito, 2014). En la presente investigación el uso de JPA fue necesario porque su capa de persistencia utiliza este API.

3.8.12. Hibernate

Hibernate es una herramienta completa de mapeo objeto/relacional que proporciona todos los beneficios de un ORM[16]. La API con la que trabajas en Hibernate es nativa y está diseñada por los desarrolladores de Hibernate (Bauer & King, 2007). En la presente investigación se utilizó Hibernate, para conseguir los refactorings en la capa de persistencia del caso de estudio.

3.8.13. JUnit Framework

JUnit es un framework de código abierto para Java. Fue creado por Kent Beck alrededor de 1997, y desde entonces ha sido, de facto, la herramienta de prueba estándar para los desarrolladores de Java. Es compatible con todos los IDEs (Eclipse, IntelliJ IDEA), herramientas de compilación (Ant, Maven, Gradle) y con frameworks populares (por ejemplo, Spring). JUnit tiene una amplia comunidad de usuarios y se complementa con una variedad de interesantes proyectos de extensión. Fue construido especialmente para pruebas unitarias, pero también se usa ampliamente para otros tipos de prueba (Kaczanowski, 2013). En la presente investigación el uso del JUnit fue necesario para la creación de las pruebas unitarias y de integración; contar con pruebas unitarias podría considerarse como un pre-requisito del refactoring.

[16] Mapeo Objeto/Relacional

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

3.8.14. Mockito Framework

Mockito es un mocking framework relativamente nuevo (o más bien un test-spy framework). Nació en el cuarto trimestre de 2007 y ha madurado rápidamente en un producto de alta calidad. Ofrece control total sobre el proceso de mocking y “te permite escribir pruebas hermosas con una API limpia y simple”. Originalmente, Mockito se derivaba de Easymock, pero ha evolucionado sustancialmente, y ahora difiere en muchos aspectos de su predecesor. Es muy fácil usar Mockito en combinación con JUnit (Kaczanowski, 2013). En esta investigación se usó Mockito para romper las dependencias entre las clases con la finalidad de conseguir pruebas unitarias.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

TESIS UNA - PUNO

Universidad Nacional del Altiplano

CAPÍTULO IV

RESULTADOS Y DISCUSIÓN

4.1. Evaluación de la calidad interna del caso de estudio

4.1.1. Descripción del caso de estudio

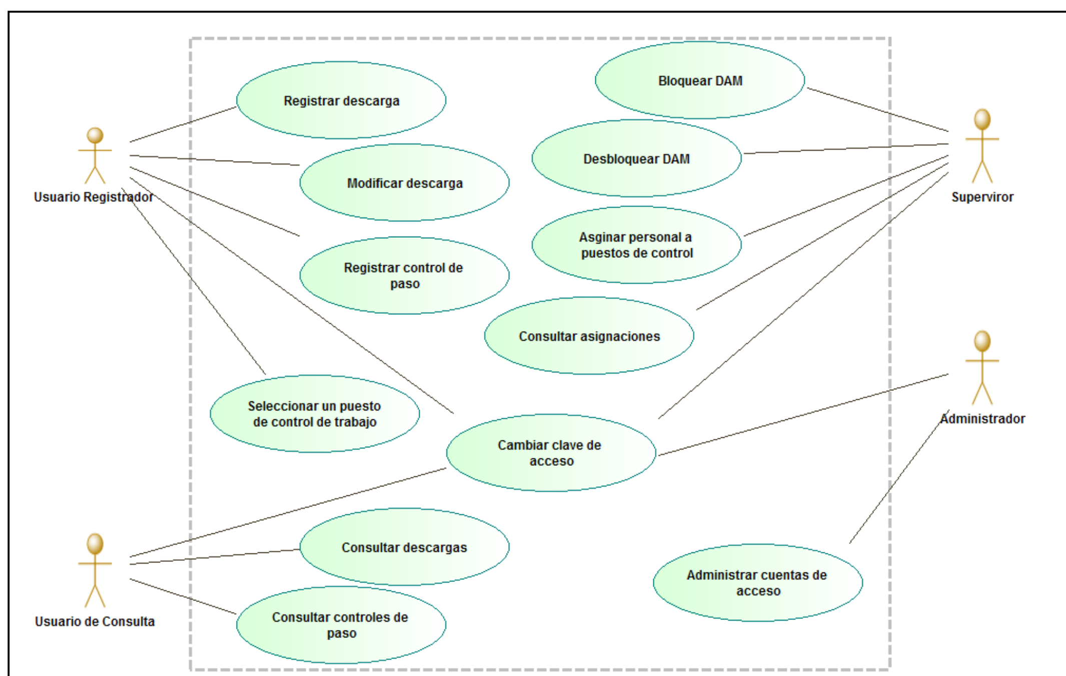
Todo Proceso de refactorización se realiza sobre el código de un sistema software y en el marco de su proyecto de desarrollo. A la hora de pensar en un refactoring, es importante conocer el código, el diseñó del sistema y su arquitectura (Alonso & Klaver, 2015).

El caso de estudio elegido para esta investigación corresponde a una Aplicación Web Java usada en una institución pública de la ciudad de Puno, el cual tiene como funcionalidad principal realizar la trazabilidad de las mercancías que son descargadas en zonas fronterizas (mercancías importadas), permitiendo tener un control de las mismas; adicionalmente, permite registrar el control de paso de los vehículos que transportan dicha mercadería, el cual se efectúa en puestos de control intermedios; además, cuenta con consultas y reportes que permiten presentar la información de las descargas de mercancías y controles a los vehículos efectuados. La Figura N° 4.1, muestra los casos de uso que implementa y los actores del sistema.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Figura N° 4.1: Diagrama de límite de sistema del caso de estudio.



Elaboración: Propia.

4.1.1.1. Funcionalidades del caso de estudio

- **Descarga de mercancías:** Posee las opciones de registrar y modificar las descargas de las mercancías importadas, llevando un control de saldos de cada importación.
- **Control de paso**: Posee la opción de registrar los controles de paso de los vehículos que trasladan la mercancía.
- **Consulta de descargas de mercancías:** Posee diversas opciones de consulta de información del proceso de descarga de mercancías.
- **Consulta de controles de paso**: Posee diversas opciones de consulta de información del proceso de control de paso.
- **Utilitarios**: Posee las opciones genéricas que permiten usar la aplicación, por ejemplo: cambio de clave de acceso entre otras.
- **Ayuda**: Presenta los manuales de usuario de la aplicación.

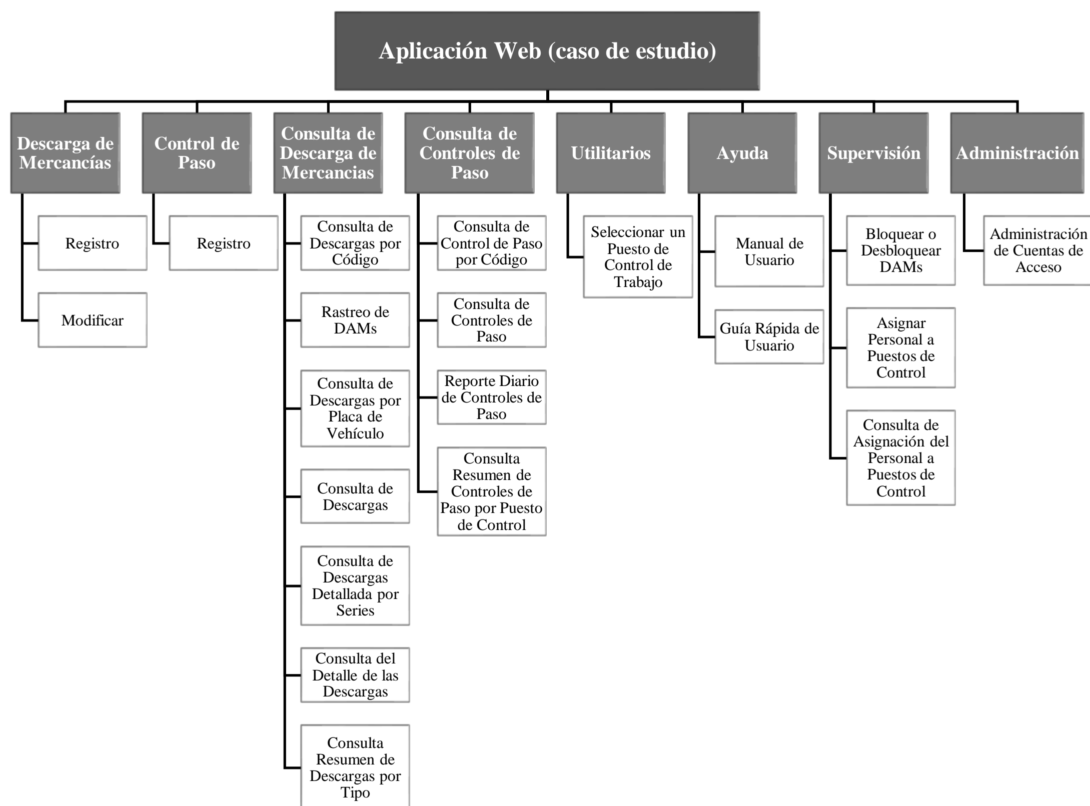
Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

- **Supervisión**: Posee opciones que solo son de acceso al usuario con perfil supervisor, por ejemplo, asignación de personal a puestos de control.
- **Administración**: Posee opciones que permiten la administración de las cuentas de acceso a la aplicación.

La Figura N° 4.2, muestra con mayor detalle las opciones que posee el caso de estudio.

Figura N° 4.2: Funcionalidades del caso de estudio.



Elaboración: Propia.

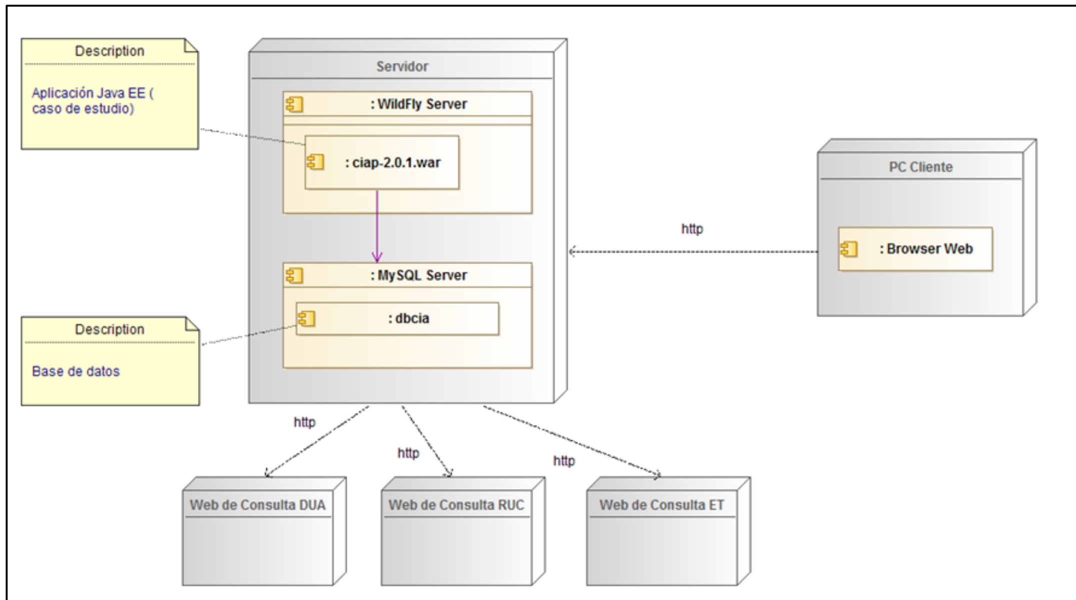
4.1.1.2. Arquitectura del caso de estudio

a. Arquitectura física del sistema

El caso de estudio está desarrollado en el lenguaje de programación Java, se ejecuta sobre un servidor de aplicaciones WildFly, y su base de datos se encuentra en el gestor MySQL Server. Adicionalmente, extrae datos de tres sitios Webs distintos. La arquitectura física de la aplicación se muestra en la Figura N° 4.3.

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

Figura N° 4.3: Arquitectura física del caso de estudio.

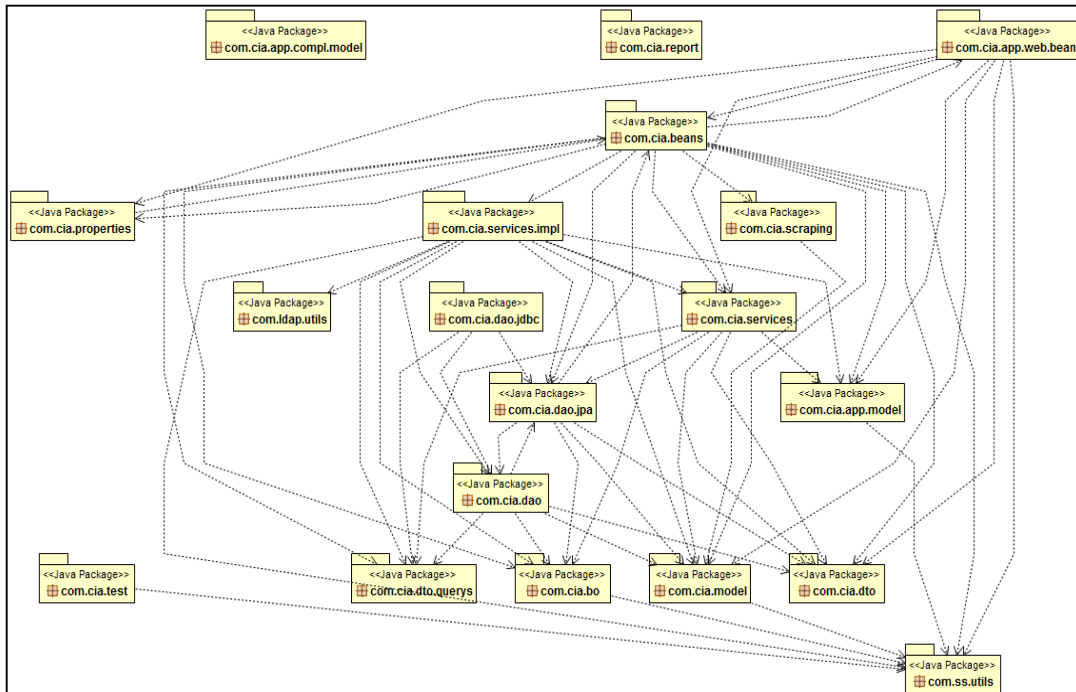


Elaboración: Propia.

b. Arquitectura lógica

La estructura de paquetes Java del caso de estudio y sus relaciones se muestran en la Figura N° 4.4.

Figura N° 4.4: Diagrama de paquetes del caso de estudio.



Elaboración: Propia.

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

En la Tabla N° 4.1, se observa la agrupación de los paquetes Java por su funcionalidad en el caso de estudio.

Tabla N° 4.1: Agrupación de paquetes por funcionalidad del caso de estudio.

Agrupación por funcionalidad	Paquete	Descripción
Modelo	com.cia.app.compl.model com.cia.app.model com.cia.bo com.cia.model com.cia.dto com.cia.dto.querys	Paquetes que agrupan las clases del modelo de dominio de la aplicación.
Servicios	com.cia.services com.cia.services.impl	Paquetes que agrupan las clases que implementan parte de la lógica de negocios.
Persistencia	com.cia.dao com.cia.dao.jdbc com.cia.dao.jpa	Paquetes que agrupan las clases de la persistencia de datos.
Presentación	com.cia.app.web.bean com.cia.beans	Paquetes que contienen las clases que corresponden a los ManagedBeans del framework JSF que usa la aplicación.
Utilerías	com.ss.utils	Paquetes que contienen las clases de utilerías.
Otros	com.cia.properties com.cia.report com.cia.scraping com.cia.test com.ldap.utils	Paquetes que contienen las clases y archivos complementarios de la aplicación.

Elaboración: Propia.

c. Tamaño del caso de estudio

El caso de estudio tiene un total de 15,571 líneas de código, conformado por 18 paquetes y 169 clases. La Tabla N° 4.2, muestra la composición del caso de estudio con mayor detalle.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Tabla N° 4.2: Tamaño inicial del caso de estudio (Aplicación Web).

Medida	Total
Líneas de código	15,571
Líneas	21,657
Sentencias	5,959
Métodos	2,452
Clases	169
Ficheros	169
Directorios (paquetes)	18
Comentarios (%)	3.2 %
Líneas de comentario	519

Elaboración: Propia.

4.1.2. Análisis de la calidad interna del caso de estudio

El análisis de la calidad interna del caso de estudio fue realizado con la herramienta SonarQube en base a las reglas de calidad que posee. La Tabla N° 4.3, muestra el resultado del análisis efectuado, considerando únicamente los archivos con extensión .java. Evidenciando un total de 918 incidencias (entre bugs, vulnerabilidades y code smells), 14.50 % de código duplicado y 0.0 % de cobertura de código; a la vez, se observa que los paquetes que concentran más del 50.33% de incidencias son: *com.cia.scraping* y *com.cia.beans*, el resto de paquetes representa el 49.67%. Adicionalmente, se aprecia que 9 paquetes tienen más del 10% de su código duplicado, siendo el 54.50% el porcentaje de código duplicado más alto que corresponde al paquete: *com.cia.dto.querys*, el que posee 82 líneas de código. La aplicación tiene un total de 15,571 líneas de código; el paquete que concentra la mayor cantidad de líneas de código es *com.cia.beans,* con un total de 5.3K líneas de código.

En resumen, se aprecia que la aplicación posee un número considerable de incidencias, alto porcentaje de código duplicado y 0% de cobertura de código.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Tabla N° 4.3: Análisis del código fuente del caso de estudio antes del proceso de refactoring.

	Aplicación Web Java (Caso de Estudio)	Líneas de código	Número de bugs	Número de vulnerabilidades	Número de code smells	Total de Incidencias	% de Incidencias	Cobertura de código (%)	Código Duplicado (%)
N° Paquete	**Totales**	**16k**	192	57	669	**918**	**100.00%**	**0.00%**	**14.50%**
1	com.cia.scraping	758	27	12	201	240	26.14%	0.00%	17.40%
2	com.cia.beans	5.3k	63	3	156	222	24.18%	0.00%	14.30%
3	com.cia.dao.jpa	1.6k	12	3	76	91	9.91%	0.00%	15.20%
4	com.cia.services.impl	1.2k	41	8	37	86	9.37%	0.00%	6.40%
5	com.cia.bo	3.6k	0	15	50	65	7.08%	0.00%	20.80%
6	com.ldap.utils	164	7	4	42	53	5.77%	0.00%	15.40%
7	com.cia.app.web.bean	724	8	4	30	42	4.58%	0.00%	6.70%
8	com.ss.utils	187	11	2	19	32	3.49%	0.00%	0.00%
9	com.cia.model	585	14	6	8	28	3.05%	0.00%	27.70%
10	com.cia.dao.jdbc	109	8	0	11	19	2.07%	0.00%	17.60%
11	com.cia.services	201	0	0	11	11	1.20%		0.00%
12	com.cia.dto	690	0	0	8	8	0.87%	0.00%	1.70%
13	com.cia.test	16	1	0	6	7	0.76%	0.00%	0.00%
14	com.cia.dao	215	0	0	6	6	0.65%		0.00%
15	com.cia.properties	5	0	0	5	5	0.54%	0.00%	0.00%
16	com.cia.dto.querys	82	0	0	2	2	0.22%	0.00%	54.50%
17	com.cia.app.compl.model	77	0	0	1	1	0.11%	0.00%	19.50%
18	com.cia.app.model	128	0	0	0	0	0.00%	0.00%	0.00%

Elaboración: Propia.

4.2. Definición del proceso de refactoring para mejorar la calidad interna de una aplicación Web Java

En principio, es necesario realizar el análisis e identificación de incidencias del código fuente del sistema para detectar errores y síntomas de problemas asociados al desarrollo y las malas prácticas de diseño. Con este diagnóstico, se pasa a la elaboración de un plan que guiará los refactorings, para posteriormente efectuar las modificaciones con el objetivo de solucionar los problemas detectados. A continuación, se realiza una evaluación de los cambios efectuados y su impacto en la calidad interna del sistema. Finalmente, se decide si conviene llevar adelante una nueva iteración de etapas para

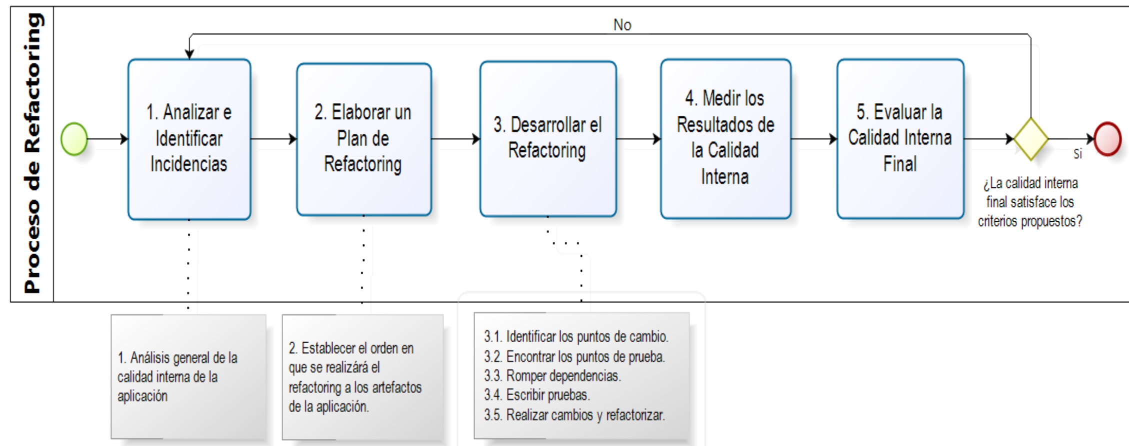
Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Universidad Nacional del Altiplano

corregir los problemas de calidad interna que pudieron persistir. El proceso de refactoring definido se muestra en la Figura N° 4.5.

Figura N° 4.5: Proceso de refactoring definido.



Elaboración: Propia.

4.2.1. Fase 1: Analizar e identificar incidencias

Esta actividad tiene como objetivo revisar la estructura interna de la aplicación con la finalidad de detectar posibles anomalías y comprender su modularización. Además, se identifican las incidencias (bugs, vulnerabilidades, code smells), código duplicado y cobertura de código de la aplicación Web, revelando los módulos, paquetes y clases que presentan anomalías, con el propósito identificar los lugares a refactorizar.

4.2.2. Fase 2: Elaborar un plan de refactoring

Identificadas las incidencias, es necesario elaborar un plan para el desarrollo de la refactorización de la aplicación Web Java. El plan elaborado, sirve como guía para realizar el refactoring del código fuente, es decir, nos indicará el orden y las actividades a realizar para la refactorización de la aplicación Web Java, lo que podría depender de la arquitectura de la aplicación.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

4.2.3. Fase 3: Desarrollar el refactoring

En esta actividad se desarrolla los refactoring teniendo en cuenta el plan elaborado en la actividad previa. Para realizar las refactorizaciones se considera emplear el algoritmo de cambio de código heredado descrito en (Feathers, 2004) el cual es:

1. *Identificar los puntos de cambio.*
2. *Encontrar los puntos de prueba.*
3. *Romper dependencias.*
4. *Escribir pruebas.*
5. *Realizar cambios y refactorizar.*

En los casos en que no sea posible escribir las pruebas por el mal diseño del código, se emplea las herramientas de refactorización que posee el Eclipse IDE. (Fowler, 1999), menciona que el objetivo principal de una herramienta de refactorización es permitir que el programador refactorice el código sin tener que volver a probar el programa. Las pruebas consumen mucho tiempo, incluso cuando están automatizadas, y eliminarlas puede acelerar el proceso de refactorización por un factor significativo.

4.2.4. Fase 4: Medir los resultados de la calidad interna

Concluido el desarrollo de los refactorings, se realiza una nueva medición de la calidad interna del caso de estudio, nuevamente se emplea una herramienta de análisis de código para obtener los nuevos resultados para su evaluación.

4.2.5. Fase 5: Evaluar la calidad interna final

El proceso finaliza con la evaluación final de la calidad interna, determinando la medida en que el proceso afectó la calidad interna del producto, y evaluar si la calidad interna lograda satisface con los criterios planteados inicialmente. De acuerdo con los resultados obtenidos, podría tomarse la decisión si conviene llevar adelante una nueva iteración de fases para corregir los problemas de calidad interna que persistieron.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

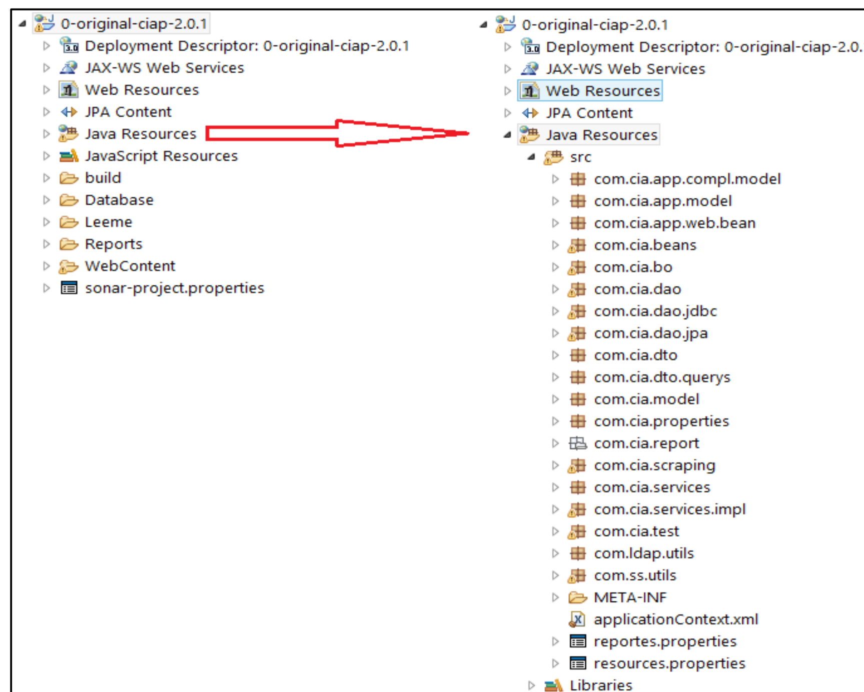
4.3. Desarrollo del proceso de refactoring sobre el caso de estudio

4.3.1. Fase 1: Analizar e identificar incidencias.

4.3.1.1. Estructura de la Aplicación

Como primera etapa se revisó la estructura interna del proyecto de la aplicación Web Java. Examinando los paquetes Java que conforman el caso de estudio se aprecia que su estructura lógica presenta una arquitectura en capas, conformada por una capa de lógica de negocio, una capa de persistencia y una capa de presentación. Sin embargo, la Figura N° 4.6, advierte que su estructura no es clara y los paquetes que la conforman se encuentran agrupados en un solo proyecto de Eclipse IDE, evidenciando la falta de modularización.

Figura N° 4.6: Estructura inicial del proyecto del caso de estudio en Eclipse IDE.



Elaboración: Propia.

4.3.1.2. Identificación de incidencias en el código fuente

La identificación de incidencias en el código fuente del caso de estudio se efectuó con la herramienta SonarQube; el resultado del análisis se presentó en la Tabla N° 4.3. El resumen de las incidencias halladas en el caso de estudio se muestra en la

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

Tabla N° 4.4, adicionalmente, se presenta la cantidad de líneas de código, la cobertura del código y el porcentaje del código duplicado que posee antes del refactoring.

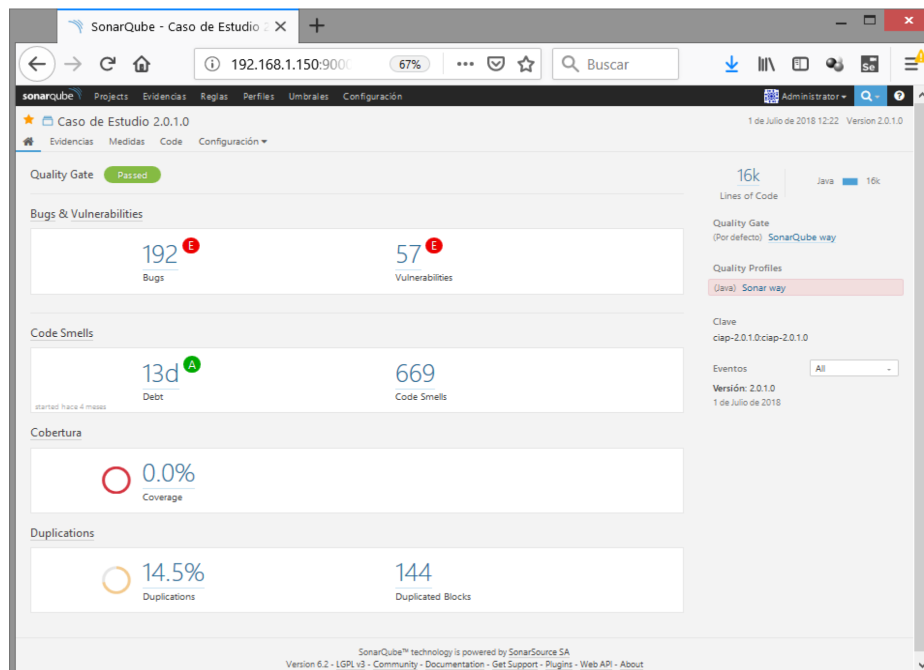
Tabla N° 4.4: Resumen del análisis del caso de estudio.

Medida	Cantidad Total
Líneas de código	15, 571
Número de bugs	192
Número de vulnerabilidades	57
Número de code smells	669
Cobertura de código	0.0 %
Código duplicado	14.50 %

Elaboración: Propia.

La Figura N° 4.7, muestra el resultado de la medición del número de bugs, vulnerabilidades, code smells, cobertura y código duplicado del caso de estudio antes del proceso de refactoring obtenido con el software SonarQube.

Figura N° 4.7: Medición de la calidad interna inicial del caso de estudio, obtenida con el software SonarQube antes del proceso de refactoring.



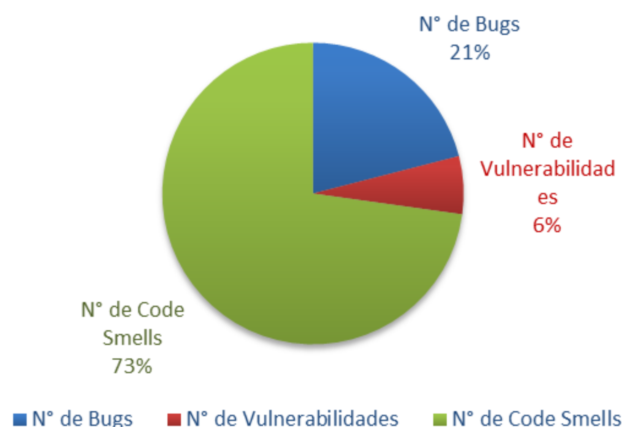
Elaboración: Propia.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

La Figura N° 4.8, muestra que el 21% de incidencias corresponde a bugs, el 6% a vulnerabilidades, y el porcentaje más alto corresponde a code smells siendo este el 73%.

Figura N° 4.8: Porcentaje inicial de incidencias del caso de estudio.



Elaboración: Propia.

En el Anexo F, se muestran diferentes reportes de SonarQube, que permiten ver con mayor detalle el análisis inicial del caso de estudio.

4.3.2. Fase 2: Elaborar un plan de refactoring

El objetivo es mejorar la calidad interna del caso de estudio, por tanto, se plantea efectuar el refactoring en etapas, que permitan mejorar la estructura del proyecto, y también, mejorar los módulos o capas que conforman su arquitectura.

4.3.2.1. Etapas

El desarrollo del refactoring, se lleva a cabo en base a dos etapas, las que tienen un conjunto de actividades a desarrollar. A continuación, se detallan estas etapas.

a. Etapa 1: Refactoring de la estructura del proyecto

La etapa inicial consiste en reestructurar el proyecto en Eclipse IDE. También, se considera la modularización del proyecto según las capas que definen la arquitectura del caso de estudio.

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

b. Etapa 2: Refactoring de las capas de la aplicación

La segunda etapa, consiste en realizar el refactoring de las capas o módulos que conforman el caso de estudio.

4.3.2.2. Actividades

La Tabla N° 4.5, muestra el detalle de las actividades de cada etapa.

Tabla N° 4.5: Plan de las etapas para desarrollar el refactoring.

Etapas / Actividades
1. Refactoring de la estructura del proyecto
1.1. Identificar los puntos de cambio.
1.2. Encontrar los puntos de prueba.
1.3. Romper dependencias.
1.4. Escribir pruebas.
1.5. Realizar cambios y refactorizar.
1.5.1. Restructuración de los paquetes Java.
1.5.2. Modularización del Proyecto.
2. Refactoring de las capas de la aplicación
2.1. Refactoring de la capa de lógica de negocios (módulo 1)
2.1.1. Identificar los puntos de cambio.
2.1.2. Encontrar los puntos de prueba.
2.1.3. Romper dependencias.
2.1.4. Escribir pruebas.
2.1.5. Realizar cambios y refactorizar
a. Refactoring usando DRY.
b. Refactoring usando S.O.L.I.D.
2.2. Refactoring de la capa de lógica de negocios (módulo 2)
2.2.1. Identificar los puntos de cambio.
2.2.2. Encontrar los puntos de prueba.
2.2.3. Romper dependencias.
2.2.4. Escribir pruebas.
2.2.5. Realizar cambios y refactorizar
a. Refactoring usando DRY.
b. Refactoring usando S.O.L.I.D.
2.3. Refactoring de la capa de persistencia
2.3.1. Identificar los puntos de cambio.
2.3.2. Encontrar los puntos de prueba.
2.3.3. Romper dependencias.
2.3.4. Escribir pruebas.
2.3.5. Realizar cambios y refactorizar
a. Refactoring usando DRY.
b. Refactoring usando S.O.L.I.D.
2.4. Refactoring de la capa de presentación
2.4.1. Identificar los puntos de cambio.
2.4.2. Encontrar los puntos de prueba.
2.4.3. Romper dependencias.
2.4.4. Escribir pruebas.
2.4.5. Realizar cambios y refactorizar
a. Refactoring usando DRY.
b. Refactoring usando S.O.L.I.D.

Elaboración: Propia.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

4.3.3. Fase 3: Desarrollar el refactoring

4.3.3.1. Refactoring de la estructura del proyecto

a. Identificar los puntos de cambio

(Microsoft, 2009), precisa que la arquitectura en capas se centra en la agrupación de funcionalidades relacionadas dentro de una aplicación en distintas capas que se apilan verticalmente una encima de la otra. La funcionalidad dentro de cada capa está relacionada por un rol o responsabilidad común. La comunicación entre capas es explícita y está ligeramente acoplada. Estructurar su aplicación apropiadamente ayuda a una fuerte separación de responsabilidades que, a su vez, admite flexibilidad y facilidad de mantenimiento.

La composición del caso de estudio se explica a continuación:

- **Capa de persistencia**: Es la capa inferior que se encarga de persistir los datos en la base de datos. El caso de estudio usa JPA/Hibernate para realizar esta tarea.
- **Capa de lógica de negocios**: Es la capa intermedia, donde tiene implementado la lógica de negocios. El caso de estudio se apoya en el framework Spring en esta capa.
- **Capa de presentación**: Es la capa de la interface de usuario. El caso de estudio emplea el framework JSF. Esta capa contiene páginas XHTML, XML, archivos JavaScript, CSS e imágenes. También, contiene código Java que corresponde a los ManagedBeans necesarios en aplicaciones JSF.

Anomalías que motivan el refactoring

- El proyecto tiene una arquitectura por capas, pero no se encuentra modularizado, trayendo desventajas en la manejabilidad del proyecto y menor productividad por que limita el trabajo en equipo.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

- Todos los paquetes se encuentran en un solo proyecto, incluido paquetes de utilerías, que podrían ser extraídos y reutilizados en otros proyectos.
- El manejo de librerías podría convertirse en un problema, no se tiene certeza de las librerías que usa el proyecto.

Principios por considerar

En (Microsoft, 2009) encontramos que entre los principios clave para tener un buen diseño se encuentran:

Principios de Diseño	Descripción
SRP	Cada componente o módulo debe ser responsable de una característica o funcionalidad específica, o agregación de funcionalidad cohesiva (Microsoft, 2009).
DRY	Solo debe especificar el propósito en un solo lugar. Por ejemplo, en términos de diseño de la aplicación, se debe implementar una funcionalidad específica en un solo componente; la funcionalidad no debe duplicarse en ningún otro componente (Microsoft, 2009).

Ambos, principios son considerados en el refactoring de la estructura del proyecto.

b. Encontrar los puntos de prueba

El refactoring de la estructura del proyecto afectará su organización en general, por lo que el éxito de esta actividad se validó con los errores que detecte el Eclipse IDE. No fue necesario escribir pruebas de cobertura, puesto que no se altera el comportamiento de la aplicación.

c. Romper dependencias

Al no tener pruebas no fue necesario romper las dependencias.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Universidad Nacional del Altiplano

d. Escribir pruebas

No se escribieron pruebas de cobertura. Se comprueba el éxito de esta actividad con la compilación y ejecución exitosa de la aplicación, además, se comprueba la creación de los módulos definidos.

e. Realizar cambios y refactorizar

e.1. Restructuración de los paquetes Java

¿Cómo debe estar organizada la estructura de paquetes de una aplicación Web?, (Marinescu, 2002), menciona que una solución obvia es sugerida por la estratificación de su arquitectura: cree un paquete por capa, más un par de paquetes adicionales para el código independiente de la capa. Esto da como resultado un conjunto de paquetes como el siguiente:

```
com.mycompany.mysystem.domain
com.mycompany.mysystem.exceptions
com.mycompany.mysystem.persistence
com.mycompany.mysystem.services
com.mycompany.mysystem.util
com.mycompany.mysystem.web (asumiendo una capa de aplicación basada en web).
```

A esta organización de paquetes se agregó las funcionalidades del caso de estudio, quedando su estructura final de la siguiente forma:

```
com.[SISTEMA].[CAPA].[TIPO].[CARACTERÍSTICA]
```

Adicionalmente, y según lo sugerido por (Marinescu, 2002), se agregó algunos paquetes independientes a cada capa, por ejemplo el paquete: com.[SISTEMA].common. Al refactorizar la estructura de paquetes del caso de estudio según la estructura definida, se obtuvo el nuevo conjunto de paquetes indicado en la Tabla N° 4.6.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Tabla N° 4.6: Estructura de paquetes del caso de estudio después del proceso de refactoring.

N°	Módulo	Paquete	N°	Módulo	Paquete
1	cia-business-model	com.cia.business.model.asignacion	33	cia-persistence	com.cia.persistence.asignacion.dao
2		com.cia.business.model.asignacion.dto	34		com.cia.persistence.asignacion.dao.jpa
3		com.cia.business.model.common	35		com.cia.persistence.common.dao
4		com.cia.business.model.common.dto	36		com.cia.persistence.common.dao.jpa
5		com.cia.business.model.common.validator	37		com.cia.persistence.controlpaso.dao
6		com.cia.business.model.controlpaso	38		com.cia.persistence.controlpaso.dao.jdbc
7		com.cia.business.model.controlpaso.dto	39		com.cia.persistence.controlpaso.dao.jpa
8		com.cia.business.model.controlpaso.reporte	40		com.cia.persistence.poliza.dao
9		com.cia.business.model.controlpaso.validators	41		com.cia.persistence.poliza.dao.jpa
10		com.cia.business.model.dto.generic	42		com.cia.persistence.retiro.dao
11		com.cia.business.model.dto.querys	43		com.cia.persistence.retiro.dao.jdbc
12		com.cia.business.model.dto.querys.util	44		com.cia.persistence.retiro.dao.jpa
13		com.cia.business.model.poliza	45	cia-web	com.cia.presentation.asignacion.controller
14		com.cia.business.model.retiro	46		com.cia.presentation.common
15		com.cia.business.model.retiro.builders	47		com.cia.presentation.common.controller
16		com.cia.business.model.retiro.dto	48		com.cia.presentation.controlpaso.controller
17		com.cia.business.model.retiro.policys	49		com.cia.presentation.log4j.servlets
18		com.cia.business.model.retiro.validators	50		com.cia.presentation.poliza.controller
19		com.cia.business.model.seguridad	51		com.cia.presentation.retiro.controller
20		com.cia.business.model.validators	52		com.cia.presentation.seguridad.controller
21	cia-business-service	com.cia.business.service.asignacion	53		com.cia.presentation.util
22		com.cia.business.service.asignacion.impl	54	cia-common	com.cia.common
23		com.cia.business.service.common	55		com.cia.common.exception
24		com.cia.business.service.common.impl	56		com.cia.common.util
25		com.cia.business.service.controlpaso			
26		com.cia.business.service.controlpaso.impl			
27		com.cia.business.service.poliza			
28		com.cia.business.service.poliza.impl			
29		com.cia.business.service.retiro			
30		com.cia.business.service.retiro.impl			
31		com.cia.business.service.util.webscraping			
32		com.cia.business.service.util.webscraping.dam			

Elaboración: Propia.

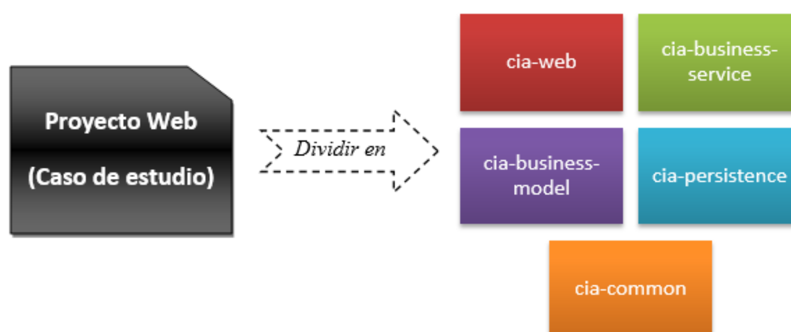
e.2. Modularización del proyecto

El principio SRP menciona que cada módulo debe ser responsable de una funcionalidad específica. Considerando la arquitectura en capas del caso de estudio se buscó cumplir con el principio SRP; por tanto, según las responsabilidades de cada capa se dividió el proyecto en módulos más pequeños que implementen una única responsabilidad en el caso de estudio; la idea de esta división en módulos se aprecia en la Figura N° 4.9.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

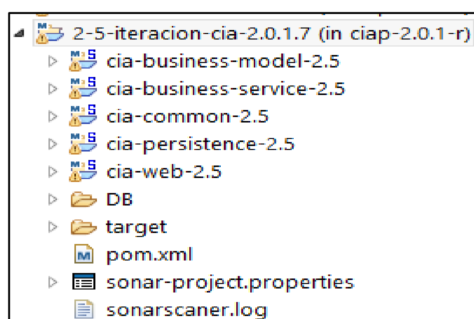
Figura N° 4.9: División del proyecto Web en múltiples módulos Maven.



Elaboración: Propia.

La modularización, gestión de librerías y sus dependencias se realizó con Maven, creando un proyecto principal con cinco módulos; además, se agregó las dependencias requeridas por el caso de estudio. El proyecto y los módulos Maven obtenidos en Eclipse IDE luego de la modularización del proyecto se aprecian en la Figura N° 4.10. Así mismo, la Figura N° 4.11, muestra el diagrama de arquitectura alcanzado, resaltando que la capa de la lógica de negocios fue separada en dos componentes: 1) *cia-business-model*, y 2) *cia-business-service*.

Figura N° 4.10: Proyecto y módulos Maven después del refactoring en Eclipse IDE.

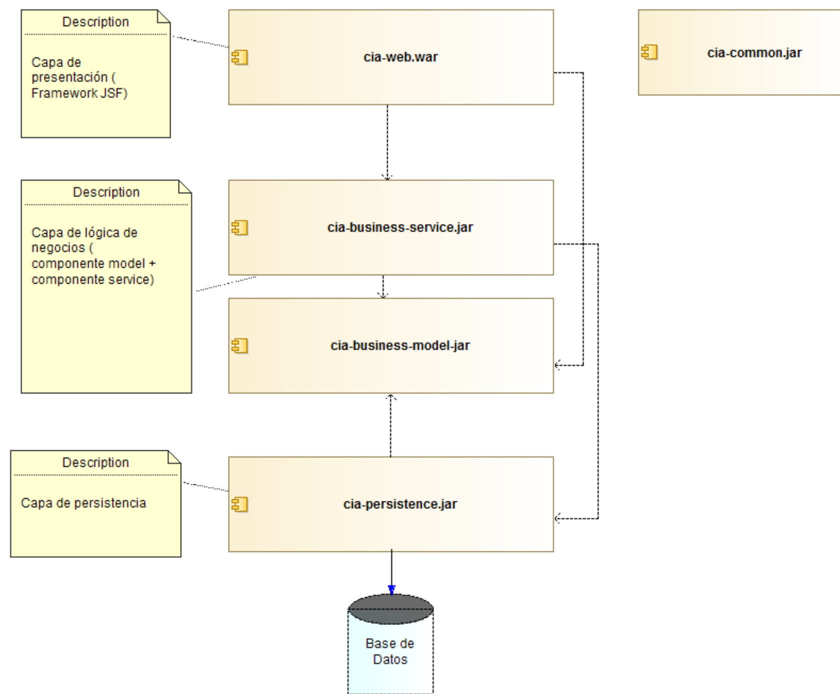


Elaboración: Propia.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Figura N° 4.11: Arquitectura del caso de estudio refactorizado.

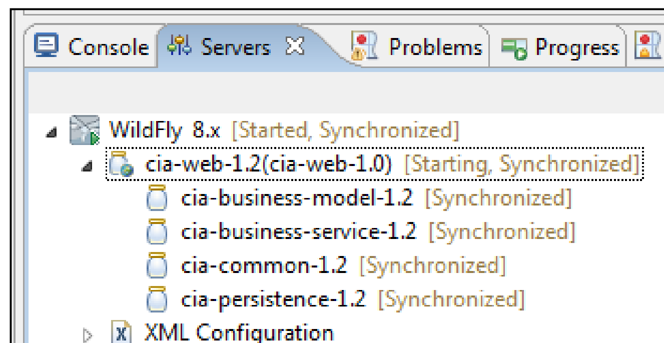


Elaboración: Propia.

Criterio de éxito

Esta etapa consistió en organizar la estructura de los paquetes y del proyecto. Para verificar que no se introdujeron errores en el caso de estudio durante las refactorizaciones, se compiló la aplicación completa; además, se comprobó la generación de los componentes que se muestran en la Figura N° 4.12 luego de desplegar la aplicación en el servidor WildFly.

Figura N° 4.12: Componentes generados del caso de estudio después del refactoring.



Elaboración: Propia.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Dificultades encontradas

Durante esta etapa se encontró algunas clases duplicadas en paquetes distintos, por ejemplo, la clase: "*Login*". Para conservar la funcionalidad existente, se conservaron estas clases en paquetes diferentes. El refactoring de estas clases se realizó en actividades posteriores.

4.3.3.2. Refactoring de la capa de lógica de negocios (módulo 1)

Este módulo (*cia-business-model*) contiene las clases del modelo de dominio[17] y es parte de la capa de lógica de negocios. Depende del módulo: *cia-common.*

a. Identificar los puntos de cambio

La Tabla N° 4.7, muestra el análisis de este módulo antes de su refactoring. Este análisis sirvió como guía para identificar que paquetes deben ser revisados para su refactoring tomando en cuenta el número de incidencias y código duplicado que poseen. Para un mejor análisis, también se obtuvo el detalle de incidencias por clase, el cual se muestra en la Tabla N° E.1 del Anexo E.

[17] Un modelo de dominio es una representación visual de las clases conceptuales u objetos del mundo real en un dominio de interés (Larman, 2003).

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Tabla N° 4.7: Análisis de la capa de lógica de negocios del caso de estudio antes de su refactoring (módulo 1).

Aplicación Web (Caso de Estudio - Iteracion 3)		Líneas de código	N° de bugs	N° de Vulnerabilities	N° de Code smells	Total de Incidencias	Cobertura de código (%)	Código duplicado (%)
Módulo	**Paquete**							
cia-business-model	com.cia.business.model.poliza	1.8k	14	16	26	**56**	53.90%	42.90%
	com.cia.business.model.retiro	924	0	4	7	**11**	33.00%	3.50%
	com.cia.business.model.common	1k	0	1	9	**10**	32.00%	9.60%
	com.cia.business.model.dto	663	0	0	8	**8**	17.80%	1.80%
	com.cia.business.model.dto.querys	82	0	0	2	**2**	0.00%	54.50%
	com.cia.business.model.controlpas(	215	0	0	1	**1**	50.50%	7.70%
	com.cia.business.model	128	0	0	0	**0**	83.60%	0.00%

Elaboración: Propia.

b. Encontrar los puntos de prueba

Es importante precisar que las otras capas (o módulos) dependen de este módulo, y cualquier cambio que se realice sobre este, afectará al resto de módulos; por esta razón, es necesario que la aplicación cuente con un set de pruebas en sus diferentes módulos que garanticen que su comportamiento inicial no se altere luego de los refactorings. Por este motivo, en esta etapa se escribieron las pruebas unitarias y de integración que ejercitan el comportamiento de todos los módulos que conforman el caso de estudio, priorizando las clases que presentan un mayor número de incidencias o tengan mayor porcentaje de código duplicado.

c. Romper dependencias

En la arquitectura n capas, las dependencias van desde la capa inferior a las capas superiores, por tanto, para escribir las pruebas unitarias de las clases y romper sus

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

TESIS UNA - PUNO

Universidad Nacional del Altiplano

dependencias se usó objetos Mocks[18]. Se utilizó Mockito y PowerMock para la creación de dobles, con los que se consiguió la creación de las pruebas unitarias.

d. Escribir pruebas

Se escribieron pruebas unitarias y de integración (según corresponda) en el orden siguiente: 1) *cia-business-model*, 2) *cia-business-service*, 3) *cia-persistence* y 4) *cia-web*; el orden se determinó por la responsabilidad de cada módulo. A continuación, se describe brevemente las acciones realizadas en la escritura de las pruebas unitarias y de integración para los módulos del caso de estudio:

- Se identificaron las clases que contienen comportamiento y se crearon sus respectivas clases de prueba.
- Para el módulo *cia-business-model,* se escribieron pruebas unitarias, y no fue necesario romper ninguna dependencia.
- Para el módulo *cia-business*-services, se escribieron pruebas unitarias y de integración; para las pruebas unitarias se usó Mockito para romper las dependencias con la capa de persistencia, y para las pruebas de integración con la capa de persistencia se usó Spring JUnit.
- Para el módulo de persistencia, se crearon pruebas de integración con la base de datos, y fue necesario usar Spring JUnit; además, se utilizó la anotación *@Rollback* para no alterar la base de datos.
- Para el módulo de presentación, se usó la herramienta PowerMock con la finalidad de crear dobles para las clases propias del framework JSF, y para no tener ningún inconveniente con los métodos estáticos de este módulo.

[18] objetos simulados, objetos que imitan el comportamiento de objetos reales de una forma controlada.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

- En general se omitió escribir pruebas para los métodos *setters* y *getters*, *equals* y *hashCode*.
- Por último, se ejecutaron todas las pruebas escritas y se comprobó que todas pasen.

Al finalizar la escritura de las pruebas unitarias y de integración, la cobertura alcanzada en esta etapa fue de 27%; sin embargo, al concluir todo el proceso de refactoring la cobertura alcanzada fue de 34.1%. En el Anexo D, se presentan algunos ejemplos de las pruebas escritas.

e. Realizar cambios y refactorizar

e.1. Refactoring usando DRY

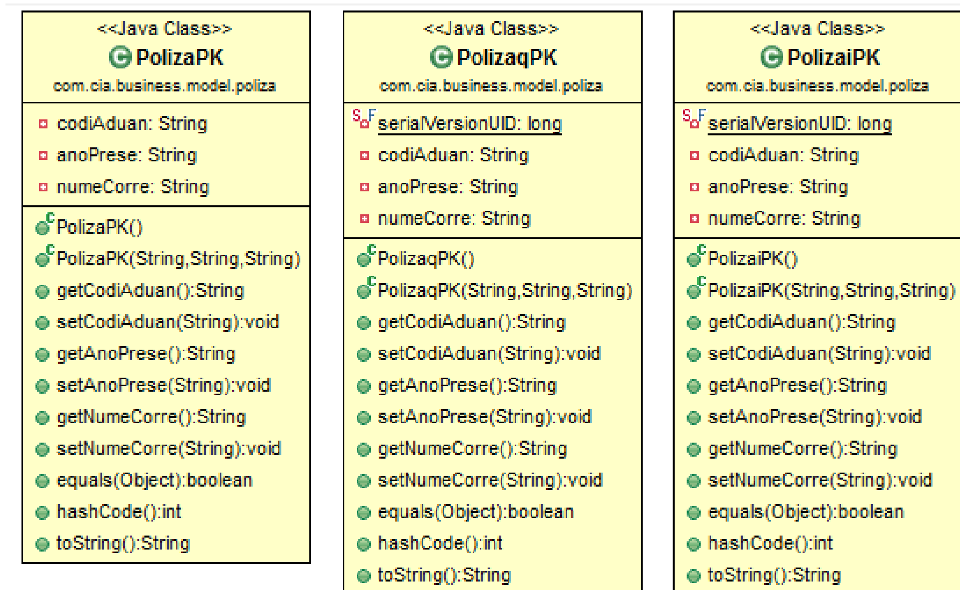
Se revisó las clases de los paquetes mostrados en la Tabla N° 4.7, que tenían el mayor porcentaje de código duplicado, y se buscó que cumplan con el principio DRY. A continuación, se describe brevemente las acciones ejecutadas en los refactoring desarrollados:

- Se revisaron las clases que declaraban atributos y métodos idénticos –clases duplicadas con distinto nombre-, y se comprobó que se trataba de código duplicado, incumpliendo con DRY. Un ejemplo de estas clases se aprecia en la Figura N° 4.13. El refactoring de esas clases consistió en eliminar las clases duplicadas, conservando una única clase que tenga la responsabilidad de implementar un único concepto del modelo de dominio.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Figura N° 4.13: Ejemplo de clases duplicadas en la capa de lógica de negocios antes del refactoring.



Elaboración: Propia.

- Se analizó el código duplicado de las clases que implementaban conceptos similares. Por ejemplo, las clases *Polizai* y *Polizaq* que implementan los conceptos de importaciones para el consumo e importación simplificada, las que son dos tipos de importaciones que implementa el caso de estudio. El refactoring de esas clases consistió en usar la herencia y especialización de clases, consiguiendo eliminar el comportamiento duplicado, los atributos repetidos y sus métodos *setters y getters* correspondientes. En algunos casos se adicionó anotaciones JPA para que las clases sean tratadas como entidades.
- Parte del porcentaje de código duplicado se debía a que algunas clases tenían atributos comunes, por ejemplo, los atributos: *usureg*, *usumod*, *fecmod*, *fecreg*, *hostmod* y *hostreg*. El refactoring de este código duplicado consistió en extraer estos atributos a una nueva clase, la que se extendió por las clases que usan estos atributos de forma repetida.

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

- Se identificó que este módulo contenía clases con nombres iguales y con atributos y operaciones duplicadas, Por ejemplo, la clase: *Login*, que se encontraba duplicada en dos paquetes diferentes. El refactoring de estas clases consistió en eliminar las clases duplicadas. Adicionalmente, se tuvo que corregir las referencias rotas en los módulos de persistencia, servicios y presentación que hacían referencia a las clases eliminadas.
- Se analizó el código duplicado que se encontraba en las clases que implementaban el patrón DTO[19] -clases que implementan consultas resúmenes en el caso de estudio-. El refactoring de estas clases se efectuó extrayendo los atributos duplicados en una clase base, la que fue extendida por las clases DTOs que lo requerían.

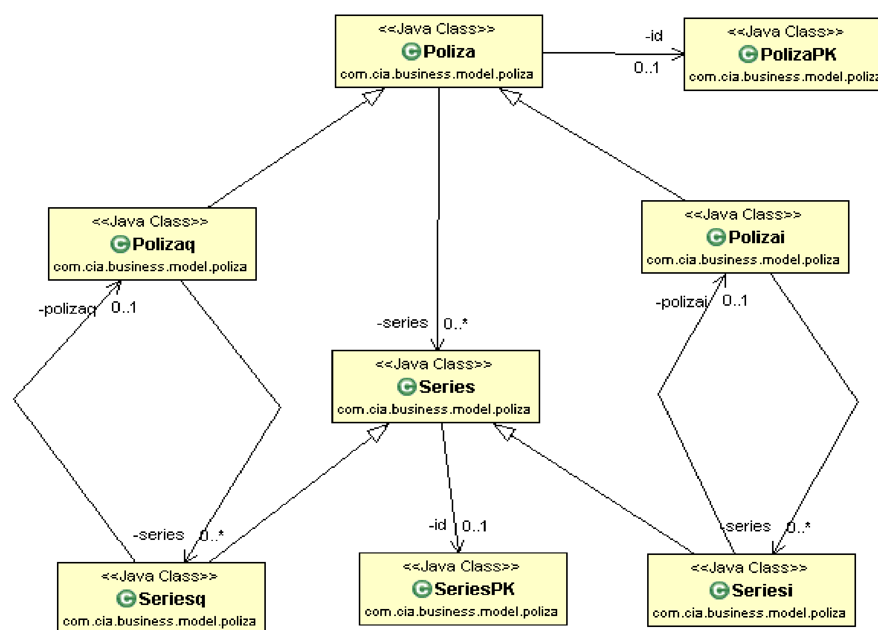
La Figura N° 4.14, muestra parte del diseño de clases obtenido después del refactoring realizado, el cual buscó cumplir con el principio DRY implementando relaciones de herencia.

[19] Objeto de Transferencia de Datos

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Figura N° 4.14: Ejemplo de herencia aplicado en el refactoring de la capa de lógica de negocios.



Elaboración: Propia.

e.2. Refactoring usando SOLID

Se examinó las partes de código que incumplan con alguno de los principios SOLID, y por medio de diversos refactorings se buscó que las cumplan. A continuación, se describen las acciones realizadas:

- El principio SRP, indica que cada clase debe tener una única responsabilidad; las clases que incumplen este principio son generalmente las clases con un volumen elevado de código que generalmente agrupan más de una responsabilidad. Por tanto, se analizó las clases con mayor número de líneas de código de la Tabla N° E.1 del Anexo E.
- Algunas clases, además, de contener los atributos del modelo de dominio, también contenían atributos constantes que se usan en la aplicación. Estas clases incumplen el principio SRP, puesto que además de implementar el modelo de dominio, también tienen la responsabilidad de contener constantes que se usan en partes diferentes de toda la aplicación. El refactoring de estas clases para cumplir con el SRP consistió

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

en extraer los atributos constantes a las clases nuevas *Constantes* y *ConstantesJPA*, ambas clases se ubican en el módulo *common*.

- Se encontró que algunas clases implementaban más de un concepto del modelo de dominio, por tanto, incumplían con el principio SRP. Por ejemplo, la clase *Retiro*, tenía el mayor número de líneas de código, con un total de 460 líneas, poseía atributos correspondientes a conceptos de Conductor, Empresa de transporte, Remitente, Destinatario y Vehículo. El refactoring de esta clase y clases similares para conseguir que cumplan con SRP consistió en extraer estos atributos a clases que implementen un solo concepto del modelo de dominio; por ejemplo, se crearon las clases: *Remitente, Destinatario, Vehículo,* etc.; esta labor se efectuó con la herramienta de refactoring “Extract Class” de Eclipse IDE. La Figura N° 4.16, muestra el resultado obtenido.
- Algunas clases ocultaban una relación de herencia, como es el caso de la clase *Retiro* que implementa las responsabilidades de realizar descargas y traslados de mercancías (dos funcionalidades del caso de estudio), ocultando de este modo más de una responsabilidad, para corregirlo se refactorizó esta clase creando dos clases *Descarga* y *Traslado* que hereden de la clase *Retiro*.
- Parte del refactoring consistió en utilizar el patrón de diseño Builder[20] para abordar el problema de la construcción de objetos completos según las reglas de negocio especificadas en el caso de estudio.

---

[20] Builder (constructor), separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones. (Gamma, Helm, Johnson, & Vlissides, 2003)

Repositorio Institucional UNA-PUNO

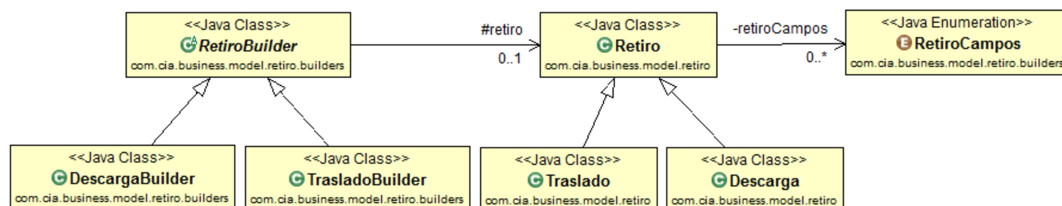
No olvide citar esta tesis

Universidad Nacional del Altiplano

- Respecto al principio LSP, se examinó las relaciones de herencia implementadas en este módulo. Se encontró que las clases que implementaban relaciones de herencia no modifican el comportamiento de sus clases base, por tanto, no se encontró alguna violación de a este principio.
- En el refactoring de este módulo, no fue necesario aplicar los principios OCP, ISP y DIP.
- De forma general, las clases y métodos que no son referenciados en todo el proyecto fueron eliminados. También, se eliminó el código sin uso, los comentarios o código comentado que son irrelevantes en este módulo.

La Figura N° 4.15 y Figura N° 4.16, muestran parte del diagrama de clases final obtenido después de desarrollar los refactorings, que buscaron remediar el incumplimiento de alguno de los principios SOLID en este módulo.

Figura N° 4.15: Uso del patrón de diseño builder en el refactoring de la capa de lógica de negocios.



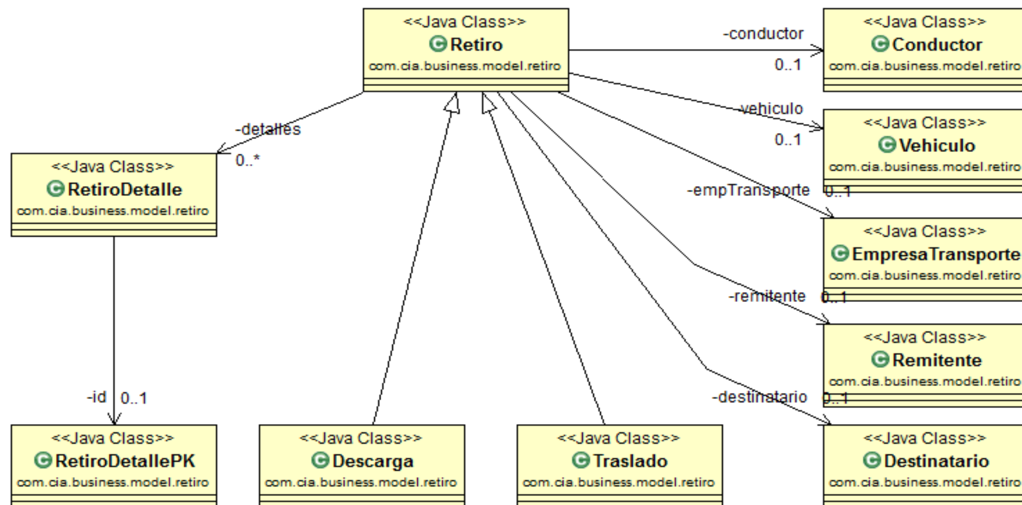
Elaboración: Propia.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Universidad Nacional del Altiplano

Figura N° 4.16: Fragmento de diagrama de clases obtenido después del refactoring de la capa de lógica de negocios.



Elaboración: Propia.

Criterios de éxito

Al finalizar el refactoring, se comprueba que todas las pruebas se ejecuten y pasen correctamente.

4.3.3.3. Refactoring de la capa de lógica de negocios (módulo 2)

Este módulo (*cia-business-service*), implementa los casos de uso de la aplicación y es parte de la capa de lógica de negocios. Utiliza el patrón Facade[21], usa el Framework Spring para aprovechar la inyección de dependencias lo que permite cumplir con el principio DIP de SOLID. Depende de los módulos: *cia-persistence*, *cia-buisness-model* y *cia-common*.

[21] (Fachada), proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar (Gamma, Helm, Johnson, & Vlissides, 2003).

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

a. Identificar los puntos de cambio

Los puntos de cambio se identificaron hallando las incidencias de los paquetes y clases de este módulo. Se realizó un nuevo análisis con SonarQube y los resultados para este módulo antes de su refactoring se muestran en la Tabla N° 4.8. El detalle de las incidencias de sus clases se aprecia en la Tabla N° E.2 del Anexo E.

Tabla N° 4.8: Análisis de la capa de lógica de negocios del caso de estudio antes de su refactoring (módulo 2).

Aplicación Web (Caso de Estudio - Iteracion 4)		Líneas de código	N° de bugs	N° de Vulnerabilities	N° de Code smells	Total de Incidencias	Cobertura de código (%)	Código duplicado (%)
Módulo	**Paquete**							
cia-business-service	com.cia.business.service.util.webscraping	375	11	4	79	**94**	57.30%	**0.00%**
	com.cia.business.service.retiro.impl	294	8	3	12	**23**	32.70%	**0.00%**
	com.cia.business.service.common.impl	371	10	2	8	**20**	33.70%	**0.00%**
	com.cia.business.service.poliza.impl	141	4	2	3	**9**	61.40%	**0.00%**
	com.cia.business.service.retiro	52	0	0	9	**9**		**0.00%**
	com.cia.business.service.controlpaso	6	0	0	2	**2**		**0.00%**
	com.cia.business.service.common	73	0	0	1	**1**		**0.00%**
	com.cia.business.service.controlpaso.impl	22	0	0	0	**0**	66.70%	**0.00%**
	com.cia.business.service.poliza	30	0	0	0	**0**		**0.00%**

Elaboración: Propia.

b. Encontrar los puntos de prueba

Los puntos de prueba se encuentran en las clases y métodos que presentan anomalías o síntomas de un mal diseño.

c. Romper dependencias

Este módulo depende de la capa de persistencia, por tanto, fue necesario romper esa dependencia con el uso de dobles y empleando el framework Mockito.

d. Escribir pruebas

En una etapa previa se consiguió cubrir el código con un set de pruebas unitarias y de integración, para el refactoring de este módulo no fue necesario escribir nuevas

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

pruebas. La Figura N° D.2 y Figura N° D.3 del Anexo D, muestran un ejemplo de las pruebas escritas en este módulo.

e. Realizar cambios y refactorizar

e.1. Refactoring usando DRY

En la Tabla N° 4.8, se aprecia que SonarQube no detecta código duplicado en los paquetes de este módulo, por tanto, no incumple el principio DRY y no se realizó algún refactoring usando este principio.

e.2. Refactoring usando SOLID

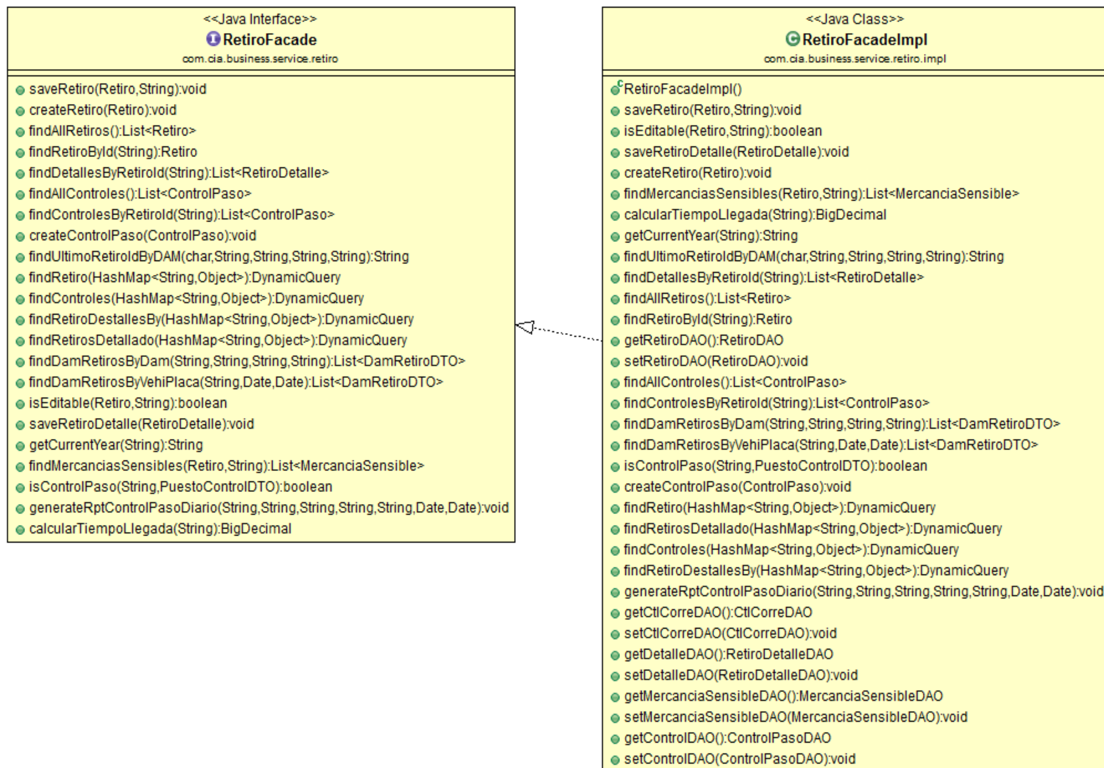
- Se revisaron las clases que contienen el mayor número de incidencias en busca del incumplimiento de algún principio SOLID.
- Se revisaron las clases con un número elevado de responsabilidades (clases con un número elevado de métodos). Por ejemplo, la Figura N° 4.17, muestra la clase *RetiroFacadeImpl*, en la que se aprecia 32 métodos públicos, esta clase implementa las funcionalidades del proceso de descarga y traslado de mercancía; además, se aprecia que tiene las responsabilidades de obtener el año actual, y funcionalidades correspondientes a controles de paso (crear, buscar, generar reportes, calcular tiempo de llegada del vehículo, etc.); por tanto, cuando la clase *RetiroFacadeImpl* tenga que admitir una nueva funcionalidad que corresponda a control de paso, tendremos que abrirla para realizar modificaciones; también, debe cambiar si tenemos que modificar la forma de obtener el año actual (por ejemplo, que no sea desde el servidor de base de datos, sino desde el servidor de aplicaciones). Estos motivos de cambio significan que la clase *RetiroFacadeImpl* incumple con SRP y OCP. Adicionalmente, se observa que la interface de esta clase incumple con el principio ISP por tener métodos poco cohesivos, obligando a los clientes a implementar métodos que probablemente no requieran.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Universidad Nacional del Altiplano

Figura N° 4.17: Ejemplo de interface y clase con varias responsabilidades antes de su refactoring en la capa de lógica de negocios.



Elaboración: Propia.

Comprobado el incumplimiento de los principios SRP y/o OCP en estas clases, su refactoring consistió en crear nuevas clases que contengan responsabilidades únicas, también, se crearon interfaces que permitan segregar sus métodos, lo que permitió cumplir con el principio ISP. Estos refactoring conllevaron a corregir la aplicación para que pasen las pruebas unitarias rotas. En algunos casos fue necesario crear nuevas clases de prueba que implementen las pruebas unitarias de las clases creadas, por ejemplo, se tuvo que crear la clase de prueba *ControlPasoFacadeImplTest* que contiene las pruebas unitarias de la nueva clase *ContolPasoFacadeImpl*.

- Algunos cambios conllevaron a corregir la capa de presentación (módulo *cia-web)*, en el que se introdujeron errores por los cambios realizados.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

- En algunos casos se encontró comportamiento duplicado, por ejemplo: la clase *RetiroFacadeImpl* tiene el método *getCurrentYer(String format)*, pero también encontramos que la clase *ParametroFacadeImpl* contiene el mismo método, esto se muestra en la Figura N° 4.18. Este método devuelve el año actual desde el servidor de base de datos, por lo que debería estar en una clase separada de ambas. Su refactoring consistió en colocar este método en la clase nueva *commonFacadeImpl* con su interface *commonFacade*, y se procedió a eliminar el método repetido de las clases *RetiroFacadeImpl y ParametroFacadeImpl,* y de sus respectivas interfaces. Se corrigió las dependencias rotas, para que usen la clase nueva: *commonFacadeImpl.*

Figura N° 4.18: Ejemplo de comportamiento duplicado en la capa de lógica de negocios antes de su refactoring.

RetiroFacadeImpl.java

```
162 @Override
163 public String getCurrentYear(String format) {
164     String year = null;
165     year = retiroDAO.currentYear();
166     if (format.toUpperCase().equals("YY")) {
167         year = year.substring(2, 4);
168     }
169     return year;
170 }
```

ParametroFacadeImpl.java

```
119 @Override
120 public String getCurrentYear(String format){
121     String year = null;
122     year = paramDAO.currentYear();
123     if (format.toUpperCase().equals("YY")) {
124         year = year.substring(2,4);
125     }
126     return year;
127 }
```

Elaboración: Propia.

- El principio SRP indica que los métodos también deben implementar una única responsabilidad, por tanto, se analizaron los métodos con mayor número de líneas. Por ejemplo, en la clase *RetiroFacadeImpl,* se revisó el método *isEditable* -entre otros métodos-, este método mostrado en la Figura N° 4.19, tiene como función comprobar dos reglas de negocio: 1) El retiro solo puede ser modificado por el mismo usuario que lo registró y 2) El retiro solo puede ser modificado en un tiempo de 24 horas desde su registro. Este método incumple SRP porque tiene dos

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

responsabilidades, la primera es verificar que el usuario es el mismo que realizó el registro, y la segunda es verificar que el tiempo transcurrido sea menor a 24 horas. Asimismo, si fuese necesario ingresar alguna nueva regla de negocio para permitir la edición de un retiro tendríamos que modificar este método, por tanto, también incumple OCP. Su refactorización, consistió en aplicar el patrón de diseño Strategy[22] con la finalidad de que cumpla con OCP, por lo que se creó la clase *RetiroEditablePolicy,* también se creó la clase *CiaAppException* derivada de *Exception* para el manejo de excepciones del caso de estudio; además, se reemplazó el uso del método *System.out.println* por *loggers*. El resultado obtenido, se aprecia en la Figura N° 4.20.

Figura N° 4.19: Ejemplo de un método que infringe los principios SRP y OCP en la capa de lógica de negocios.

```java
@Override
public boolean isEditable(Retiro retiro, String userName) throws Exception {
    short hPermitidas = 1440; // 24 horas
    long mTrans = 0;
    long hTrans = 0;

    System.out.println(retiro.getRetiroId());
    System.out.println(userName);

    mTrans = HelperDate.dateDiff(retiro.getFecreg(), new Date());
    hTrans = mTrans / (1000 * 60); // minutos

    logger.info("TIEMPO TRANSCURRIDO (MINUTOS): " + hTrans);

    if (!userName.equals(retiro.getUsureg().trim().toLowerCase())) {
        throw new Exception("EL USUARIO QUE INTENTA MODIFICAR ES DIFERENTE "
                + "AL USUARIO QUE REGISTRO LA DESCARGA. NO ES POSIBLE MODIFICAR LA DESCARGA");
    }

    if (hTrans > hPermitidas) {
        throw new Exception("EL TIEMPO TRANSCURRIDO: " + (hTrans / 60) + "(HORAS), ES SUPERIOR A 24 HORAS. "
                + "NO ES POSIBLE MODIFICAR LA DESCARGA");
    }
    return true;
}
```

Elaboración: Propia.

---

[22] (Estrategia), Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan (Gamma, Helm, Johnson, & Vlissides, 2003).

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Figura N° 4.20: Ejemplo de método después de su refactoring en la capa de lógica de negocio.

```
@Override
public boolean isEditable(Retiro retiro, String userName) throws CiaAppException {
    logger.info("Comprobando reglas de edicion para el Retiro ID / Usuario Id: " + retiro.getRetiroId() + " / "
            + userName);

    for (RetiroEditablePolicy politica : RetiroEditablePolicy.values()) {
        politica.validar(retiro, userName);
    }
    return true;
}
```

Elaboración: Propia.

- Se encontró que las clases que procesaban datos de páginas Webs externas, concentraban un alto número de incidencias, por ejemplo, la clase *DamHelper* tenía 54 incidencias, esta clase tenía tres métodos estáticos con la responsabilidad de extraer los datos de una declaración de importación desde una página Web pública. Estas clases, incumplían los principios SRP y OCP, su refactoring consistió en las siguientes acciones:
  - Eliminar todas las variables sin uso.
  - Se creó la clase nueva *ConstantesHtml, con la responsabilidad* de contener los atributos constantes.
  - Se creó paquetes nuevos que contengan a las nuevas clases creadas.
  - Se implementó el patrón de diseño Cadena de Responsabilidad[23], para construir objetos con los datos extraídos de páginas Web públicas, por ejemplo, para la creación de objetos de la clase *Poliza*.
  - Se comprobó que las nuevas clases no infrinjan alguno de los principios SOLID.

---

[23] Chain of Responsibility (Cadena de Responsabilidad), evita acoplar el emisor de una petición a su receptor, al dar a más de un objeto la posibilidad de responder a la petición. Crea una cadena con los objetos receptores y pasa la petición a través de la cadena hasta que ésta sea tratada por algún objeto (Gamma, Helm, Johnson, & Vlissides, 2003).

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

- Respecto al principio DIP, se comprobó que este módulo dependía solo de abstracciones (interfaces) de la capa inferior (persistencia), por tanto, cumple con este principio. La Figura N° 4.21, muestra un ejemplo, donde se consigue la inyección de dependencias mediante el framework Spring con la anotación *@Resource*.
- Respecto al principio ISP, se consiguió interfaces pequeñas con métodos cohesivos. La Figura N° 4.22, muestra un ejemplo.

Figura N° 4.21: Ejemplo de inyección de dependencias en la capa de persistencia.

```
ConductorFacadeImpl.java
1  package com.cia.business.service.common.impl;
2
3⊕ import javax.annotation.Resource;
12
13 @Service(value = "conductorFacade")
14 public class ConductorFacadeImpl implements ConductorFacade {
15
16     private ConductorRetiroDAO conductorDAO = null;
17
18⊖    @Override
19     @Transactional(propagation = Propagation.REQUIRED, readOnly = true)
20     public ConductorRetiroDTO findByNumDoc(String tipoDoc, String numDoc) {
21         return conductorDAO.findByNumDoc(tipoDoc, numDoc);
22     }
23
24⊖    public ConductorRetiroDAO getConductorDAO() {
25         return conductorDAO;
26     }
27
28⊖    @Resource
29     public void setConductorDAO(ConductorRetiroDAO conductorDAO) {
30         this.conductorDAO = conductorDAO;
31     }
32
33 }
34
```

Elaboración: Propia.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

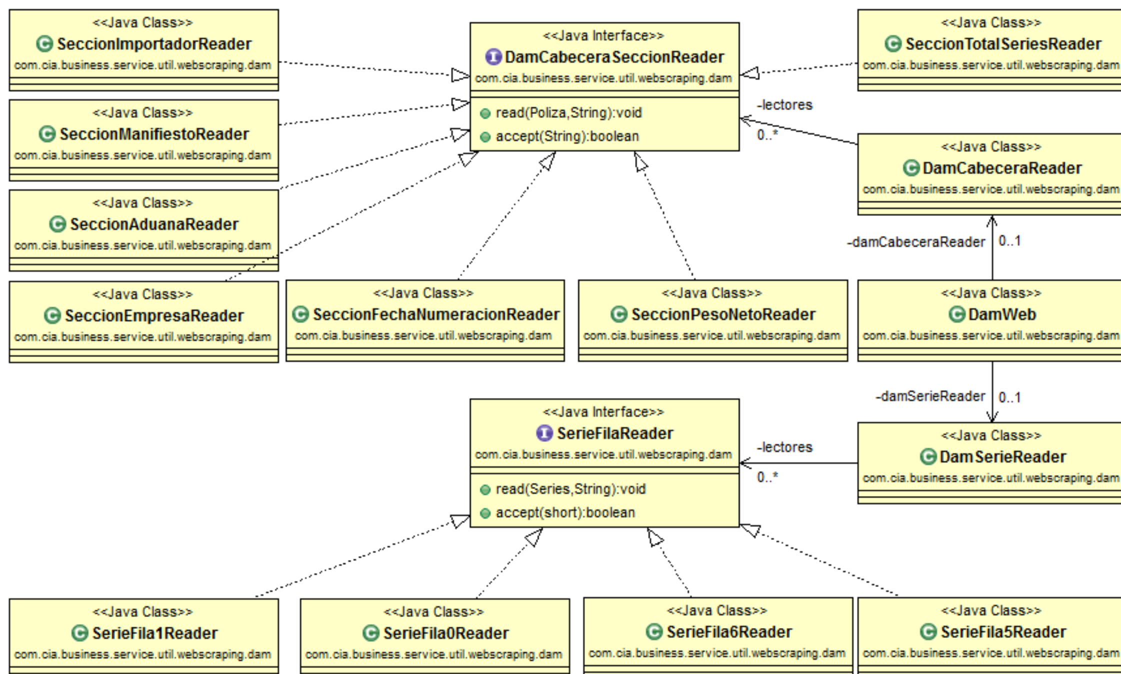
Figura N° 4.22: Ejemplo de interface que cumple con ISP en la capa de persistencia.

```java
LoginFacade.java
1  package com.cia.business.service.common;
2
3⊕ import java.io.Serializable;
7
8  public interface LoginFacade extends Serializable {
9
10     public void createLogin(Login user);
11
12     public void saveLogin(Login user);
13
14     public boolean cambiarClave(Login login, String passActual, String newPasswd, String passConfirm);
15
16     public boolean resetearClave(Login login, String password, String passConfirm);
17
18     public Login findLoginById(String loginId);
19
20     public Login findLoginByEmpleadoId(String empId);
21
22     public List<Login> findLoginByObserv(String observ);
23
24 }
25
```

Elaboración: Propia.

La Figura N° 4.23, muestra parte del diagrama de clases obtenido después del refactoring, en el cual se usó el patrón de diseño Cadena de Responsabilidad.

Figura N° 4.23: Implementación del patrón de diseño cadena de responsabilidad en la capa de lógica de negocios.



Elaboración: Propia.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Criterios de éxito

Al finalizar el refactoring se comprueba que todas las pruebas unitarias y de integración se ejecuten y pasen correctamente.

4.3.3.4. Refactoring de la capa de persistencia

La capa de persistencia está conformada por el módulo *cia-persistence*. Este módulo implementa los métodos que persisten los objetos del modelo de dominio en una base de datos relacional. Implementa el patrón de diseño DAO[24] usando JPA/Hibernate. Usa el framework Spring para aprovechar la inyección de dependencias, lo que favorece al cumplimiento del principio DIP de SOLID.

a. Identificar los puntos de cambio

La Tabla N° 4.9, muestra el análisis de este módulo antes su refactoring, lo que sirvió como guía para identificar los puntos de cambio. El detalle de incidencias por clases de este módulo se muestra en la Tabla N° E.3 del Anexo E.

Tabla N° 4.9: Análisis de la capa de persistencia del caso de estudio antes de su refactoring.

Aplicación Web (Caso de Estudio - Iteracion 5)		Líneas de código	N° de bugs	N° de Vulnerabilities	N° de Code smells	Total de Incidencias	Cobertura de código (%)	Código duplicado (%)
Módulo	Paquete							
cia-persistence	com.cia.persistence.dao.common.jpa	487	10	3	15	28	54.20%	0.00%
	com.cia.persistence.dao.retiro.jpa	389	4	0	24	28	7.60%	18.30%
	com.cia.persistence.dao.jdbc	109	7	0	8	15	3.30%	0.00%
	com.cia.persistence.dao.poliza.jpa	271	2	0	8	10	52.80%	46.40%
	com.cia.persistence.dao.controlpaso.jpa	131	2	0	6	8	64.80%	7.80%
	com.cia.persistence.dao.retiro	39	0	0	6	6		0.00%
	com.cia.persistence.dao.common	74	0	0	2	2		0.00%
	com.cia.persistence.dao.controlpaso	20	0	0	2	2		0.00%
	com.cia.persistence.dao.asignacion	13	0	0	0	0		0.00%
	com.cia.persistence.dao.poliza	46	0	0	0	0		0.00%

Elaboración: Propia.

[24] Objeto de Acceso a Datos.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

b. Encontrar los puntos de prueba

Las clases y sus métodos que presentan incidencias son considerados como puntos de prueba. El refactoring se realizó con el set de pruebas que se escribieron en una etapa previa y no fue necesario crear nuevas pruebas.

c. Romper dependencias

Este módulo cuenta con pruebas de integración con la base de datos, por tanto, no fue necesario romper sus dependencias.

d. Escribir pruebas

Las pruebas de integración escritas previamente para este módulo utilizan las librerías de Spring JUnit. La Figura N° D.4 del Anexo D, muestra un ejemplo de las pruebas escritas en esta capa.

e. Realizar cambios y refactorizar

e.1. Refactoring usando DRY

Las acciones ejecutadas se describen brevemente a continuación:

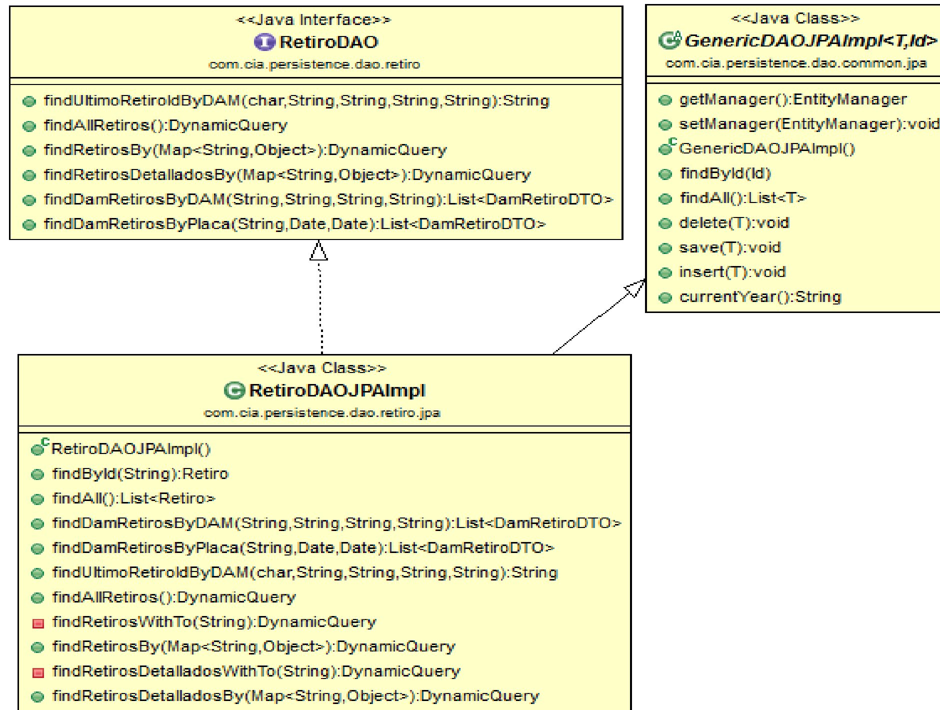
- Se identificó las clases con código duplicado. Por ejemplo, las clases: *RetiroDAOJPAImpl*, *ControlPasoDAOJPAImpl*, entre otras.
- Se identificó que algunas clases redefinían métodos implementados en su clase base, estos métodos, además de tener el mismo nombre definían el mismo comportamiento. Por ejemplo, los métodos *findById y findAll*, que se encuentran implementados en la clase base *GenericDAOJPAImpl*. El refactoring realizado consistió en eliminar los métodos duplicados de las clases derivadas, previendo, además no infringir el principio LSP.
- Algunas clases que presentaban un porcentaje elevado de líneas de código, por su complejidad, se optó por dejar su refactoring usando los principios SOLID. Por

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

ejemplo, la clase *RetiroDAOJPAImpl*, presentaba el 26% de código duplicado, esta clase se aprecia en la Figura N° 4.24.

Figura N° 4.24: Ejemplo de clase con código duplicado en la capa de persistencia.



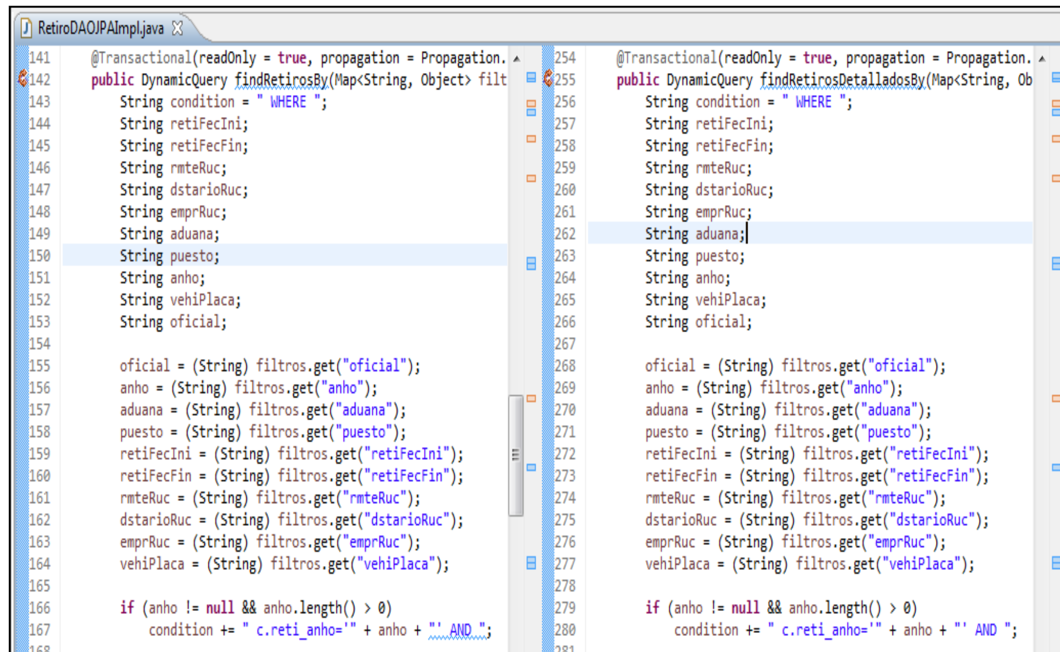
Elaboración: Propia.

Al examinar los métodos: *findRetirosBy(Map<String, Object> filtros) y findRetirosDetalladosBy(Map<String, Object> filtros), mostrados* en la Figura N° 4.25, se observa que presentan código duplicado, y tienen como responsabilidad construir consultas SQL que están definidas en la misma clase, adicionalmente, en estos métodos se aprecia una excesiva complejidad ciclomática (cadena de ifs). Sin embargo, para eliminar estas anomalías primero se debe refactorizar la clase *DinamycQuery*, además, modificar la forma en que se construyen las consultas SQL en busca de cumplir con SOLID.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Figura N° 4.25: Ejemplo de métodos con código duplicado que infringen el principio DRY en la capa de persistencia.



```
RetiroDAOJPAImpl.java
141    @Transactional(readOnly = true, propagation = Propagation.
142    public DynamicQuery findRetirosBy(Map<String, Object> filt
143        String condition = " WHERE ";
144        String retiFecIni;
145        String retiFecFin;
146        String rmteRuc;
147        String dstarioRuc;
148        String emprRuc;
149        String aduana;
150        String puesto;
151        String anho;
152        String vehiPlaca;
153        String oficial;
154
155        oficial = (String) filtros.get("oficial");
156        anho = (String) filtros.get("anho");
157        aduana = (String) filtros.get("aduana");
158        puesto = (String) filtros.get("puesto");
159        retiFecIni = (String) filtros.get("retiFecIni");
160        retiFecFin = (String) filtros.get("retiFecFin");
161        rmteRuc = (String) filtros.get("rmteRuc");
162        dstarioRuc = (String) filtros.get("dstarioRuc");
163        emprRuc = (String) filtros.get("emprRuc");
164        vehiPlaca = (String) filtros.get("vehiPlaca");
165
166        if (anho != null && anho.length() > 0)
167            condition += " c.reti_anho='" + anho + "' AND ";
```

```
254    @Transactional(readOnly = true, propagation = Propagation.
255    public DynamicQuery findRetirosDetalladosBy(Map<String, Ob
256        String condition = " WHERE ";
257        String retiFecIni;
258        String retiFecFin;
259        String rmteRuc;
260        String dstarioRuc;
261        String emprRuc;
262        String aduana;
263        String puesto;
264        String anho;
265        String vehiPlaca;
266        String oficial;
267
268        oficial = (String) filtros.get("oficial");
269        anho = (String) filtros.get("anho");
270        aduana = (String) filtros.get("aduana");
271        puesto = (String) filtros.get("puesto");
272        retiFecIni = (String) filtros.get("retiFecIni");
273        retiFecFin = (String) filtros.get("retiFecFin");
274        rmteRuc = (String) filtros.get("rmteRuc");
275        dstarioRuc = (String) filtros.get("dstarioRuc");
276        emprRuc = (String) filtros.get("emprRuc");
277        vehiPlaca = (String) filtros.get("vehiPlaca");
278
279        if (anho != null && anho.length() > 0)
280            condition += " c.reti_anho='" + anho + "' AND ";
```

Elaboración: Propia.

e.2. Refactoring usando SOLID

- Se verificó que las clases que implementan relaciones de herencia no infrinjan el principio LSP. Por ejemplo, la clase *GenericDAOJPAImpl,* es la clase base para el resto de clases DAOs, las que no modifican su comportamiento, por tanto, no infringen este principio.
- Se comprobó que el principio ISP, también se cumple en este módulo, comprobando que no posea interfaces gordas (con muchos métodos), en su lugar, tenga interfaces específicas para el comportamiento que exponen. Por ejemplo, el mostrado en la Figura N° 4.26.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Figura N° 4.26: Ejemplo de interface y clase que cumplen con ISP en la capa de persistencia.

GenericDAO.java

```java
package com.cia.persistence.common.dao;

import java.io.Serializable;

/**
 *
 * @author wsucasac
 *
 * @param <T>
 * @param <Id>
 */
public interface GenericDAO<T, I extends Serializable> extends Serializable {

    public T findById(I id);

    public List<T> findAll();

    public void merge(T objeto);

    public void remove(T objeto);

    public void persist(T objeto);

    public Connection getConnecion();
}
```

*GenericDAOJPAImpl.java

```java
package com.cia.persistence.common.dao.jpa;

import java.io.Serializable;

@Repository
public abstract class GenericDAOJPAImpl<T, I extends Serializable>
    implements GenericDAO<T, I>, Serializable {

    private static final long serialVersionUID = 1L;

    private Class<T> claseDePersistencia;

    private EntityManager manager;

    public EntityManager getManager() {

    public void setManager(EntityManager manager) {

    public Connection getConnecion() {

    public GenericDAOJPAImpl() {

    public T findById(I id) {

    public List<T> findAll() {

    public void remove(T objeto) {

    public void merge(T objeto) {

    public void persist(T objeto) {
}
```

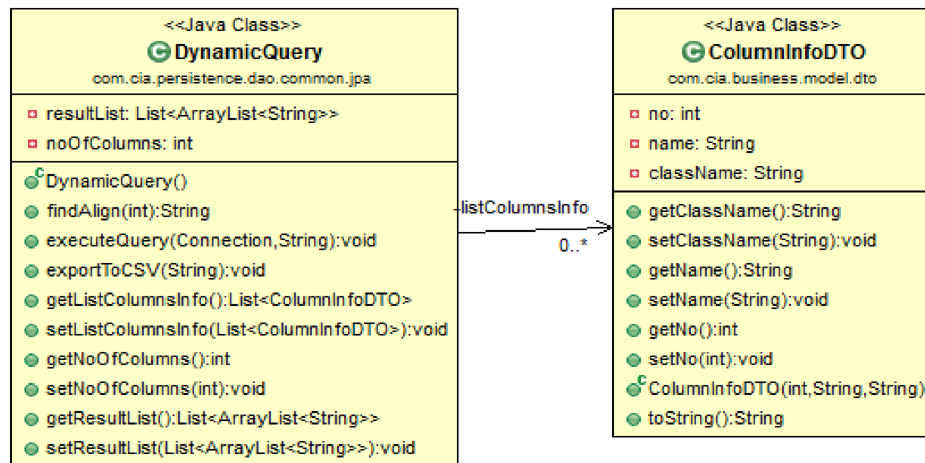
Elaboración: Propia.

- Se observó que algunas clases DAOs definían y ejecutaban consultas SQL nativas, las que eran mapeadas a objetos POJO[25]. El refactoring de estas clases para conseguir que cumplan con SRP y OCP, consistió en: extraer las sentencias SQL a clases independientes, se creó métodos privados que se encarguen del mapeo del objeto ResultSet a los objetos POJO correspondientes, se corrigió el tratamiento de excepciones que manejan las conexiones a la base de datos, y se agregó loggers para la captura de las excepciones lanzadas.
- Un ejemplo, de una clase que incumplía con SRP y OCP, es la clase *DynamicQuery* mostrada en la Figura N° 4.27, está clase tiene las responsabilidades de: 1) Ejecutar consultas SQL nativas, y 2) Exportar el resultado en formato CSV, incumpliendo los principios SRP por tener dos motivos de cambio, y también incumple OCP, porque

[25] Antiguo Objeto Plano de Java.

si deseamos exportar el resultado en algún otro formato como por ejemplo HTML tendríamos que modificar esta clase añadiendo un nuevo método.

Figura N° 4.27: Ejemplo de clases que incumplen con SRP y OCP en la capa de persistencia.



Elaboración: Propia.

El refactoring de esta clase consistió en:

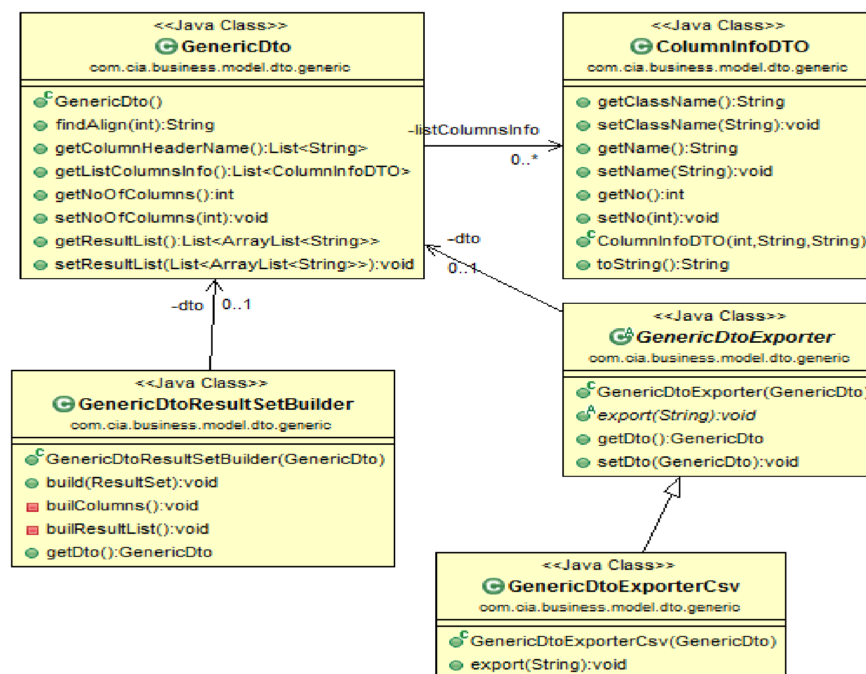
- Se renombró la clase a *GenericDto*, puesto que puede emplearse para obtener el resultado de cualquier consulta nativa SQL.
- Se extrajo la funcionalidad de exportar el resultado en formato CSV a la nueva clase *GenericDtoExporterCsv*, que extiende de la clase abstracta *GenericDtoExporter*, de esta manera, si fuese necesario ampliar la funcionalidad de exportar los datos a un formato como HTML, solo será necesario crear una clase concreta que extienda e implemente el método *export* de la clase *GenericDtoExporter*, cumpliendo con el principio OCP. La Figura N° 4.31, muestra el código fuente obtenido.
- La responsabilidad de ejecutar consultas SQL y el manejo de la conexión con la base de datos, se dejó a las clases DAOs; además, se creó una clase *GenericDtoResultSetBuilder*, que implementa el patrón de diseño Builder, con la responsabilidad crear los objetos en base al *ResultSet* que se le pase.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

- Se verifica que las nuevas clases cumplan con el principio LSP, toda vez que las clases derivadas no alteran el comportamiento de la clase base.
- De forma general fue necesario corregir las pruebas que se rompieron por los cambios efectuados. El diseño final obtenido después de su refactoring se aprecia en la Figura N° 4.28.

Figura N° 4.28: Clases GenericDtos, obtenidos después del refactoring de la capa de persistencia.



Elaboración: Propia.

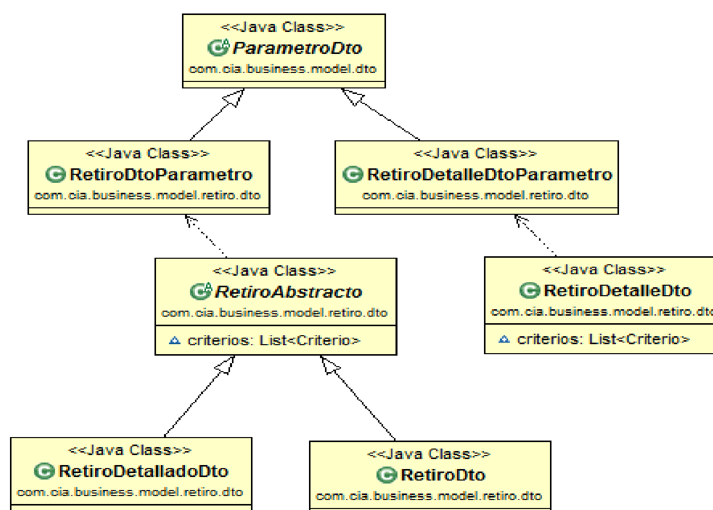
- Otro ejemplo del refactoring de esta capa, se presenta con la clase *RetiroDAOJPAImpl.* Con el objetivo de eliminar el código repetido que posee y conseguir que no infrinja con SOLID, se realizó las siguientes acciones en su refactoring:
    - Se extrajeron las consultas SQL a las clases nuevas: *RetiroDetalladoDto*, *RetiroDto*, y *RetiroDetalleDto*, las que extienden su comportamiento de la clase abstracta *RetiroAbstracto.*

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

- Se crearon las clases: *ParametroDto*, *RetiroDtoParametro* y *RetiroDetalleDtoParametro*, que contienen los atributos que permiten aplicar los criterios de consulta en las sentencias SQL. El resultado obtenido se aprecia en la Figura N° 4.29.

Figura N° 4.29: Clases DTOs y Parámetros, obtenidos después del refactoring de la capa de persistencia.



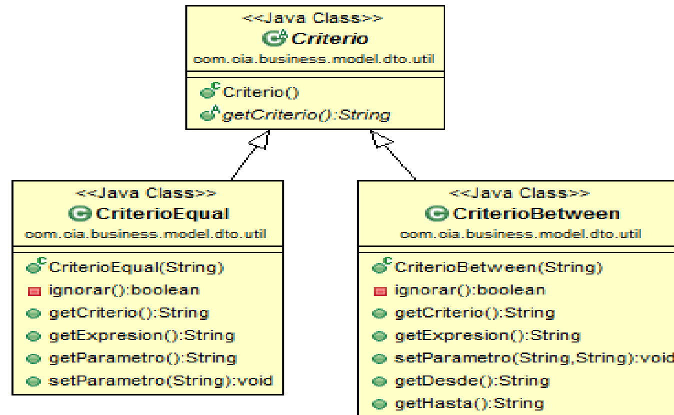
Elaboración: Propia.

- Para construir las consultas SQL con criterios variables, se creó la clase abstracta *Criterio*, junto a las clases que la implementan *CriterioEqual* y *CriterioBetween*. La Figura N° 4.30, muestra el diseño de estas clases, las cuales permitieron eliminar la complejidad ciclomática (cadenas de ifs consideradas como code smell). Las clases obtenidas no infringen los principios SOLID.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Figura N° 4.30: Clases que corresponden a criterios para consultas SQL obtenidos después del refactoring de la capa de persistencia.



Elaboración: Propia.

- De manera general, en las clases de este módulo se reemplazaron los valores literales por constantes.
- De forma general, fue necesario crear nuevos paquetes para mejorar su estructura.

La Figura N° 4.31 y Figura N° 4.32, muestran parte del código obtenido después del refactoring desarrollado.

Figura N° 4.31: Ejemplo de una clase obtenida después del refactoring de la capa de persistencia.

```
GenericDtoExporterCsv.java
1  package com.cia.business.model.dto.generic;
2
3  import java.io.FileWriter;
7
8  public class GenericDtoExporterCsv extends GenericDtoExporter {
9
10     private static final String SEPARATOR = ",";
11
12     public GenericDtoExporterCsv(GenericDto dto) {
13         super(dto);
14     }
15
16     public void export(final String fileReport) throws IOException {
17         try (FileWriter writer = new FileWriter(fileReport, true)) {
18
19             writer.write(String.join(SEPARATOR, this.getDto().getColumnHeaderName()) + System.lineSeparator());
20
21             for (ArrayList<String> list : this.getDto().getResultList()) {
22                 List<String> rows = new ArrayList<>();
23                 for (String field : list) {
24                     rows.add("'" + field + "'");
25                 }
26                 writer.append(String.join(SEPARATOR, rows) + System.lineSeparator());
27             }
28             writer.flush();
29         }
30     }
31 }
```

Elaboración: Propia.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Figura N° 4.32: Fragmento de código fuente obtenido después del refactoring de la capa de persistencia.

Criterio.java

```java
package com.cia.business.model.dto.querys.util;

public abstract class Criterio {

    protected static final String SQL_AND = " AND ";

    protected static final String SQL_BETWEEN = " BETWEEN ";

    protected String atributo;

    public abstract String getCriterio();

}
```

CriterioBetween.java

```java
package com.cia.business.model.dto.querys.util;

public class CriterioBetween extends Criterio {

    private String desde;

    private String hasta;

    public CriterioBetween(String atributo) {
        this.atributo = atributo;
        this.desde = null;
        this.hasta = null;
    }

    private boolean ignorar() {
        return getDesde() == null || getDesde().isEmpty() || getHasta() == null || getHasta().isEmpty();
    }

    @Override
    public String getCriterio() {
        return this.ignorar() ? "" : getExpresion();
    }

    public String getExpresion() {
        return SQL_AND + this.atributo + SQL_BETWEEN + " '" + getDesde() + "' AND '" + getHasta() + "' ";
    }

    public void setParametro(String param1, String param2) {
```

CriterioEqual.java

```java
package com.cia.business.model.dto.querys.util;

public class CriterioEqual extends Criterio {

    private String parametro;

    public CriterioEqual(String atributo) {
        this.atributo = atributo;
        this.setParametro(null);
    }

    private boolean ignorar() {
        return getParametro() == null || getParametro().isEmpty();
    }

    @Override
    public String getCriterio() {
        return ignorar() ? "" : getExpresion();
    }

    public String getExpresion() {
        return SQL_AND + this.atributo + " = '" + getParametro() + "' ";
    }

    public String getParametro() {
        return parametro;
    }

    public void setParametro(String parametro) {
        this.parametro = parametro;
    }
}
```

Elaboración: Propia.

Criterios de éxito

Al finalizar el refactoring se comprueba que todas las pruebas se ejecuten y pasen correctamente.

4.3.3.5. Refactoring de la capa de presentación

La capa de persistencia está conformada por el módulo *cia-web*. Usa el framework JSF, y está configurada para que el framework Spring realice la inyección de dependencias con la capa de lógica de negocios (módulo de servicios).

a. Identificar los puntos de cambio

La Tabla N° 4.10, muestra el análisis de los paquetes de este módulo antes de su refactoring. Este análisis se usó como guía para identificar los puntos de cambio. El análisis detallado de sus clases se muestra en la Tabla N° E.4 del Anexo E.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Tabla N° 4.10: Análisis de la capa de presentación del caso de estudio antes de su refactoring.

Aplicación Web (Caso de Estudio - Iteracion 6)		Líneas de código	N° de bugs	N° de Vulnerabilities	N° de Code smells	Total de Incidencias	Cobertura de código (%)	Código duplicado (%)
Módulo	**Paquete**							
cia-web	com.cia.presentation.retiro.controller	**2.7k**	73	0	48	**121**	**17.80%**	**15.30%**
	com.cia.presentation.util	**383**	12	1	36	**49**	**0.00%**	**0.00%**
	com.cia.presentation.common.controller	**872**	22	4	20	**46**	**8.30%**	**3.80%**
	com.cia.presentation.controlpaso.controlle	**956**	21	0	10	**31**	**19.90%**	**14.00%**
	com.cia.presentation.poliza.controller	**425**	11	0	5	**16**	**0.00%**	**12.70%**
	com.cia.presentation.asignacion.controller	**434**	7	0	6	**13**	**0.00%**	**0.00%**

Elaboración: Propia.

b. Encontrar los puntos de prueba

Este módulo cuenta con un set de pruebas escritas en un paso previo, las que se usaron para comprobar los refactorings.

c. Romper dependencias

En una actividad previa, se escribió pruebas unitarias, por lo que fue necesario romper sus dependencias utilizando el framework Mockito y PowerMock. En la Figura N° D.5 del Anexo D, se muestra un ejemplo de las pruebas unitarias escritas en esta capa.

d. Escribir pruebas

No se escribieron nuevas pruebas.

e. Realizar cambios y refactorizar

e.1. Refactoring usando DRY

Las acciones realizadas son descritas brevemente a continuación:

- Se identificaron las clases con código duplicado y se buscó que cumplan con DRY.
- Se extrajeron los atributos repetidos de los ManagedBean a nuevas clases, por ejemplo, se crearon las clases nuevas *Periodo* y *Parametro*. Las clases creadas se

Repositorio Institucional UNA-PUNO

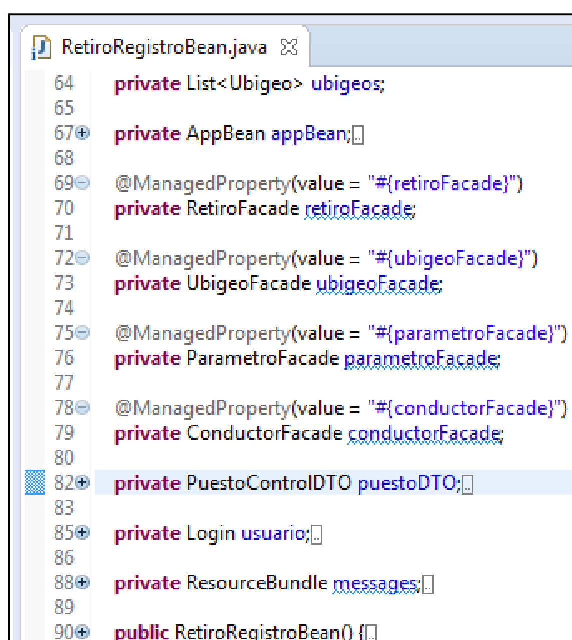
No olvide citar esta tesis

usan por medio de la composición en los ManagedBean que lo requerían (esencialmente en las funcionalidades de consulta de información).

e.2. Refactoring usando SOLID

- Respecto al principio ISP, este módulo carece de interfaces, por lo que no incumple este principio.
- Respecto al principio DIP, al ser la capa superior de nuestra arquitectura en capas, este módulo solo debe depender de abstracciones y no de concreciones de las capas inferiores, por tal motivo, se verificó que los ManagedBean dependan solo de interfaces, por ejemplo, las interfaces Facades desarrolladas en la capa de lógica de negocios; y por medio del framework Spring se realizó la inyección de dependencias. La Figura N° 4.33, muestra un ejemplo del uso de propiedades administradas de JSF que son gestionadas por el framework Spring.

Figura N° 4.33: Fragmento de código que muestra el uso de la inyección de dependencia en la capa de presentación.



Elaboración: Propia.

Universidad Nacional del Altiplano

- Respecto al principio LSP, en un inicio este módulo no contenía ninguna relación de herencia entre sus clases, y producto de los refactorings se añadió algunas relaciones de herencia teniendo en cuenta que no infrinjan el principio LSP.
- Se introdujo la clase *ConstantesVista*, con la responsabilidad de agrupar todos los valores constantes que son usadas en esta capa.
- Se renombró algunas clases para que describan mejor su responsabilidad, por ejemplo, la clase: *UtilReportBean* fue renombrada a *JasperUtils*.
- De forma general se extrajeron los atributos repetidos de los *ManagedBean* y se colocaron en clases nuevas.
- De forma general se modificó algunos métodos como *save*, que tenían como responsabilidades persistir nuevos objetos (crear) o guardar algún cambio en caso de edición; en su lugar se crearon métodos independientes que se encarguen de una sola responsabilidad buscando cumplir con SRP.
- Algunos refactoring realizados conllevaron a modificar los módulos de las capas inferiores, principalmente por el incumplimiento del principio SRP y OCP.
- El refactoring de las clases de esta capa, conllevo a realizar modificaciones mínimas en los archivos: xhtml, sin cambiar su comportamiento.
- De forma general, se reemplazó el uso de valores literales por constantes en las diferentes clases.
- Una de las clases con el mayor número de incidencias -en total 48- era la clase *UtilsBean*, mostrada en la Figura N° 4.34.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Universidad Nacional del Altiplano

Figura N° 4.34: Clase que presenta diversas incidencias en la capa de presentación del caso de estudio antes de su refactoring.

<<Java Class>>
UtilsBean
com.cia.presentation.util

UtilsBean()
populateTipos(ParametroFacade,String):LinkedList<SelectItem>
populateBuscarControles():LinkedList<SelectItem>
populateBuscarRetiros():LinkedList<SelectItem>
populateTipoExps():LinkedList<SelectItem>
populateAnios(int,int):LinkedList<SelectItem>
populatePorcentajes():LinkedList<SelectItem>
showMensaje(String):void
showMensaje(String,int):void
showPage(String):void
getUserName():String
getUsuario():Login
getAppParam(int):String
getAppParam0():boolean
setAppParam0(boolean):void
setAppParam(int,String):void
setAppParamPoliza(Poliza):void
setAppPuestoControl(PuestoControlDTO):void
getAppParamPoliza():Poliza
getPathAppWeb():String
populateAduanas(AduanaFacade):List<SelectItem>
populatePuestos(AduanaFacade,String):List<SelectItem>
populateAduanaDuas(AduanaFacade,String):List<SelectItem>
populateAduanasAndPuestos(AduanaFacade):List<SelectItem>
populateRetiroTipos():LinkedList<SelectItem>
downloadFile(String):StreamedContent
getClientIpAddress():String

Elaboración: Propia.

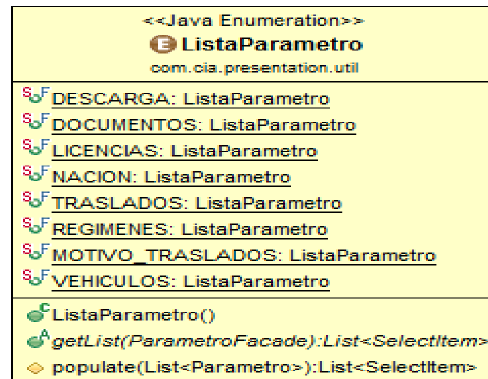
El refactoring de esta clase consistió en:

- Se creó una clase abstracta *GenericFaces,* que contenga los métodos genéricos y comunes a todos los ManagedBean de este módulo, eliminando de esta forma los métodos estáticos, y quitando la diversidad de responsabilidades de la clase *UtilsBean*. Se verifica que las clases que extienden de esta clase genérica no modifiquen su comportamiento para cumplir con LSP.
- Se empleó el patrón Strategy para poblar las listas que se muestran en la interface de usuario, consiguiendo cumplir con OCP. La Figura N° 4.35, muestra lo indicado.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Figura N° 4.35: Uso del patrón Strategy después del refactoring de la capa de presentación.



Elaboración: Propia.

- Se crean nuevas clases: *ListaAduana*, *ListaCommon y ListaParametro*, que permiten poblar las listas usadas en las interfaces de usuario y que no se llenen desde la base de datos.
- Se eliminan los métodos que no se usan, y pueden ser reemplazados por propiedades administradas en los ManagedBeans.
- Por último, se renombra está clase a un nombre que exprese mejor su responsabilidad *FacesUtils*.

- Se encontró que algunas clases en este módulo, además de cumplir con la responsabilidad de controlar la interface de usuario, también implementaban funcionalidades que corresponden a la capa de lógica de negocios. Por ejemplo: El paquete: *com.cia.presentation.retiro.controller*, presentaba un total de 121 incidencias, con 15.30% de código duplicado, en este paquete se encontró que la clase *RetiroDAMBean*, implementaba comportamiento de lógica de negocios, lo que por concepto no es responsabilidad de esta capa. La refactorización de estas clases consistió de forma general en:

Repositorio Institucional UNA-PUNO

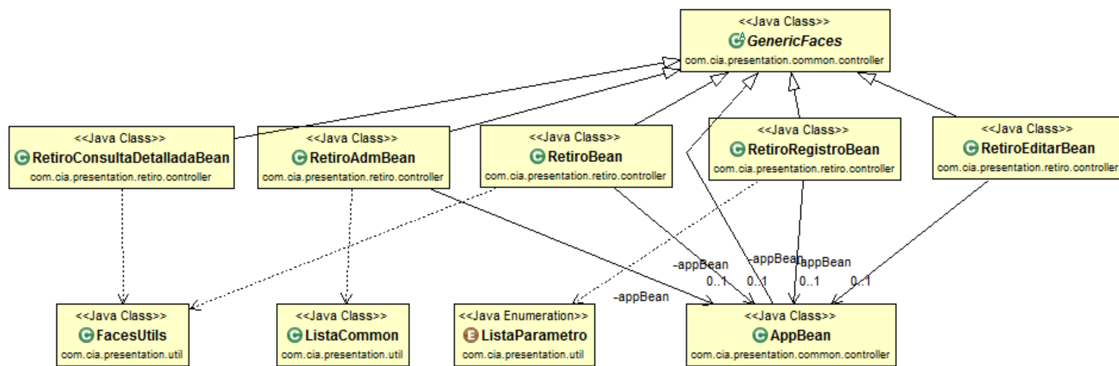
No olvide citar esta tesis

Universidad Nacional del Altiplano

- Se extrajo las responsabilidades de la lógica de negocio a nuevas clases y se las coloca en los módulos *business-model* o *business-services* según sus responsabilidades. Las clases creadas no rompen DRY o SOLID.
- En algunos casos se implementó el patrón de diseño Builder por la necesidad de construir objetos complejos o con características particulares que modifican el ingreso de datos en la interface de usuario de registro de nuevos retiros.
- En algunos casos se crearon nuevas clases que implementen el patrón de diseño Strategy, por ejemplo, la clase *UsuarioValidator*.

Parte del diseño final obtenido después del refactoring sobre la capa de presentación se muestra en la Figura N° 4.36.

Figura N° 4.36: Fragmento de diseño de clases obtenido después del refactoring de la capa de presentación.



Elaboración: Propia.

La Figura N° 4.37, muestra el código fuente que implementa el patrón de diseño Strategy en la capa de presentación.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Figura N° 4.37: Implementación del patrón de diseño Strategy después del refactoring de la capa de presentación.

ListaParametro.java

```java
package com.cia.presentation.util;

import java.util.LinkedList;[]

public enum ListaParametro {

    DESCARGA {[]
    DOCUMENTOS {[]
    LICENCIAS {[]
    NACION {[]
    TRASLADOS {[]
    REGIMENES {[]
    MOTIVO_TRASLADOS {[]
    VEHICULOS {[]

    public abstract List<SelectItem> getList(ParametroFacade facade);

    protected List<SelectItem> populate(List<Parametro> parametros) {
        List<SelectItem> lstItems = new LinkedList<>();
        for (Parametro parametro : parametros) {
            SelectItem si = new SelectItem();
            si.setLabel(parametro.getDenom());
            si.setValue(parametro.getId().getParamid());
            lstItems.add(si);
        }
        return lstItems;
    }
}
```

Elaboración: Propia.

Criterios de éxito

Se ejecutaron todas las pruebas y se corroboró que pasen en su totalidad. Adicionalmente, se compiló el proyecto y se verificó la creación de los módulos de todo el caso de estudio.

Para garantizar el funcionamiento inicial del caso de estudio, fue necesario realizar algunas pruebas de sistema y corregir algunos errores que se encontraron. Lo que evidencia que la cobertura de 34.1%, no fue la suficiente para garantizar el funcionamiento correcto de la aplicación durante el proceso de refactoring desarrollado.

4.3.4. Fase 4: Medir los resultados de la calidad interna

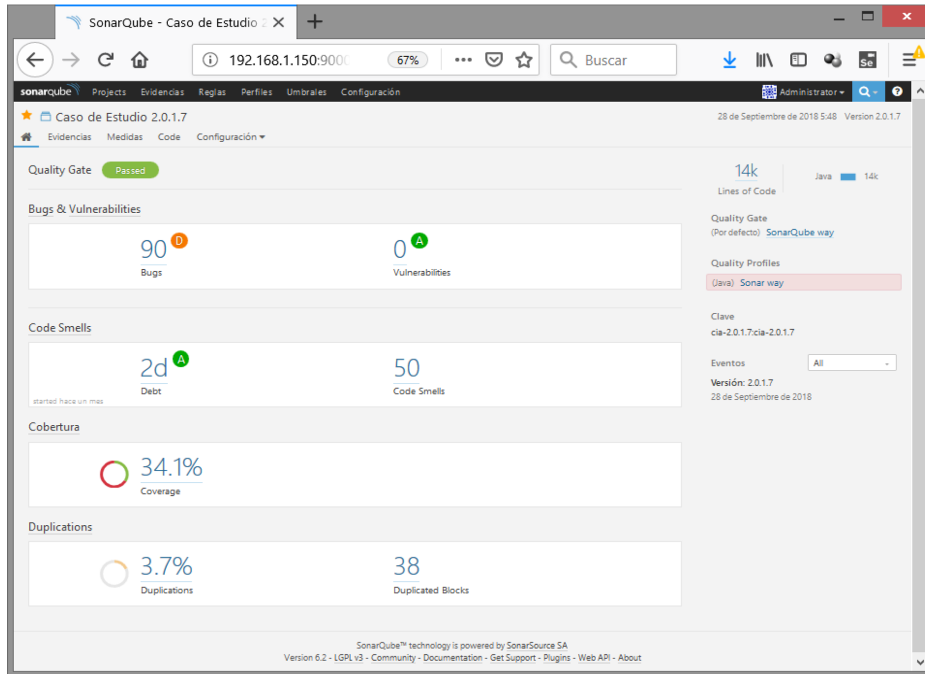
Al concluir el desarrollo del refactoring del caso de estudio, se realizó un análisis final con SonarQube. La Figura N° 4.38, muestra los resultados obtenidos. El número de incidencias obtenido fue de 140, el cual está conformado por 90 bugs, 0

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

(cero) vulnerabilidades, 50 code smells. Adicionalmente, se aprecia, 34.1% de cobertura de código y 3.7% de código duplicado.

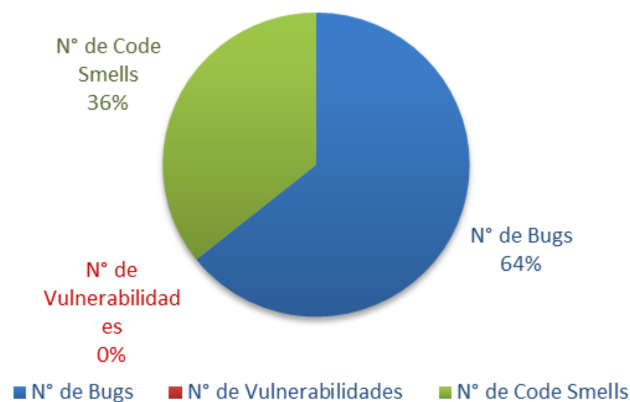
Figura N° 4.38: Medición de la calidad interna final del caso de estudio, obtenida con el software SonarQube después del proceso de refactoring.



Elaboración: Propia.

La Figura N° 4.39, muestra que del total de incidencias halladas al finalizar el proceso de refactoring, el 36% corresponde a code smells, 64% corresponde a bugs y 0% a vulnerabilidades.

Figura N° 4.39: Porcentaje final de incidencias del caso de estudio.



Elaboración: Propia.

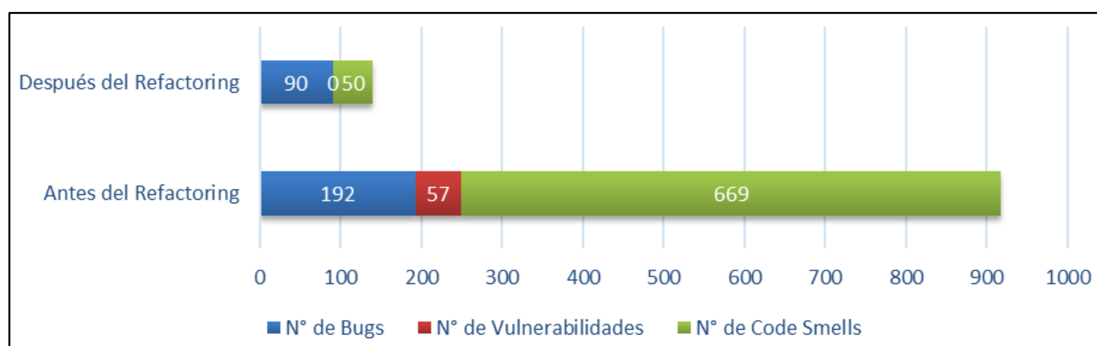
Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

En el Anexo G, se muestran diferentes reportes de SonarQube, que permiten ver con mayor detalle el análisis final del caso de estudio.

4.3.5. Fase 5: Evaluar la calidad interna final

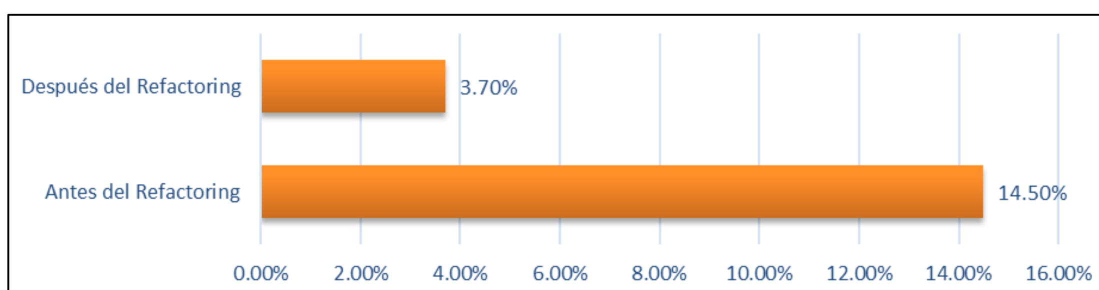
Para esta investigación, no se definió ninguna meta sobre la calidad interna deseada. Las Figura N° 4.40 y Figura N° 4.41, permiten comparar los resultados obtenidos sobre el número total de incidencias y el porcentaje código duplicado. En suma, las incidencias disminuyeron de 918 a 140 incidencias, el porcentaje de código duplicado disminuyo de 14.50% a 3.70%, con estos resultados, se cree conveniente concluir el proceso de refactoring, precisando que aún podría existir partes de código que podrían infringir con alguno de los principios DRY o SOLID.

Figura N° 4.40: Comparación de la calidad interna inicial y final del caso de estudio.



Elaboración: Propia.

Figura N° 4.41: Comparación del porcentaje de código duplicado inicial y final del caso de estudio.



Elaboración: Propia.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

4.4. Evaluación de la medida en que el uso de los principios DRY y S.O.L.I.D. en el proceso de refactoring desarrollado afectó la calidad interna del caso de estudio

La Tabla N° 4.11, presenta los resultados de la calidad interna alcanzada, medido por los atributos: mantenibilidad (número de code smells), fiabilidad (número de bugs), seguridad (número de vulnerabilidades) y código duplicado; adicionalmente, presenta la medición de su tamaño y cobertura de código. Esta evaluación se efectuó después de desarrollar el proceso de refactoring, y se obtuvo por medio del análisis al código fuente del caso de estudio con la herramienta SonarQube. Esta tabla permite apreciar una mejora en su calidad interna, obteniendo una reducción del número de incidencias de 918 a 140, logrando eliminar 778 incidencias que representan el 84.75% del total. Respecto al código duplicado, se aprecia una reducción de 14.50% a 3.70%, eliminándose el 10.80% de código duplicado. Adicionalmente, se aprecia que el tamaño se redujo de 15,571 a 13,872 líneas de código, notándose una reducción del 10.91%. Asimismo, la cobertura de código final es de 34.10%.

Tabla N° 4.11: Comparación de los atributos de calidad interna del caso de estudio antes y después del proceso de refactoring.

Tipo	N° de Bugs	N° de Vulnerabilidades	N° de Code Smells	Total Incidencias	Tamaño (líneas de código)	Cobertura de código (%)	Código duplicado (%)
Antes del Refactoring	192	57	669	**918**	**15,571**	**0.00%**	**14.50%**
Después del Refactoring	90	0	50	**140**	**13,872**	**34.10%**	**3.70%**
Mejora Alcanzada	102	57	619	**778**	**1,699**	**34.10%**	**10.80%**

Elaboración: Propia.

La Tabla N° 4.12, presenta el resultado del análisis de los paquetes Java que conforman el caso de estudio refactorizado.

Tabla N° 4.12: Análisis de los paquetes Java después del proceso de refactoring del caso de estudio.

Aplicación Web Java (Caso de Estudio) Refactorizado		Líneas de código	N° de bugs	N° de Vulnerabilities	N° de Code smells	Total de Incidencias	Cobertura de código (%)	Código duplicado (%)
Módulo	**Paquete**	**14k**	**90**	**0**	**50**	**140**	**34.10%**	**3.70%**
cia-business-model	com.cia.business.model.asignacion	77	0	0	0	**0**	86.40%	0.00%
	com.cia.business.model.asignacion.dto	110	0	0	0	**0**	58.80%	0.00%
	com.cia.business.model.common	786	0	0	2	**2**	33.70%	0.00%
	com.cia.business.model.common.dto	522	0	0	0	**0**	26.10%	0.00%
	com.cia.business.model.common.validator	65	0	0	0	**0**	81.80%	0.00%
	com.cia.business.model.controlpaso	218	0	0	0	**0**	69.70%	0.00%
	com.cia.business.model.controlpaso.dto	106	0	0	0	**0**	98.10%	0.00%
	com.cia.business.model.controlpaso.reporte	56	0	0	0	**0**	0.00%	0.00%
	com.cia.business.model.controlpaso.validators	13	0	0	0	**0**	75.00%	0.00%
	com.cia.business.model.dto.generic	164	2	0	0	**2**	73.10%	0.00%
	com.cia.business.model.dto.querys	77	0	0	0	**0**	83.30%	0.00%
	com.cia.business.model.dto.querys.util	61	0	0	0	**0**	75.00%	0.00%
	com.cia.business.model.poliza	1.1k	0	0	6	**6**	57.70%	7.20%
	com.cia.business.model.retiro	1.3k	2	0	4	**6**	31.10%	0.00%
	com.cia.business.model.retiro.builders	217	2	0	0	**2**	0.00%	0.00%
	com.cia.business.model.retiro.dto	320	0	0	0	**0**	51.40%	0.00%
	com.cia.business.model.retiro.policys	29	0	0	0	**0**	0.00%	0.00%
	com.cia.business.model.retiro.validators	14	0	0	0	**0**	75.00%	0.00%
	com.cia.business.model.seguridad	252	0	0	1	**1**	64.60%	0.00%
	com.cia.business.model.validators	4	0	0	1	**1**		0.00%
cia-business-service	com.cia.business.service.asignacion	16	0	0	0	**0**		0.00%
	com.cia.business.service.asignacion.impl	76	0	0	0	**0**	60.00%	0.00%
	com.cia.business.service.common	68	0	0	2	**2**		0.00%
	com.cia.business.service.common.impl	374	0	0	0	**0**	37.00%	0.00%
	com.cia.business.service.controlpaso	27	0	0	2	**2**		0.00%
	com.cia.business.service.controlpaso.impl	118	0	0	0	**0**	37.80%	0.00%
	com.cia.business.service.poliza	28	0	0	0	**0**		0.00%
	com.cia.business.service.poliza.impl	152	0	0	1	**1**	62.10%	15.50%
	com.cia.business.service.retiro	56	0	0	3	**3**		0.00%
	com.cia.business.service.retiro.impl	474	0	0	1	**1**	22.40%	5.00%
	com.cia.business.service.util.webscraping	189	0	0	2	**2**	44.90%	0.00%
	com.cia.business.service.util.webscraping.dam	361	0	0	1	**1**	92.90%	0.00%
cia-common	com.cia.common	111	0	0	0	**0**	0.00%	0.00%
	com.cia.common.exception	16	0	0	0	**0**	25.00%	0.00%
	com.cia.common.util	96	3	0	2	**5**	32.80%	0.00%
cia-persistence	com.cia.persistence.asignacion.dao	13	0	0	0	**0**		0.00%
	com.cia.persistence.asignacion.dao.jpa	65	0	0	0	**0**	100.00%	0.00%
	com.cia.persistence.common.dao	77	0	0	0	**0**		0.00%
	com.cia.persistence.common.dao.jpa	387	1	0	1	**2**	57.50%	0.00%
	com.cia.persistence.controlpaso.dao	26	0	0	2	**2**		0.00%
	com.cia.persistence.controlpaso.dao.jdbc	48	0	0	0	**0**	60.40%	0.00%
	com.cia.persistence.controlpaso.dao.jpa	119	0	0	1	**1**	46.30%	10.60%
	com.cia.persistence.poliza.dao	42	0	0	0	**0**		0.00%
	com.cia.persistence.poliza.dao.jpa	129	0	0	0	**0**	100.00%	0.00%
	com.cia.persistence.retiro.dao	45	0	0	2	**2**		0.00%
	com.cia.persistence.retiro.dao.jdbc	46	0	0	0	**0**	58.70%	0.00%
	com.cia.persistence.retiro.dao.jpa	227	0	0	1	**1**	35.30%	5.10%
cia-web	com.cia.presentation.asignacion.controller	399	4	0	0	**4**	0.00%	0.00%
	com.cia.presentation.common	90	0	0	0	**0**	57.90%	0.00%
	com.cia.presentation.common.controller	418	9	0	0	**9**	34.00%	0.00%
	com.cia.presentation.controlpaso.controller	931	19	0	3	**22**	21.60%	10.20%
	com.cia.presentation.log4j.servlets	15	0	0	0	**0**	0.00%	0.00%
	com.cia.presentation.poliza.controller	431	6	0	4	**10**	0.00%	11.80%
	com.cia.presentation.retiro.controller	2.1k	40	0	7	**47**	23.40%	11.20%
	com.cia.presentation.seguridad.controller	377	2	0	0	**2**	0.00%	0.00%

Elaboración: Propia.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

4.4.1. En cuanto al número de Bugs

SonarQube, mide el atributo de fiabilidad con el número de bugs hallados en el caso de estudio. La Tabla N° 4.13, muestra que el número de bugs eliminados es 102, quedando 90 bugs en el caso de estudio después del proceso de refactoring desarrollado.

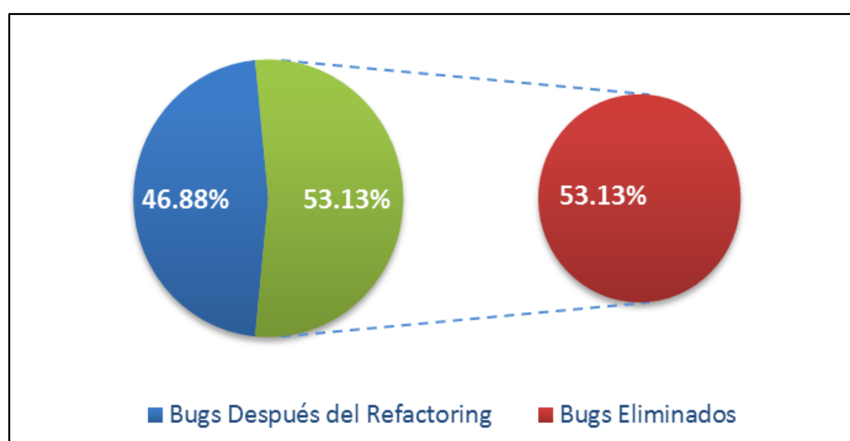
Tabla N° 4.13: Comparación del número de bugs antes y después del proceso de refactoring.

Bugs Antes del Refactoring		Bugs Después del Refactoring		Bugs Eliminados	
Cantidad	%	Cantidad	%	Cantidad	%
192	100.00%	90	46.88%	102	53.13%

Elaboración: Propia.

La Figura N° 4.42, muestra que la cantidad de bugs eliminados representa el 53.13% del número inicial, quedando el 46.88% en el caso de estudio después del refactoring. Este porcentaje eliminado representa una mejora en cuanto al número de bugs inicial y al alcanzado con el proceso de refactoring.

Figura N° 4.42: Porcentaje bugs eliminado después del proceso de refactoring.



Elaboración: Propia.

4.4.2. En cuanto al número de Vulnerabilidades

SonarQube, mide el atributo de seguridad con el número de vulnerabilidades halladas en el caso de estudio. La Tabla N° 4.14, muestra que el número de

vulnerabilidades eliminados es 57, quedando 0 (cero) vulnerabilidades en el caso de estudio después del proceso refactoring realizado.

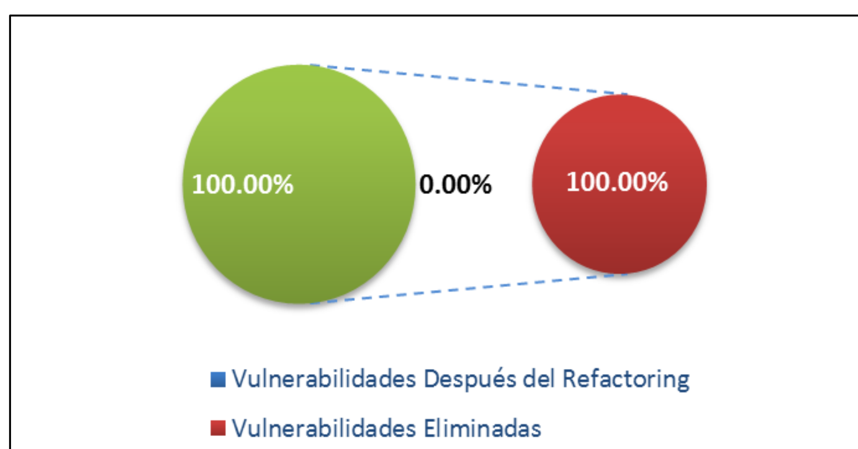
Tabla N° 4.14: Comparación del número de vulnerabilidades antes y después del proceso de refactoring.

Vulnerabilidades Antes del Refactoring		Vulnerabilidades Después del Refactoring		Vulnerabilidades Eliminadas	
Cantidad	%	Cantidad	%	Cantidad	%
57	100.00%	0	0.00%	57	100.00%

Elaboración: Propia.

La Figura N° 4.43, muestra que la cantidad de vulnerabilidades eliminadas representan el 100% del número inicial, quedado 0% de vulnerabilidades en el caso de estudio después de desarrollar el proceso de refactoring. Este porcentaje representa una mejora en cuanto al número de vulnerabilidades inicial y al alcanzado con el proceso de refactoring.

Figura N° 4.43: Porcentaje de vulnerabilidades eliminado después del proceso de refactoring.



Elaboración: Propia.

4.4.3. En cuanto al número de Code Smells

Respecto al atributo de mantenibilidad, este es medido por SonarQube considerando el número de code smells presentes en el caso de estudio. La Tabla N°

4.15, muestra que el número de code smells se redujo de 669 a 50, lográndose eliminar 619 code smells después del proceso refactoring efectuado. La cantidad de incidencias eliminadas de este tipo es el más alto, comparado con las incidencias de los atributos de fiabilidad y seguridad alcanzados en el caso de estudio.

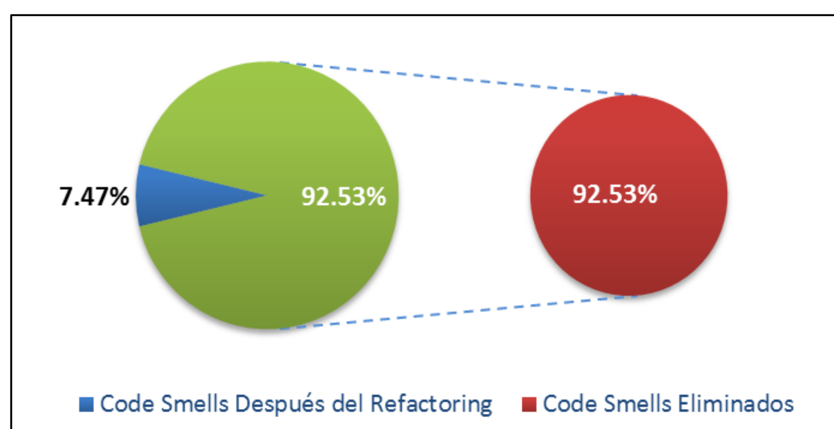
Tabla N° 4.15: Comparación del número de code smells antes y después del proceso de refactoring.

Code Smells Antes del Refactoring		Code Smells Después del Refactoring		Code Smells Eliminados	
Cantidad	%	Cantidad	%	Cantidad	%
669	100.00%	50	7.47%	619	92.53%

Elaboración: Propia.

La Figura N° 4.44, muestra que la cantidad de code smells eliminados representa el 92.53% del número inicial, quedando el 7.47% en el caso de estudio después del proceso de refactoring. El porcentaje eliminado corresponde a la mejora alcanzada en cuanto al atributo de mantenibilidad.

Figura N° 4.44: Porcentaje de code smells eliminado después del proceso de refactoring.



Elaboración: Propia.

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

Universidad Nacional del Altiplano

4.4.4. En cuanto al Código Duplicado

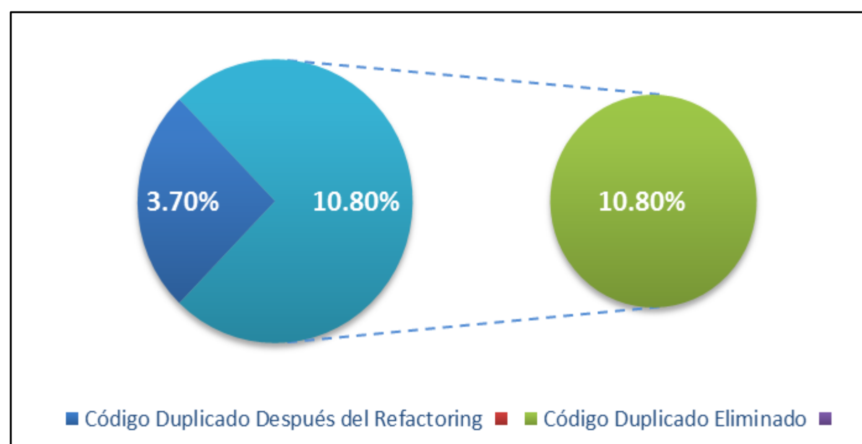
Respecto al porcentaje de código duplicado la Tabla N° 4.16, muestra que se obtuvo una reducción del 14.50% a 3.70% de código duplicado en el caso de estudio después del proceso del refactoring desarrollado.

Tabla N° 4.16: Comparación del porcentaje de código duplicado antes y después del proceso de refactoring.

Código Duplicado Antes del Refactoring	Código Duplicado Después del Refactoring	Código Duplicado Eliminado
%	%	%
14.50%	3.70%	10.80%

Elaboración: Propia.

Figura N° 4.45: Porcentaje de código duplicado eliminado después del proceso de refactoring.



Elaboración: Propia.

4.4.5. En cuanto al Tamaño de la Aplicación

La Tabla N° 4.17, muestra que el tamaño de la aplicación se redujo de 15,571 a 13,872 líneas de código, consiguiendo eliminar 1,699 líneas de código con los refactoring desarrollados.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

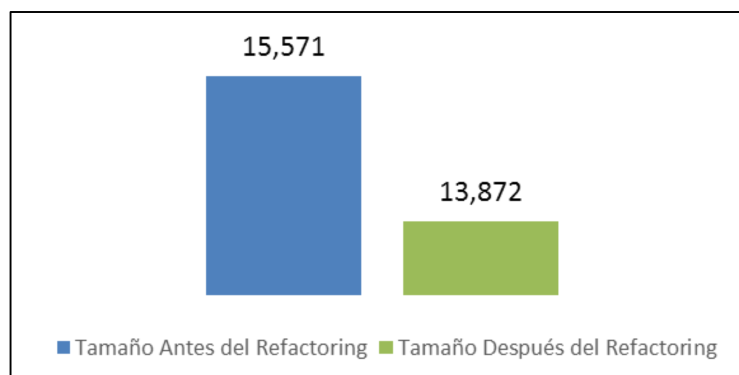
Universidad Nacional del Altiplano

Tabla N° 4.17: Comparación del tamaño antes y después del proceso de refactoring.

Tamaño Antes del Refactoring	Tamaño Después del Refactoring	Reducción del Tamaño
15,571	13,872	1,699

Elaboración: Propia.

Figura N° 4.46: Comparación del tamaño antes y después del proceso de refactoring.



Elaboración: Propia.

4.4.6. En cuanto a la Cobertura de Código

La Tabla N° 4.18, muestra que la cobertura alcanzada del caso de estudio es de 34.10% siendo el inicial 0%.

Tabla N° 4.18: Comparación de la cobertura de código antes y después del proceso de refactoring.

Cobertura Antes del Refactoring	Cobertura Después del Refactoring	Mejora de la Cobertura
0.00%	34.10%	34.10%

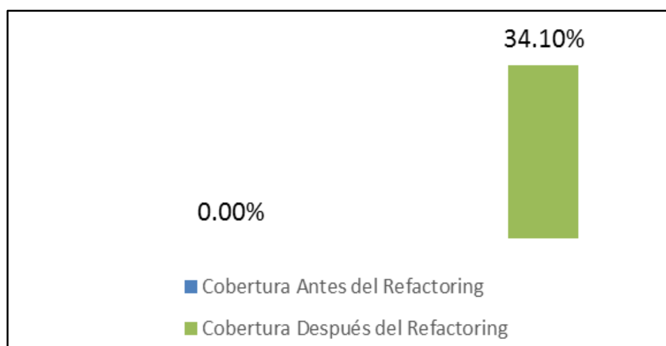
Elaboración: Propia.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Universidad Nacional del Altiplano

Figura N° 4.47: Comparación de la cobertura de código antes y después del proceso de refactoring.



Elaboración: Propia.

4.5. Prueba de hipótesis

La prueba de hipótesis se realizó con la prueba U de Mann-Whitney, debido a que los datos obtenidos no corresponden a una distribución normal.

La Tabla N° 4.19, muestra el número de incidencias (n° de bugs + n° de vulnerabilidades + n° code smells) de los paquetes Java que conforman el caso de estudio antes y después del proceso de refactoring desarrollado. Los datos proceden del resultado del análisis efectuado con SonarQube.

Tabla N° 4.19: Número de incidencias de los paquetes Java del caso de estudio antes y después del proceso de refactoring desarrollado.

Paquetes Java antes del proceso de refactoring	240	222	91	86	65	53	42	32	28	19	11	8	7	6	5	2	1	0
Paquetes Java después del proceso de refactoring	0	0	2	0	0	0	0	0	0	2	0	0	6	6	2	0	0	0
	1	1	0	0	2	0	2	0	0	1	3	1	2	1	0	0	5	0
	0	0	2	2	0	1	0	0	2	0	1	4	0	9	22	0	10	47
	2	1																

Elaboración: Propia.

Se desea conocer si el número de incidencias –que determina la calidad interna- varía con relación al proceso de refactoring desarrollado. Por tanto, las hipótesis de investigación se plantean de la siguiente forma:

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Universidad Nacional del Altiplano

- H_0: No existe diferencias en la calidad interna entre los paquetes Java antes y después del uso de los principios DRY y S.O.L.I.D. en el proceso de refactoring del caso de estudio (aplicación Web Java).
- H_1: Existe diferencias significativas en la calidad interna entre los paquetes Java antes y después del uso de los principios DRY y S.O.L.I.D. en el proceso de refactoring del caso de estudio (aplicación Web Java).

El nivel de significancia[26] considerado es $\alpha = 0.05$. La regla de decisión es si $p \leq 0.05$ se rechaza H_0.

El resultado del análisis de la prueba U de las dos muestras independientes (una con 18 paquetes y la otra con 56 paquetes) obtenido con el software SPSS se muestra en la Figura N° 4.48.

Figura N° 4.48: Resultados de la prueba de U efectuado con SPSS.

Prueba de Mann-Whitney

Rangos

	Paquetes Java	N	Rango promedio	Suma de rangos
Nro de Incidencias	Antes del proceso de refactoring	18	59,67	1074,00
	Después del proceso de refactoring	56	30,38	1701,00
	Total	74		

Estadísticos de prueba[a]

	Nro de Incidencias
U de Mann-Whitney	105,000
W de Wilcoxon	1701,000
Z	-5,217
Sig. asintótica (bilateral)	,000

a. Variable de agrupación: Paquetes Java

Elaboración: Propia.

[26] El nivel de significancia, es un nivel de la probabilidad de equivocarse y que fija de manera a priori el investigador (Hernández Sampieri, Fernández-Colado, & Baptista Lucio, 2006).

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Interpretación: Como puede apreciarse en la Figura N° 4.48, el estadígrafo de U de Mann-Whitney fue de 105,000 y el valor de p -Sig. Asintótica (bilateral)- es 0.000 por lo que se rechaza la hipótesis nula y se concluye que existen diferencias significativas en la calidad interna entre los paquetes Java antes y después del uso de los principios DRY y S.O.L.I.D. en el proceso de refactoring del caso de estudio (aplicación Web Java).

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

TESIS UNA - PUNO

Universidad Nacional del Altiplano

CONCLUSIONES

PRIMERA

Al término del proceso de refactoring del caso estudio -aplicación Web Java-, se observó la reducción del 84.75% de las incidencias halladas inicialmente, y su código duplicado disminuyó de 14.50% a 3.70%; estos resultados evidencian que el uso de los principios DRY y S.O.L.I.D. en el proceso de refactoring de una aplicación Web Java, repercuten positivamente en su calidad interna, respecto a los atributos de mantenibilidad, fiabilidad y seguridad.

SEGUNDA

La evaluación de la calidad interna del caso de estudio –aplicación Web Java-, permitió comprender su código, diseño y arquitectura. A la vez, permitió identificar las anomalías de su código fuente, para lo cual se usó la herramienta de análisis SonarQube que facilitó la obtención del número de vulnerabilidades, número de bugs, número de code smells, porcentaje código duplicado, porcentaje de cobertura de código y tamaño de la aplicación.

TERCERA

La definición de un proceso de refactoring para mejorar la calidad interna de una aplicación Web Java, permitió fijar cinco fases, las que son: analizar e identificar las incidencias, elaborar un plan de refactoring, desarrollar el refactoring, medir y evaluar los resultados de la calidad interna alcanzada.

CUARTA

El proceso de refactoring desarrollado sobre el caso de estudio, consistió en la realización de las cinco fases definidas, lo que conllevó a efectuar tareas como el refactoring de la estructura del proyecto y el refactoring de las capas de la aplicación Web Java. Al finalizar el proceso de refactoring, el número de anomalías halladas

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

TESIS UNA - PUNO

Universidad Nacional del Altiplano

inicialmente en el caso de estudio, disminuyó de 918 a 140, además, el porcentaje de código duplicado se redujo a 3.70%.

QUINTA

La evaluación de la medida en que el uso de los principios DRY y S.O.L.I.D. en el proceso de refactoring desarrollado afectó la calidad interna del caso de estudio, permitió validar el incremento de la calidad interna en cuanto a los atributos de mantenibilidad -medido por el número de code smells-, seguridad –medido por el número vulnerabilidades- y fiabilidad –medido por el número de bugs-, al ser comparado con su calidad interna inicial antes del proceso de refactoring efectuado. Los resultados obtenidos, muestran que el número de bugs disminuyó en 53.13%; el número de vulnerabilidades disminuyó en 100.00%; el número de code smells disminuyó en 92.53%; se eliminó el 10.80% del código duplicado; el tamaño de la aplicación se redujo en 1,699 líneas de código; y la cobertura de código se incrementó de 0.00% a 34.10%.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

RECOMENDACIONES

PRIMERA

Se recomienda efectuar un proceso de refactoring a las aplicaciones Web Java que evidencien problemas con su calidad interna, toda vez que el refactoring permite mejorar su estructura o diseño interno, de esta manera, su mantenimiento y extensión serán labores más llevaderas de realizar.

SEGUNDA

Se recomienda el uso de los principios DRY y S.O.L.I.D. en el refactoring y/o desarrollo de aplicaciones Web Java, con la finalidad de obtener aplicaciones con una adecuada calidad interna y que sean fáciles de adaptar a los cambios y exigencias del negocio.

TERCERA

Se recomienda el uso de las herramientas de análisis de código fuente en el refactoring y/o desarrollo de aplicaciones Web Java, porque permiten evaluar de forma sencilla su calidad interna; toda vez que los resultados facilitan la detección de anomalías, identificando los lugares del código fuente que requieren de alguna corrección o mejora.

CUARTA

Se recomienda afrontar un proceso de refactoring con la cobertura de código más alta posible, de esta manera, se evitará introducir errores no deseados, perder comportamiento o crear un mal comportamiento en las aplicaciones Web Java.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

REFERENCIAS

- Alonso, M., & Klaver, F. H. (2015). *Aplicación de un Proceso de Refactoring guiado por Escenarios y de Modificabilidad y Code Smells.* Tandil, Argentina: Universidad Nacional del Centro de la Provincia de Buenos Aires.
- Álvarez Caules, C. (2012). *Arquitectura Java Sólida.*
- Badeón Villanes, E. J. (2015). *Método para la evaluación de calidad de software basado en ISO/IEC 25000.* Lima, Perú: Universidad de San Martin de Porres.
- Bauer, C., & King, G. (2007). *Java Persistence with Hibernate.* Manning.
- Blé Jurado, C. (2010). *Diseño Ágil con TDD* (Primera ed.). www.iExpertos.com.
- Dai, N., Mandel, L., & Ryman, A. (2007). *Eclipse Web Tools Platform, Developing Java Web Applications.* Pearson Education, Inc.
- DBeaver. (30 de 06 de 2018). *DBeaver, Free Universal Database Tool*. Obtenido de https://dbeaver.io/
- DZone. (08 de 02 de 2018). *DZone*. Obtenido de https://dzone.com/articles/what-is-refactoring
- Feathers, M. C. (2004). *Working Effectively With Legacy Code.* Prentice Hall Professional Tecnical Reference.
- Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code.* Addison Wesley.
- Galicia, X. d. (2014). *I Jornada sobre calidad del producto software w ISO 25000.* Santiago de Compestela: 233 Grados de TI S.L.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (2003). *Patrones de Diseño, Elementos de software orientado a objetos reutilizable.* Pearson Educación.
- Garrido, A., Rossi, G., & Distante, D. (2007). *Model refactoring in web applications.* IEEE.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

- Gilfillan, I. (s.f.). *La Biblia MySQL.* Anaya.
- Hernández Sampieri, R., Fernández-Colado, C., & Baptista Lucio, P. (2006). *Metodología de la Investigación* (Cuarta ed.). México: McGraw-Hill.
- Hunt, A., & Thomas, D. (2000). *The Pragmatic Programmer.* Addison Wesley Longman, Inc.
- ISO/IEC 25000:2014. (15 de 08 de 2017). *ISO - International Organization for Standardization*. Obtenido de https://www.iso.org/obp/ui/es/#iso:std:iso-iec:25000:ed-2:v1:en
- Jendrock, E., Cervera-Navarro, R., Evans, I., Haase, K., & Markito, W. (2014). *Java Platform, Enterprise Edition The Java EE Tutorial, Release 7.* Oracle.
- Kaczanowski, T. (2013). *Practical Unit Testing with JUnit and Mockito.* kaczanowscy.pl Tomasz Kaczanowski.
- Kniberg, H. (2015). *Scrum and XP from the trenches.* C4Media.
- Larman, C. (2003). *UML y Patrones, Una introducción al análisis y diseño orientado a objetos y al proceso unificado* (Segunda ed.). España: Pearson Educación, S.A.
- Marinescu, F. (2002). *EJB Design Patterns.* John Wiley & Sons, Inc.
- Martin, R. C. (2012). *Código Limpio, Manual de estilo para el desarrollo ágil de software.* ANAYA.
- Martin, R. C., & Kriens, P. (2012). *Java Application Architecture - Modularity Patterns With Examples Using OSGi.*
- Martin, R. C., & Martin, M. (2006). *Agile Principles, Patterns, and Practices in C#.* Prentice Hall.
- McLean Hall, G. (2014). *Adaptive Code via C# Agile coding with design patterns and SOLID principles.* Microsoft Press.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

- Microsoft. (2009). *Microsoft Application Architecture Guide (Patterns & Practices)* (2nd Edition ed.). Microsoft Corporation.
- modelio. (30 de 06 de 2018). *modelio, the open source modeling environment*. Obtenido de https://www.modelio.org/
- ObjectAid. (24 de 09 de 2018). *ObjectAid*. Obtenido de http://www.objectaid.com/
- Pérez García, F. J. (2011). *Refactoring planning for design smell correction in object-oriented software.* Universidad de Valladolid, Escuela Técnica Superior de Ingeniería Informática, Departamento de Informática.
- Portal ISO 25000. (11 de 10 de 2017). *Portal ISO 25000*. Obtenido de http://iso25000.com/
- Pressman, R. S. (2010). *Ingeniería del Software, Un enfoque práctico* (Séptima ed.). Mc Graw Hill.
- Sommerville, I. (2011). *Ingeniería de Software* (9na edición ed.). México: Pearson Edicación.
- SonarQube. (10 de 11 de 2018). *SonarQube*. Obtenido de https://docs.sonarqube.org/7.4/user-guide/concepts/
- SonarSource S.A. (29 de 06 de 2017). *sonarqube*. Obtenido de https://www.sonarqube.org/
- The Apache Software Foundation. (30 de 09 de 2018). *Apache Maven Project*. Obtenido de https://maven.apache.org/what-is-maven.html
- UncleBob. (10 de 10 de 2017). *PrinciplesOfOod*. Obtenido de http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
- Walls, G. (2015). *Spring in Action* (Fourth ed.). Manning.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Universidad Nacional del Altiplano

ANEXOS

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Anexo A. Análisis de aplicaciones Web Java

Tabla N° A.1: Análisis de cinco aplicaciones Web Java, efectuado con SonarQube.

Aplicación Web Java	Lugar de Desarrollo	Mantenibilidad (N° de code smells)	fiabilidad (N° de bugs)	Seguridad (N° de vulnerabilidades)	% Cobertura de código	% Código duplicado	Tamaño en líneas de código
Aplicación Web Java de Gestión Notarial	Juliaca	2,626	1,392	450	0.0 %	12.8 %	43, 608
Aplicación Web Java de Gestión de Créditos y Ahorros para Cooperativas.	Juliaca	1, 126	607	150	0.0 %	8.0 %	19, 447
Aplicación Web Java de Gestión Abastecimiento y Almacenes para Municipalidades	Puno	1, 821	699	203	0.0 %	28.4 %	21, 418
Aplicación Web Java para el Reporte de Operaciones de las Cooperativas.	Lima	2, 871	559	20	0.0 %	28.1 %	37, 929
Aplicación Web Java (Caso de Estudio)	Puno	669	192	57	0.0 %	14.5 %	15, 571

Elaboración: Propia.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Anexo B. Archivo pom.xml del proyecto Maven del caso de estudio obtenido después del proceso de refactoring

En la actividad de modularización del proyecto se obtuvo el archivo pom.xml mostrado en la Figura N° B.1, este archivo muestra la configuración necesaria para que SonarQube obtenga la cobertura de código de los módulos Maven del caso de estudio.

Figura N° B.1: Archivo pom.xml del proyecto Maven del caso de estudio.

```
*2-5-iteracion-ciap-2.0.1.7/pom.xml
1   <?xml version="1.0" encoding="UTF-8"?>
2   <project xmlns="http://maven.apache.org/POM/4.0.0"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6     <groupId>com.cia</groupId>
7     <artifactId>cia</artifactId>
8     <version>2.0.1.7</version>
9     <packaging>pom</packaging>
10    <name>cia</name>
11    <properties>
28    <repositories>
42    <dependencies>
339   <modules>
346   <build>
347     <plugins>
348       <plugin>
349         <groupId>org.jacoco</groupId>
350         <artifactId>jacoco-maven-plugin</artifactId>
351         <version>${jacoco.version}</version>
352         <executions>
353           <execution>
354             <id>agent-for-ut</id>
355             <goals>
356               <goal>prepare-agent</goal>
357             </goals>
358             <configuration>
359               <append>true</append>
360               <destFile>${sonar.jacoco.reportPaths}</destFile>
361             </configuration>
362           </execution>
363
364           <execution>
365             <id>agent-for-it</id>
366             <phase>package</phase>
367             <goals>
368               <goal>prepare-agent-integration</goal>
369             </goals>
370             <configuration>
371               <append>true</append>
372               <destFile>${sonar.jacoco.itReportPath}</destFile>
373             </configuration>
374           </execution>
375         </executions>
376       </plugin>
377     </plugins>
378   </build>
379 </project>
```

Elaboración: Propia.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Universidad Nacional del Altiplano

Anexo C. Archivo de propiedades para el análisis con SonarQube

Para llevar a cabo el análisis del código fuente del caso de estudio con la herramienta sonar-scanner se utilizó el archivo indicado en la Figura N° C.1 se presenta el archivo de propiedades necesario para realizar el análisis con sonar-scanner, este archivo precisa los módulos, librerías y los reportes de las pruebas realizadas con JUnit.

Figura N° C.1: Archivo sonar-project.properties del caso de estudio.

```
# must be unique in a given SonarQube instance
sonar.projectKey=cia-2.0.1.7:cia-2.0.1.7
# this is the name and version displayed in the SonarQube UI. Was mandatory prior to SonarQube 6.1.
sonar.projectName=Caso de Estudio 2.0.1.7
sonar.projectVersion=2.0.1.7

# Path is relative to the sonar-project.properties file. Replace "\" by "/" on Windows.
# Since SonarQube 4.2, this property is optional if sonar.modules is set.
# If not set, SonarQube starts looking for source code from the directory containing
# the sonar-project.properties file.
sonar.sources=cia-business-model/src/main, cia-business-service/src/main, cia-common/src/main, cia-persistence/src/main, cia-web/src/main/java
sonar.jacoco.reportPaths=target/jacoco.exec
sonar.junit.reportPaths=cia-business-model/target/surefire-reports, cia-business-service/target/surefire-reports, cia-persistence/target/surefire-reports, cia-web/target/surefire-reports
sonar.java.binaries=cia-common/target/classes, ../../cia-business-model/target/classes, ../../cia-business-service/target/classes, ../../cia-persistence/target/classes, ../../cia-web/target/classes
sonar.java.libraries=cia-web/target/cia-web-2.5/WEB-INF/lib

# Encoding of the source code. Default is default system encoding
#sonar.sourceEncoding=UTF-8
```

Elaboración: Propia.

Repositorio Institucional UNA-PUNO

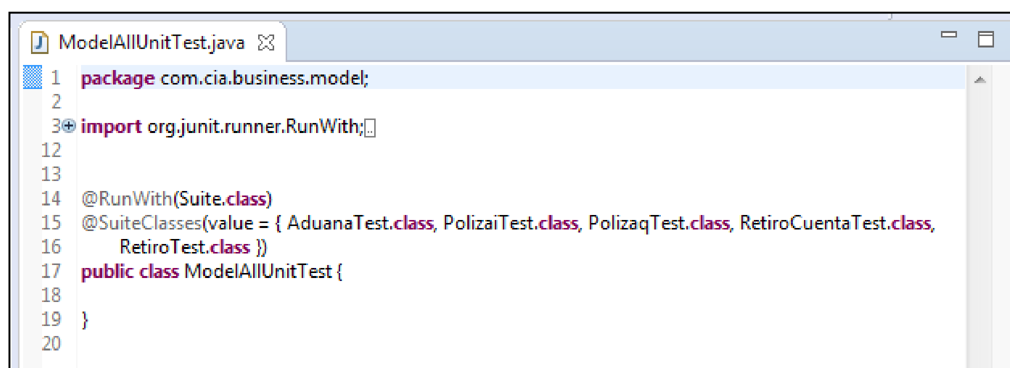
No olvide citar esta tesis

Anexo D. Pruebas unitarias y de integración en el caso de estudio

Las pruebas escritas en el caso de estudio tuvieron sus particularidades según las capas en las que se implementaron. A continuación, se muestran algunos ejemplos de las pruebas unitarias y de integración desarrollados en el proceso de refactoring.

- La Figura N° D.1, presenta el código fuente para ejecutar uno de los suites de pruebas unitarias desarrolladas.

Figura N° D.1: Ejemplo de un Suite de pruebas en la capa de lógica de negocios del caso de estudio.



Elaboración: Propia.

- La Figura N° D.2, muestra un ejemplo de prueba de unitaria en la capa de lógica de negocios.

Universidad Nacional del Altiplano

Figura N° D.2: Ejemplo de una prueba unitaria con JUnit y Mockito en el caso de estudio.

```
ControlPasoFacadeImplTest.java

3⊕ import static org.junit.Assert.assertEquals;
26
27 @RunWith(MockitoJUnitRunner.class)
28 public class ControlPasoFacadeImplTest {
29
30⊖    @InjectMocks
31     private ControlPasoFacadeImpl controlPasoFacade;
32
33⊖    @Mock
34     private ControlPasoDAO controlDAO;
35
36⊖    @Test
37     public void testCreateFindControlPaso() throws CiaAppException {
38         ControlPaso control = new ControlPaso();
39         ControlPasoPK id = new ControlPasoPK();
40         id.setRetiroid("A181020317000008Y");
41         id.setCtlseq(1);
42         control.setId(id);
43         control.setAduaCod("262");
44         control.setLlegadaHoras(new BigDecimal("1.2"));
45
46⊖        when(controlDAO.findByRetiroId(Mockito.anyString())).thenAnswer(new Answer<List<ControlPaso>>() {
47⊖            @Override
48             public List<ControlPaso> answer(InvocationOnMock invocation) throws Throwable {
49                 String retiroId = (String) invocation.getArguments()[0];
50                 List<ControlPaso> controles = new LinkedList<>();
51                 control.getId().setRetiroid(retiroId);
52                 controles.add(control);
53                 return controles;
54             }
55         });
56
57         controlPasoFacade.createControlPaso(control);
58
59         ControlPaso controlResult = new ControlPaso();
60         controlResult = controlPasoFacade.findControlesByRetiroId(control.getId().getRetiroid()).get(0);
61         assertNotNull(controlResult);
62         assertEquals("A181020317000008Y", controlResult.getId().getRetiroid());
63         assertEquals(1, controlResult.getId().getCtlseq());
64         verify(controlDAO, times(1)).persist(Mockito.any(ControlPaso.class));
65         verify(controlDAO, times(1)).findByRetiroId(Mockito.anyString());
66
67     }
68
69 }
```

Elaboración: Propia.

- La Figura N° D.3, muestra el uso de Spring JUnit para ejecutar pruebas de integración con la capa de persistencia.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Figura N° D.3: Ejemplo de una prueba de integración en la capa de lógica de negocios del caso de estudio.

```
RetiroFacadeIntegrationTest.java
1  package com.cia.business.service.retiro.impl;
2
3⊕ import static org.junit.Assert.assertEquals;
18
19 @RunWith(SpringJUnit4ClassRunner.class)
20 @ContextConfiguration("classpath:applicationContext.xml")
21 @TransactionConfiguration(defaultRollback = true)
22 @Transactional
23 public class RetiroFacadeIntegrationTest {
24
25⊖    @Resource
26     private RetiroFacade retiroFacade;
27
28⊖    @Test
29     @Rollback(true)
30     public void testFindByIdRetiro() {
31         final String retiroId = "A181020317000008Y";
32         Retiro retiroResult = retiroFacade.findRetiroById(retiroId);
33         assertNotNull(retiroResult);
34         assertEquals(retiroId, retiroResult.getRetiroId());
35     }
36
37 }
38
```

Elaboración: Propia.

- La Figura N° D.4, muestra el uso de Spring JUnit, resaltando que los test que realizan alguna interacción con la base de datos se marcan con la etiqueta *@Rollback* para no alterar la base de datos del caso de estudio.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Figura N° D.4: Ejemplo de una prueba de integración en la capa de persistencia del caso de estudio.

```java
RetiroDAOJPAImplIntegrationTest.java
1   package com.cia.persistence.retiro.dao.jpa;
2
3⊕  import static org.junit.Assert.assertEquals;
28
29  @RunWith(SpringJUnit4ClassRunner.class)
30  @ContextConfiguration("classpath:applicationContext.xml")
31  @TransactionConfiguration(defaultRollback = true)
32  @Transactional
33  public class RetiroDAOJPAImplIntegrationTest {
34
35⊖     @Resource
36      private RetiroDAO retiroDAO;
37
40⊕     public void testFindUltimoRetiroIdByDAM() {
50
51⊖     @Test
52      @Rollback(true)
53      public void testFindRetirosBy() throws ParseException {
54          SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd");
55          Date fechaInicial = formatter.parse("2017-10-25");
56          Date fechaFinal = formatter.parse("2017-10-25");
57
58          RetiroDtoParametro parametro = new RetiroDtoParametro();
59          parametro.setOficial(null);
60          parametro.setPeriodo(new Periodo(fechaInicial, fechaFinal));
61          parametro.setRmteRuc(null);
62          parametro.setDstarioRuc(null);
63          parametro.setEmprRuc(null);
64          parametro.setAduana(null);
65          parametro.setPuesto(null);
66          parametro.setAnho(null);
67          parametro.setVehiPlaca(null);
68
69          GenericDto dynamicQuery = retiroDAO.findRetirosBy(parametro);
70
71          assertNotNull(dynamicQuery);
72          assertEquals("Resultado :", 94, dynamicQuery.getResultList().size());
73      }
74
77⊕     public void testFindRetirosDetalladosBy() throws IOException, ParseException {
98
101⊕    public void testFindAndExportRetirosDetalladosBy() throws IOException, ParseException {
124
125 }
```

Elaboración: Propia.

- La Figura N° D.5, muestra el uso de PowerMock para conseguir doblar los métodos estáticos y algunos objetos propios del framework JSF.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Figura N° D.5: Ejemplo de una prueba unitaria en la capa de presentación del caso de estudio.

```
*RetiroEditarBeanTest.java
46
47  @RunWith(PowerMockRunner.class)
48  @PrepareForTest({ FacesContext.class, FacesUtils.class })
49  public class RetiroEditarBeanTest {
50
52⊕     private RetiroEditarBean retiroEditarBean;
53
55⊕     private RetiroFacade retiroFacade;
56
58⊕     private RetiroCuentaFacade retiroCuentaFacade;
59
61⊕     private FacesMessage facesMessage;
62
64⊕     private FacesContext facesContext;
65
67⊕     private ExternalContext externalContext;
68
70⊕     private HttpServletRequest httpServletRequestt;
71
73⊕     public void setUp() throws Exception {
81
83⊕     public void testFindRetiroNoEditable() throws Throwable {
128
130⊕    public void testFindRetiroEditable() throws Exception {
183
185⊕    public void testSaveRetiroDetalleOk() throws CiaAppException {
208
210⊕    public void testSaveRetiroDetalleErrorSinSaldo() throws CiaAppException {
225
226⊖    @Test
227     public void testEditRetiroEditable() throws Exception {
228         AppBean appBean = new AppBean();
229         String retiroId = "A181020318000002C";
230         Retiro retiro = new Retiro();
231         retiro.setRetiroId(retiroId);
232         Login usuario = new Login();
233         usuario.setLoginId("anonimo");
234         retiroEditarBean.setUsuario(usuario);
235         retiroEditarBean.setAppBean(appBean);
236
237         when(retiroFacade.isEditable(Mockito.any(Retiro.class), Mockito.anyString())).thenReturn(true);
238
239         retiroEditarBean.editRetiro(retiro);
240         String result = retiroEditarBean.getAppBean().getIncludePage();
241
242         verify(retiroFacade, times(1)).isEditable(Mockito.any(Retiro.class), Mockito.anyString());
243         assertEquals("/pages/retiro/retiro_reg.xhtml", result);
244
245     }
```

Elaboración: Propia.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Anexo E. Análisis de clases Java en las iteraciones del proceso de refactoring

Tabla N° E.1: Análisis de las clases Java antes del refactoring del módulo 1 de la capa de lógica de negocios.

Aplicación Web (Caso de Estudio - Iteracion 3) Módulo: "cia-business-model" / Clases	Líneas de código	N° de bugs	N° de Vulnerabilities	N° de Code smells	Total de Incidencias	Cobertura de código (%)	Código duplicado (%)
com.cia.business.model.poliza.Series	202	10	6	4	20	78.00%	11.90%
com.cia.business.model.poliza.SeriesiSaldoAB	17	0	6	3	9	100.00%	0.00%
com.cia.business.model.poliza.Poliza	213	4	0	2	6	79.10%	31.20%
com.cia.business.model.poliza.SeriesqSaldoAB	15	0	4	2	6	100.00%	0.00%
com.cia.business.model.retiro.RetiroControl	56	0	4	2	6	25.00%	0.00%
com.cia.business.model.dto.AsignacionDTO	178	0	0	3	3	25.90%	6.80%
com.cia.business.model.retiro.Retiro	460	0	0	3	3	33.30%	3.50%
com.cia.business.model.common.Asignacion	133	0	0	2	2	25.80%	9.60%
com.cia.business.model.common.PuestoControl	78	0	0	2	2	0.615	43.00%
com.cia.business.model.poliza.PolizaBloqueoPK	91	0	0	2	2	49.50%	26.50%
com.cia.business.model.poliza.Polizai	199	0	0	2	2	46.70%	72.00%
com.cia.business.model.poliza.Polizaq	215	0	0	2	2	43.20%	65.30%
com.cia.business.model.retiro.RetiroControlPK	75	0	0	2	2	4.50%	22.00%
com.cia.business.model.common.Aduana	89	0	0	1	1	66.70%	37.70%
com.cia.business.model.common.AduanaDuaPK	55	0	0	1	1	15.10%	0.00%
com.cia.business.model.common.Ctlcorre	69	0	0	1	1	19.20%	0.00%
com.cia.business.model.common.Login	152	0	1	0	1	81.10%	0.00%
com.cia.business.model.common.MercanciaSensible	77	0	0	1	1	0.063	0.00%
com.cia.business.model.common.Parametro	94	0	0	1	1	6.90%	17.50%
com.cia.business.model.controlpaso.ControlPaso	170	0	0	1	1	58.00%	9.60%
com.cia.business.model.dto.ColumnInfoDTO	35	0	0	1	1	33.30%	0.00%
com.cia.business.model.dto.ConductorRetiroDTO	70	0	0	1	1	0.65	0.00%
com.cia.business.model.dto.DamRetiroDTO	94	0	0	1	1	0	0.00%
com.cia.business.model.dto.PuestoControlDTO	79	0	0	1	1	30.80%	0.00%
com.cia.business.model.dto.querys.ControlPasoResumenDTO	45	0	0	1	1	0	50.00%
com.cia.business.model.dto.querys.RetiroResumenTipoDTO	37	0	0	1	1	0.00%	60.00%
com.cia.business.model.dto.RetiroDetalleDTO	207	0	0	1	1	0.00%	0.00%
com.cia.business.model.poliza.PolizaiPK	64	0	0	1	1	65.00%	61.00%
com.cia.business.model.poliza.PolizaPK	59	0	0	1	1	59.10%	64.50%
com.cia.business.model.poliza.PolizaqPK	62	0	0	1	1	65.00%	62.50%
com.cia.business.model.poliza.Seriesi	97	0	0	1	1	60.60%	65.20%
com.cia.business.model.poliza.SeriesiPK	64	0	0	1	1	12.50%	78.60%
com.cia.business.model.poliza.SeriesPK	62	0	0	1	1	40.00%	64.20%
com.cia.business.model.poliza.Seriesq	97	0	0	1	1	60.60%	64.70%
com.cia.business.model.poliza.SeriesqPK	60	0	0	1	1	12.80%	62.00%
com.cia.business.model.poliza.SeriesSaldo	190	0	0	1	1	32.40%	11.50%
com.cia.business.model.common.AduanaDua	49	0	0	0	0	40.00%	0.00%
com.cia.business.model.common.Contribuyente	34	0	0	0	0	0.00%	0.00%
com.cia.business.model.common.CtlcorrePK	46	0	0	0	0	0.455	0.00%
com.cia.business.model.common.ParametroPK	43	0	0	0	0	0.069	0.00%
com.cia.business.model.common.PControlPK	61	0	0	0	0	21.10%	0.00%
com.cia.business.model.common.Ubigeo	53	0	0	0	0	53.30%	0.00%
com.cia.business.model.controlpaso.ControlPasoPK	45	0	0	0	0	33.30%	0.00%
com.cia.business.model.Login	128	0	0	0	0	0.836	0.00%
com.cia.business.model.poliza.PolizaBloqueo	106	0	0	0	0	0.313	0.00%
com.cia.business.model.retiro.RetiroDetalle	273	0	0	0	0	45.30%	0.00%
com.cia.business.model.retiro.RetiroDetallePK	60	0	0	0	0	34.10%	0.00%

Elaboración: Propia.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Tabla N° E.2: Análisis de las clases Java antes del refactoring del módulo 2 de la capa de lógica de negocios.

Aplicación Web (Caso de Estudio - Iteracion 4) Módulo: "cia-business-service" — Clases	Líneas de código	N° de bugs	N° de Vulnerabilities	N° de Code smells	Total de Incidencias	Cobertura de código (%)	Código duplicado (%)
com.cia.business.service.util.webscraping.DamHelper	189	4	1	49	54	90.00%	0.00%
com.cia.business.service.retiro.impl.RetiroFacadeImpl	249	8	3	12	23	29.90%	0.00%
com.cia.business.service.util.webscraping.SunatDualLevanteHelper	105	3	2	17	22	0.00%	0.00%
com.cia.business.service.common.impl.LoginFacadeImpl	104	2	2	6	10	19.40%	0.00%
com.cia.business.service.util.webscraping.SunatIntranetHelper	45	2	0	8	10	0.00%	0.00%
com.cia.business.service.poliza.impl.PolizaFacadeImpl	95	4	2	3	9	52.50%	0.00%
com.cia.business.service.common.impl.ParametroFacadeImpl	119	8	0	0	8	8.00%	0.00%
com.cia.business.service.util.webscraping.EmpresaTransporteHelper	36	2	1	5	8	69.00%	0.00%
com.cia.business.service.retiro.RetiroFacade	40	0	0	6	6		0.00%
com.cia.business.service.controlpaso.ControlPasoConsultaFacade	6	0	0	2	2		0.00%
com.cia.business.service.retiro.RetiroConsultaFacade	6	0	0	2	2		0.00%
com.cia.business.service.common.ConductorFacade	5	0	0	1	1		0.00%
com.cia.business.service.common.impl.AduanaFacadeImpl	42	0	0	1	1	90.90%	0.00%
com.cia.business.service.common.impl.ConductorFacadeImpl	23	0	0	1	1	85.70%	0.00%
com.cia.business.service.retiro.RetiroDetalleFacade	6	0	0	1	1		0.00%
com.cia.business.service.common.AduanaFacade	12	0	0	0	0		0.00%
com.cia.business.service.common.AsignacionFacade	15	0	0	0	0		0.00%
com.cia.business.service.common.impl.AsignacionFacadeImpl	57	0	0	0	0	85.70%	0.00%
com.cia.business.service.common.impl.UbigeoFacadeImpl	26	0	0	0	0	57.10%	0.00%
com.cia.business.service.common.LoginFacade	14	0	0	0	0		0.00%
com.cia.business.service.common.ParametroFacade	20	0	0	0	0		0.00%
com.cia.business.service.common.UbigeoFacade	7	0	0	0	0		0.00%
com.cia.business.service.controlpaso.impl.ControlPasoConsultaFacadeImp	22	0	0	0	0	66.70%	0.00%
com.cia.business.service.poliza.impl.PolizaBloqueoFacadeImpl	46	0	0	0	0	82.40%	0.00%
com.cia.business.service.poliza.PolizaBloqueoFacade	12	0	0	0	0		0.00%
com.cia.business.service.poliza.PolizaFacade	18	0	0	0	0		0.00%
com.cia.business.service.retiro.impl.RetiroConsultaFacadeImpl	22	0	0	0	0	66.70%	0.00%
com.cia.business.service.retiro.impl.RetiroDetalleFacadeImpl	23	0	0	0	0	66.70%	0.00%

Elaboración: Propia.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Tabla N° E.3: Análisis de las clases Java antes del refactoring de la capa de persistencia.

Aplicación Web (Caso de Estudio - Iteracion 5) Módulo: "cia-persistence" Clase	Líneas de código	N° de bugs	N° de Vulnerabilities	N° de Code smells	Total de Incidencias	Cobertura de código (%)	Código duplicado (%)
com.cia.persistence.dao.retiro.jpa.RetiroDAOJPAImpl	268	3	0	22	25	7.70%	26.00%
com.cia.persistence.dao.common.jpa.DynamicQuery	105	8	3	11	22	37.30%	0.00%
com.cia.persistence.dao.jdbc.ControlPasoResumenDAOJDBCImpl	57	4	0	5	9	3.10%	0.00%
com.cia.persistence.dao.controlpaso.jpa.ControlPasoDAOJPAImpl	131	2	0	6	8	64.80%	7.80%
com.cia.persistence.dao.jdbc.RetiroResumenDAOJDBCImpl	52	3	0	3	6	3.60%	0.00%
com.cia.persistence.dao.retiro.RetiroDAO	16	0	0	4	4		0.00%
com.cia.persistence.dao.common.jpa.GenericDAOJPAImpl	58	1	0	2	3	47.60%	0.00%
com.cia.persistence.dao.poliza.jpa.PolizaiDAOJPAImpl	64	1	0	2	3	2.60%	64.80%
com.cia.persistence.dao.poliza.jpa.PolizaqDAOJPAImpl	64	1	0	2	3	2.60%	64.80%
com.cia.persistence.dao.retiro.jpa.RetiroDetalleDAOJPAImpl	96	1	0	2	3	6.80%	0.00%
com.cia.persistence.dao.common.GenericDAO	12	0	0	2	2		0.00%
com.cia.persistence.dao.controlpaso.ControlPasoResumenDAO	6	0	0	2	2		0.00%
com.cia.persistence.dao.poliza.jpa.SeriesiDAOJPAImpl	54	0	0	2	2	100.00%	41.10%
com.cia.persistence.dao.poliza.jpa.SeriesqDAOJPAImpl	57	0	0	2	2	100.00%	34.70%
com.cia.persistence.dao.retiro.RetiroResumenDAO	6	0	0	2	2		0.00%
com.cia.persistence.dao.common.jpa.AduanaDAOJPAImpl	45	0	0	1	1	100.00%	0.00%
com.cia.persistence.dao.common.jpa.LoginDAOJPAImpl	36	1	0	0	1	100.00%	0.00%
com.cia.persistence.dao.common.jpa.UbigeoDAOJPAImpl	28	0	0	1	1	100.00%	0.00%
com.cia.persistence.dao.asignacion.AsignacionDAO	13	0	0	0	0		0.00%
com.cia.persistence.dao.common.AduanaDAO	11	0	0	0	0		0.00%
com.cia.persistence.dao.common.ConductorRetiroDAO	5	0	0	0	0		0.00%
com.cia.persistence.dao.common.CtlCorreDAO	6	0	0	0	0		0.00%
com.cia.persistence.dao.common.jpa.AsignacionDAOJPAImpl	46	0	0	0	0	100.00%	0.00%
com.cia.persistence.dao.common.jpa.ConductorRetiroDAOJPAImpl	18	0	0	0	0	100.00%	0.00%
com.cia.persistence.dao.common.jpa.CtlCorreDAOJPAImpl	19	0	0	0	0	100.00%	0.00%
com.cia.persistence.dao.common.jpa.MercanciaSensibleDAOJPAImp	21	0	0	0	0	100.00%	0.00%
com.cia.persistence.dao.common.jpa.ParametroDAOJPAImpl	111	0	0	0	0	8.10%	0.00%
com.cia.persistence.dao.common.LoginDAO	8	0	0	0	0		0.00%
com.cia.persistence.dao.common.MercanciaSensibleDAO	6	0	0	0	0		0.00%
com.cia.persistence.dao.common.ParametroDAO	20	0	0	0	0		0.00%
com.cia.persistence.dao.common.UbigeoDAO	6	0	0	0	0		0.00%
com.cia.persistence.dao.controlpaso.ControlPasoDAO	14	0	0	0	0		0.00%
com.cia.persistence.dao.poliza.jpa.PolizaBloqueoDAOJPAImpl	32	0	0	0	0	100.00%	0.00%
com.cia.persistence.dao.poliza.PolizaBloqueoDAO	10	0	0	0	0		0.00%
com.cia.persistence.dao.poliza.PolizaiDAO	9	0	0	0	0		0.00%
com.cia.persistence.dao.poliza.PolizaqDAO	9	0	0	0	0		0.00%
com.cia.persistence.dao.poliza.SeriesiDAO	9	0	0	0	0		0.00%
com.cia.persistence.dao.poliza.SeriesqDAO	9	0	0	0	0		0.00%
com.cia.persistence.dao.retiro.jpa.RetiroDetalleDtoDAOJPAImpl	25	0	0	0	0	12.50%	0.00%
com.cia.persistence.dao.retiro.RetiroDetalleDAO	10	0	0	0	0		0.00%
com.cia.persistence.dao.retiro.RetiroDetalleDtoDAO	7	0	0	0	0		0.00%

Elaboración: Propia.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Universidad Nacional del Altiplano

Tabla N° E.4: Análisis de las clases Java antes del refactoring de la capa de presentación.

Aplicación Web (Caso de Estudio - Iteracion 6) Módulo: "cia-web" Clases	Líneas de código	N° de bugs	N° de Vulnerabilities	N° de Code smells	Total de Incidencias	Cobertura de código (%)	Código duplicado (%)
com.cia.presentation.util.UtilsBean	365	12	1	35	48	0.00%	0.00%
com.cia.presentation.retiro.controller.RetiroRegistroBean	631	19	0	22	41	13.60%	0.00%
com.cia.presentation.retiro.controller.RetiroDAMBean	544	16	0	6	22	44.10%	6.30%
com.cia.presentation.retiro.controller.RetiroDamTrazaBean	281	11	0	8	19	0.00%	11.60%
com.cia.presentation.common.controller.LoginManagedBean	176	1	1	12	14	30.30%	0.00%
com.cia.presentation.asignacion.controller.AsignacionRegistroBean	315	5	0	4	9	0.00%	0.00%
com.cia.presentation.common.controller.LoginAdmBean	168	6	1	2	9	0.00%	0.00%
com.cia.presentation.poliza.controller.PolizaBloqueoAdmBean	217	6	0	3	9	0.00%	8.00%
com.cia.presentation.common.controller.AppManagedBean	172	4	2	2	8	0.00%	0.00%
com.cia.presentation.controlpaso.controller.ControlPasoConsultaResumenBean	169	6	0	1	7	0.00%	34.80%
com.cia.presentation.retiro.controller.RetiroConsultaResumenBean	160	6	0	1	7	0.00%	37.10%
com.cia.presentation.retiro.controller.RetiroEditarBean	166	6	0	1	7	64.40%	0.00%
com.cia.presentation.controlpaso.controller.ControlPasoAdmBean	110	3	0	3	6	0.00%	0.00%
com.cia.presentation.common.controller.DamLevanteBean	105	5	0	0	5	0.00%	0.00%
com.cia.presentation.common.controller.LoginRegistroBean	106	3	0	2	5	0.00%	15.70%
com.cia.presentation.controlpaso.controller.ControlPasoConsultaBean	166	3	0	2	5	0.00%	35.30%
com.cia.presentation.controlpaso.controller.ControlPasoReporteBean	136	2	0	3	5	0.00%	12.60%
com.cia.presentation.controlpaso.controller.ControlRetiroBean	178	4	0	1	5	53.50%	0.00%
com.cia.presentation.retiro.controller.RetiroBean	210	3	0	2	5	0.00%	0.00%
com.cia.presentation.retiro.controller.RetiroConsultaDetalleBean	189	3	0	2	5	0.00%	18.80%
com.cia.presentation.asignacion.controller.AsignacionConsultaBean	119	2	0	2	4	0.00%	0.00%
com.cia.presentation.poliza.controller.PolizaDesbloquearBean	111	3	0	1	4	0.00%	15.00%
com.cia.presentation.retiro.controller.RetiroAdmBean	97	2	0	2	4	0.00%	0.00%
com.cia.presentation.retiro.controller.RetiroConsultaBean	171	3	0	1	4	0.00%	68.10%
com.cia.presentation.retiro.controller.RetiroConsultaDetalladaBean	172	3	0	1	4	0.00%	68.10%
com.cia.presentation.common.controller.EstablecerPControlBean	96	2	0	1	3	0.00%	0.00%
com.cia.presentation.controlpaso.controller.ControlPasoBean	197	3	0	0	3	33.60%	0.00%
com.cia.presentation.poliza.controller.PolizaBloquearBean	97	2	0	1	3	0.00%	20.50%
com.cia.presentation.retiro.controller.RetiroConsultaPlacaBean	89	1	0	2	3	0.00%	17.60%
com.cia.presentation.common.controller.MenuBean	49	1	0	1	2	0.00%	32.40%
com.cia.presentation.util.UtilReportBean	18	0	0	1	1	0.00%	0.00%

Elaboración: Propia.

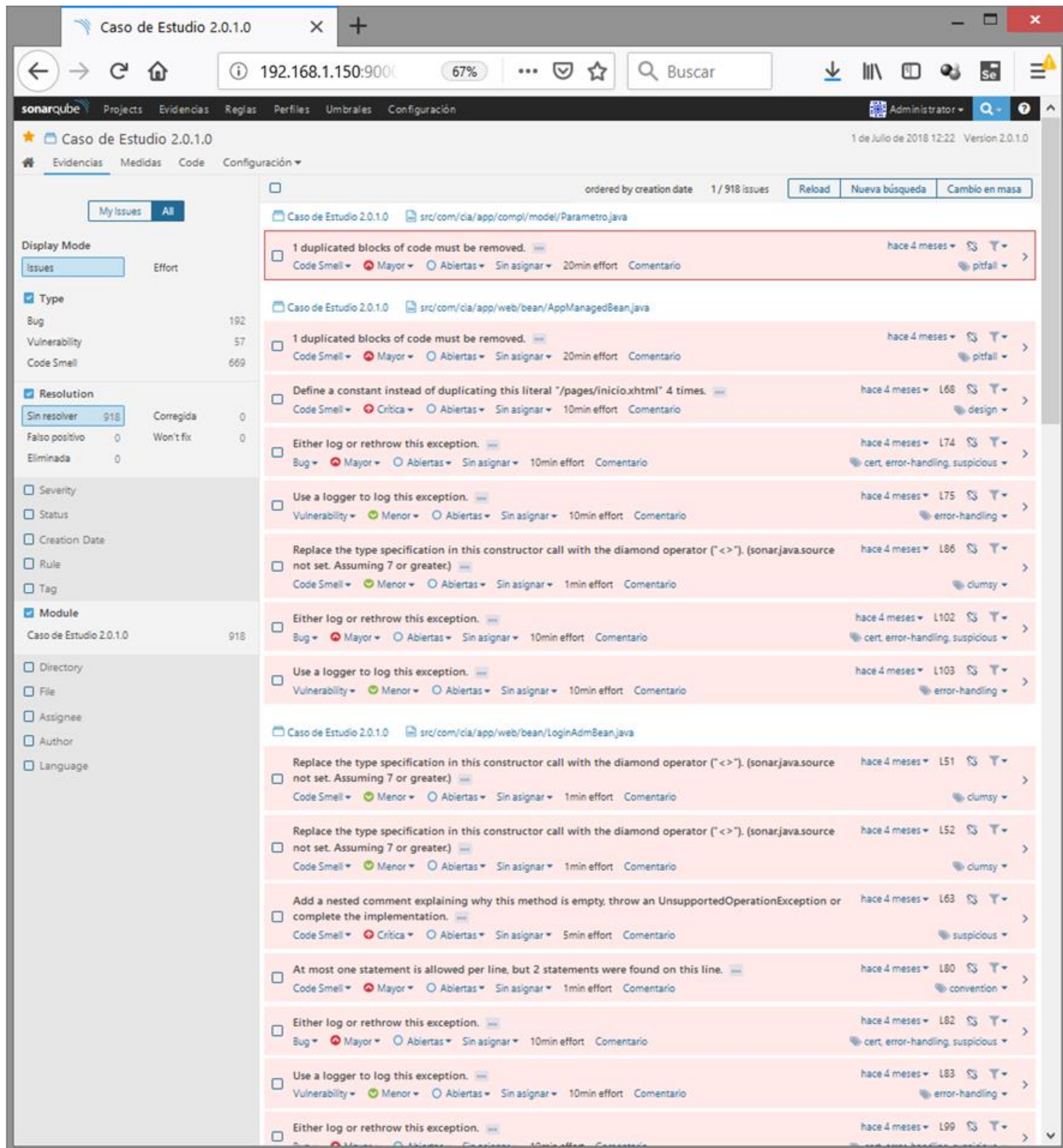
Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

ANEXO F. Reportes de SonarQube del análisis efectuado al caso de estudio antes del proceso de refactoring

A continuación, se presentan algunos reportes del software SonarQube del análisis realizado al caso de estudio antes del proceso de refactoring.

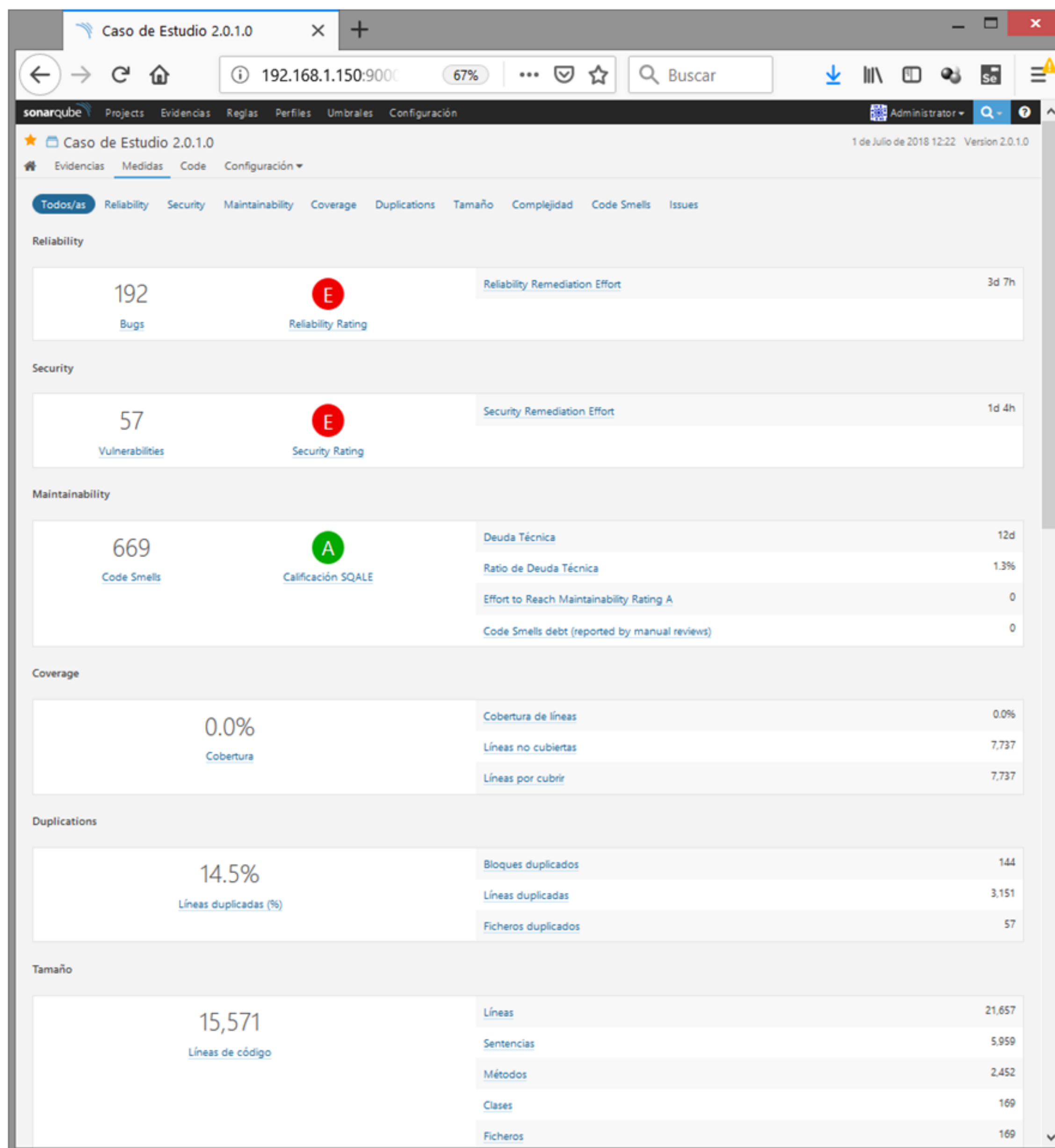
Figura N° F.1: Reporte de SonarQube, evidencias halladas en el caso de estudio antes de su refactoring.



Elaboración: Propia.

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

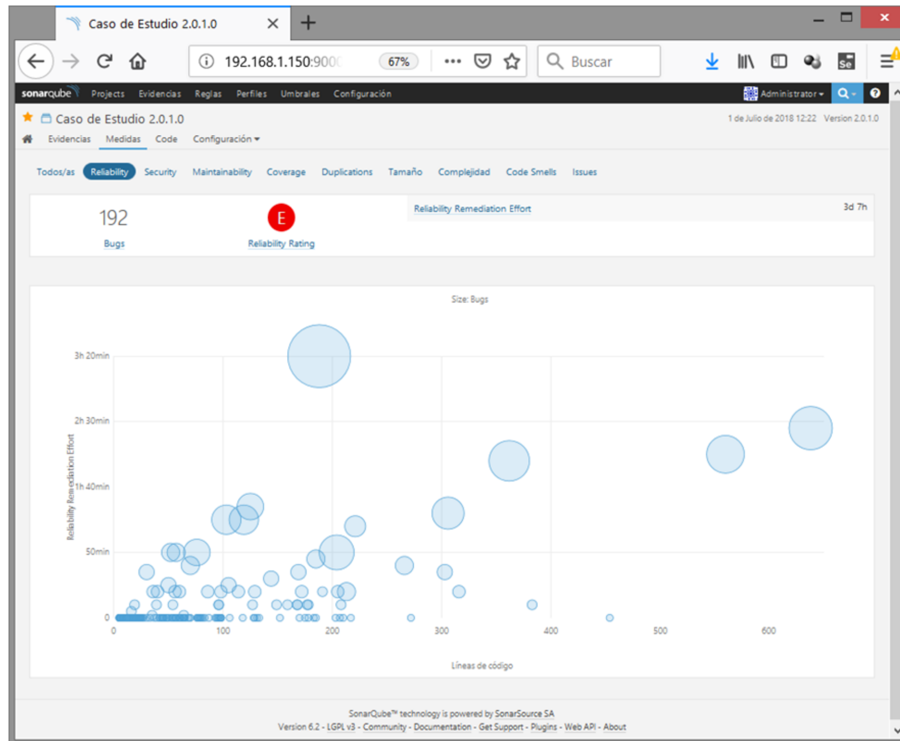
Figura N° F.2: Reporte de SonarQube, medidas halladas en el caso de estudio antes de su refactoring.



Elaboración: Propia.

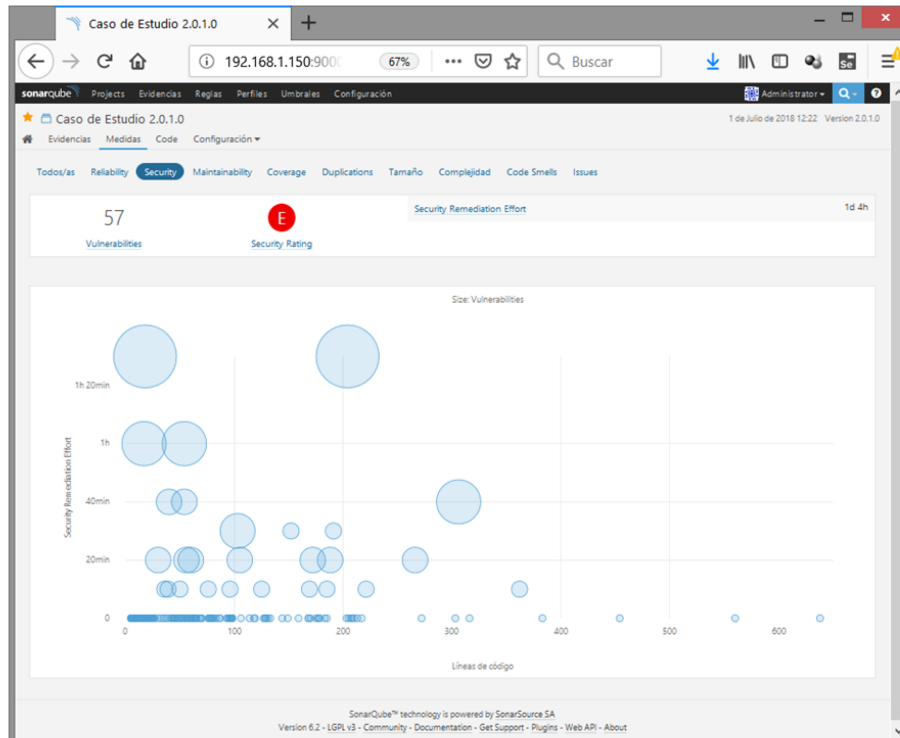
Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

Figura N° F.3: Reporte de SonarQube, medida de la fiabilidad hallada en el caso de estudio antes de su refactoring.



Elaboración: Propia.

Figura N° F.4: Reporte de SonarQube, medida de la seguridad hallada en el caso de estudio antes de su refactoring.

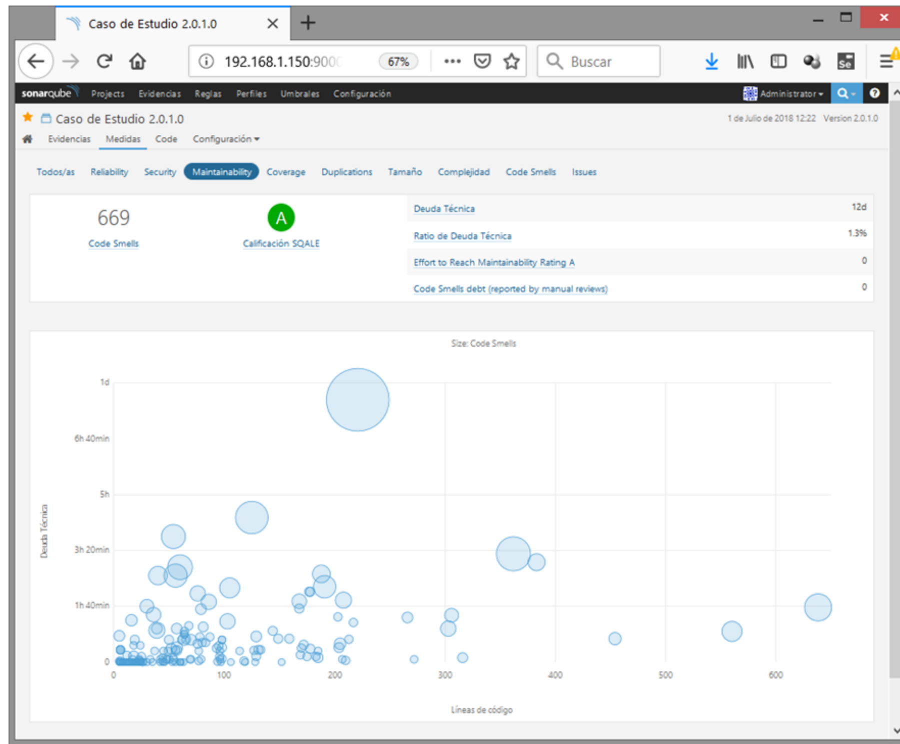


Elaboración: Propia.

Repositorio Institucional UNA-PUNO

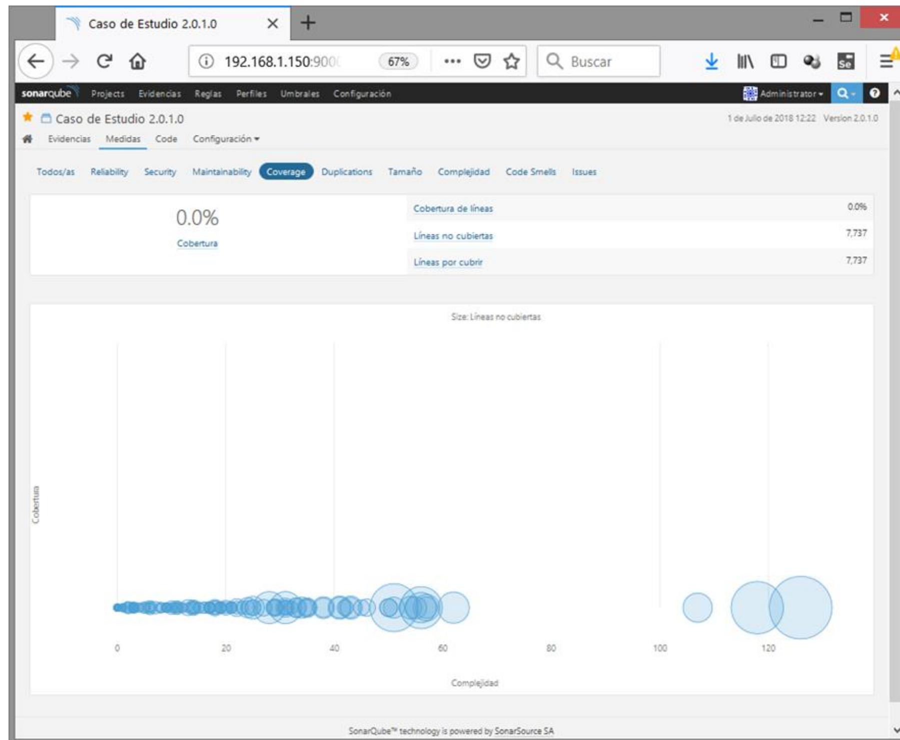
No olvide citar esta tesis

Figura N° F.5: Reporte de SonarQube, medida de la mantenibilidad hallada en el caso de estudio antes de su refactoring.



Elaboración: Propia.

Figura N° F.6: Reporte de SonarQube, medida de la cobertura hallada en el caso de estudio antes de su refactoring.

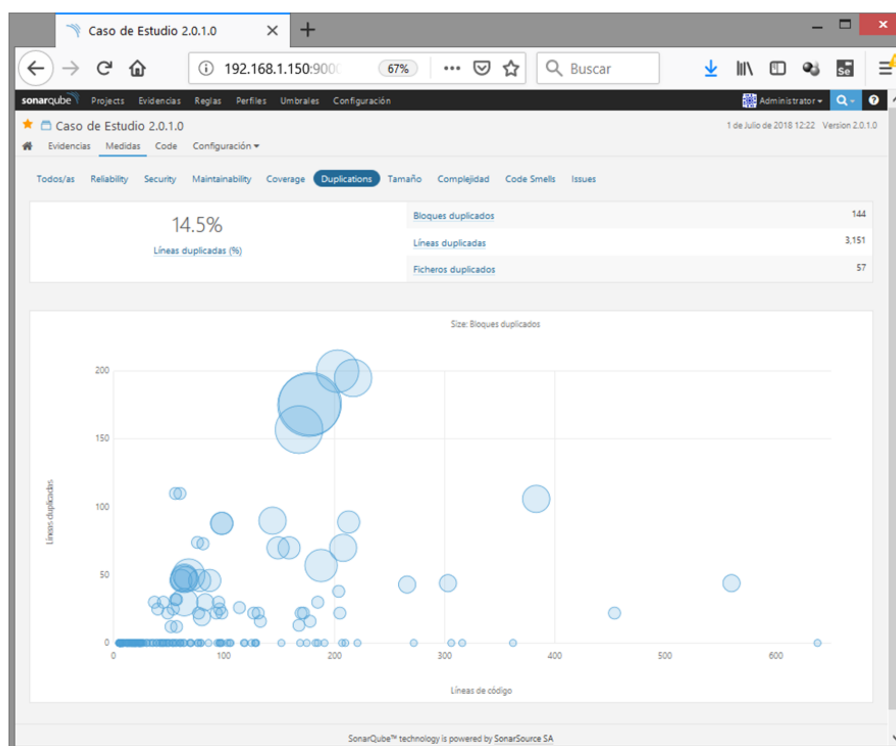


Elaboración: Propia.

Repositorio Institucional UNA-PUNO

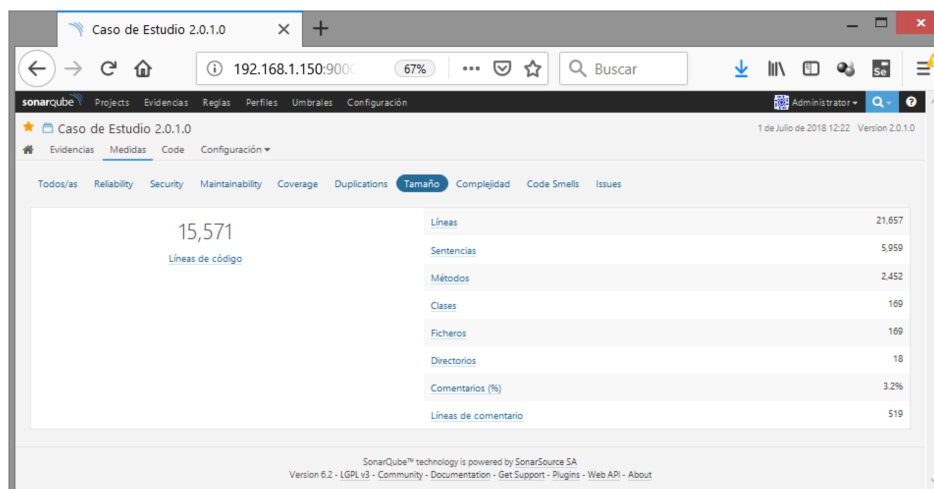
No olvide citar esta tesis

Figura N° F.7: Reporte de SonarQube, medida de la duplicación del caso de estudio antes de su refactoring.



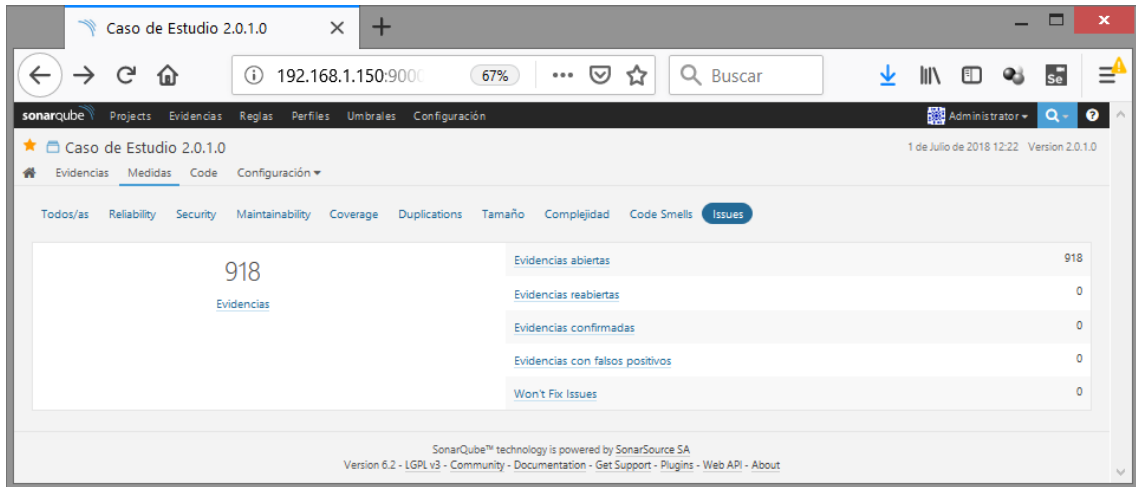
Elaboración: Propia.

Figura N° F.8: Reporte de SonarQube, medida del tamaño del caso de estudio antes de su refactoring.



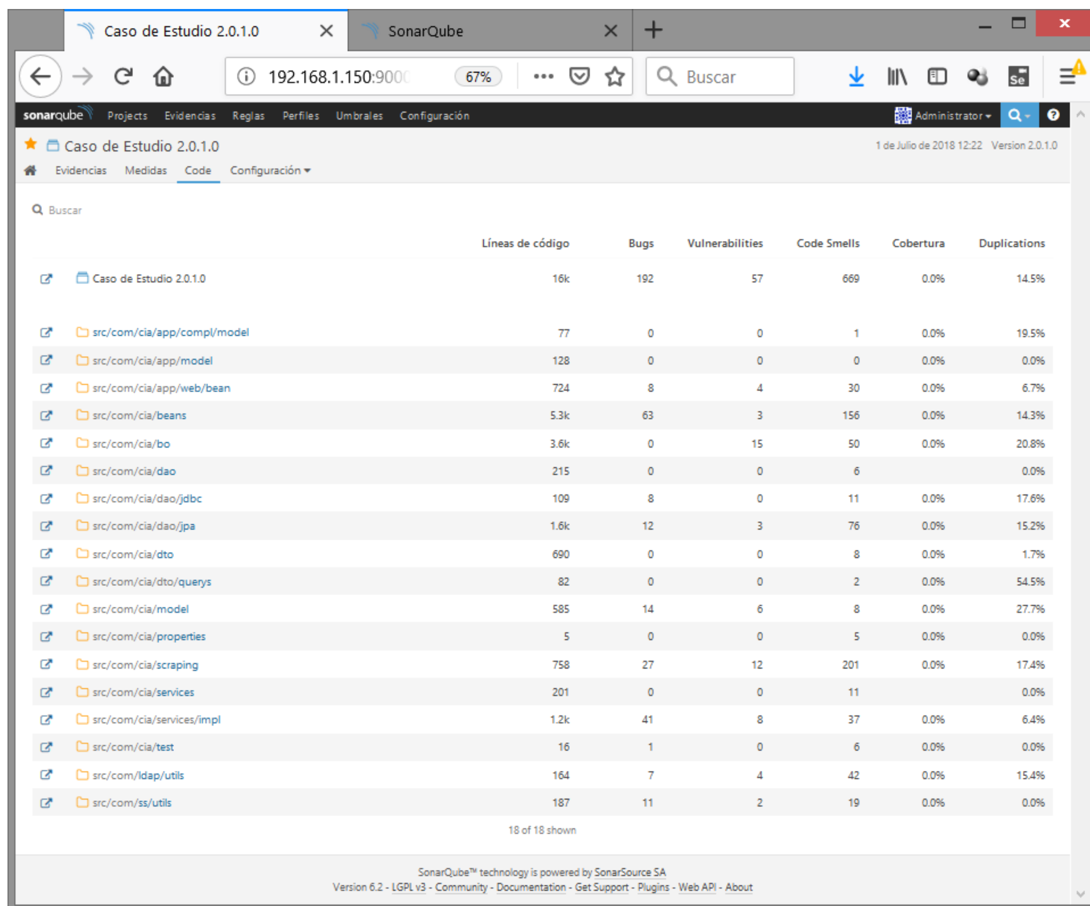
Elaboración: Propia.

Figura N° F.9: Reporte de SonarQube, medida de las evidencias halladas en el caso de estudio antes de su refactoring.



Elaboración: Propia.

Figura N° F.10: Reporte SonarQube, código del caso de estudio antes de su refactoring.



	Líneas de código	Bugs	Vulnerabilities	Code Smells	Cobertura	Duplications
Caso de Estudio 2.0.1.0	16k	192	57	669	0.0%	14.5%
src/com/cia/app/compl/model	77	0	0	1	0.0%	19.5%
src/com/cia/app/model	128	0	0	0	0.0%	0.0%
src/com/cia/app/web/bean	724	8	4	30	0.0%	6.7%
src/com/cia/beans	5.3k	63	3	156	0.0%	14.3%
src/com/cia/bo	3.6k	0	15	50	0.0%	20.8%
src/com/cia/dao	215	0	0	6		0.0%
src/com/cia/dao/jdbc	109	8	0	11	0.0%	17.6%
src/com/cia/dao/jpa	1.6k	12	3	76	0.0%	15.2%
src/com/cia/dto	690	0	0	8	0.0%	1.7%
src/com/cia/dto/querys	82	0	0	2	0.0%	54.5%
src/com/cia/model	585	14	6	8	0.0%	27.7%
src/com/cia/properties	5	0	0	5	0.0%	0.0%
src/com/cia/scraping	758	27	12	201	0.0%	17.4%
src/com/cia/services	201	0	0	11		0.0%
src/com/cia/services/impl	1.2k	41	8	37	0.0%	6.4%
src/com/cia/test	16	1	0	6	0.0%	0.0%
src/com/ldap/utils	164	7	4	42	0.0%	15.4%
src/com/ss/utils	187	11	2	19	0.0%	0.0%

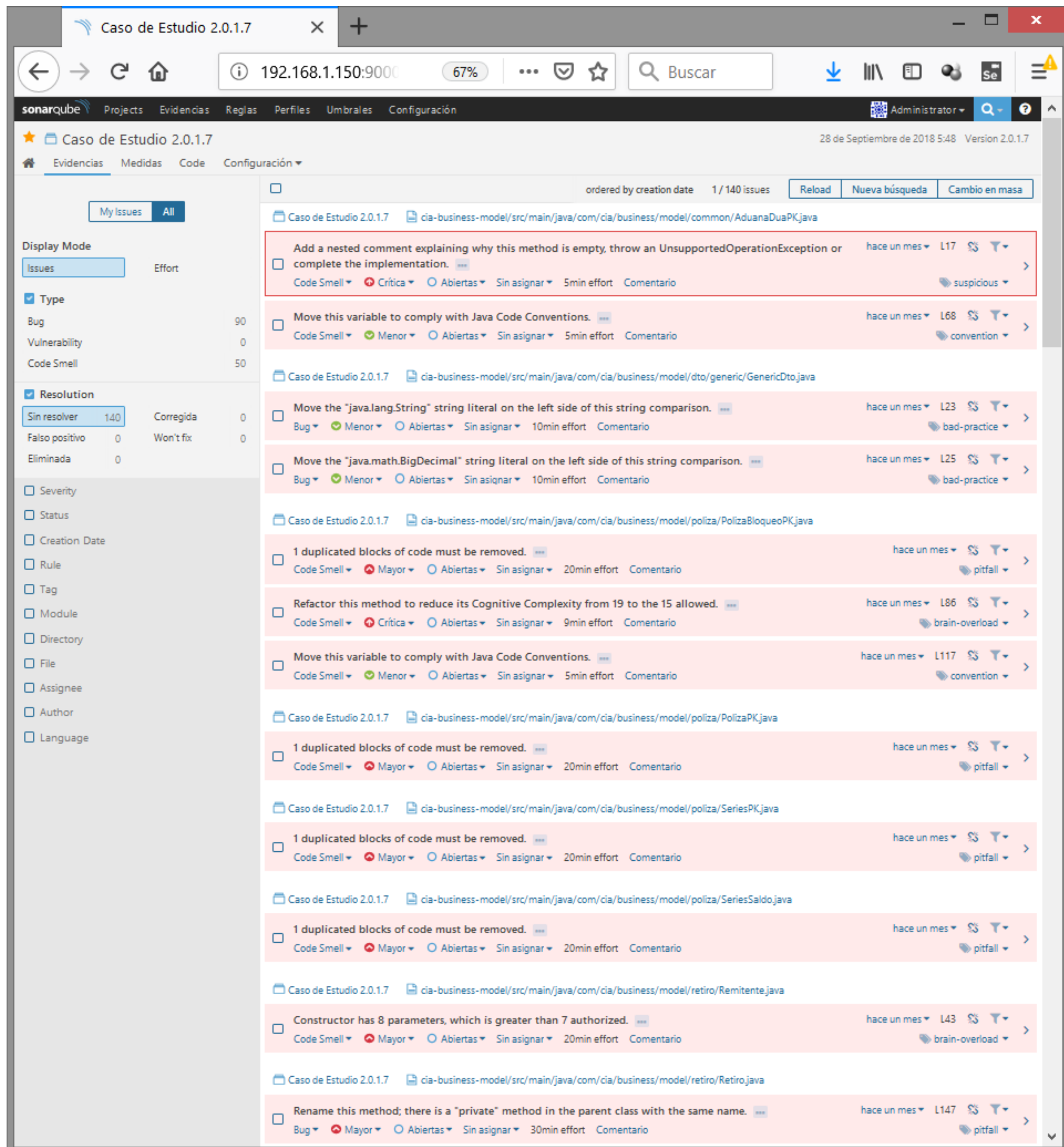
18 of 18 shown

Elaboración: Propia.

ANEXO G. Reportes de SonarQube del análisis efectuado al caso de estudio después del proceso de refactoring

A continuación, se presentan algunos reportes del software SonarQube del análisis realizado al caso de estudio después del proceso de refactoring.

Figura N° G.1: Reporte de SonarQube, evidencias halladas en el caso de estudio después de su refactoring.

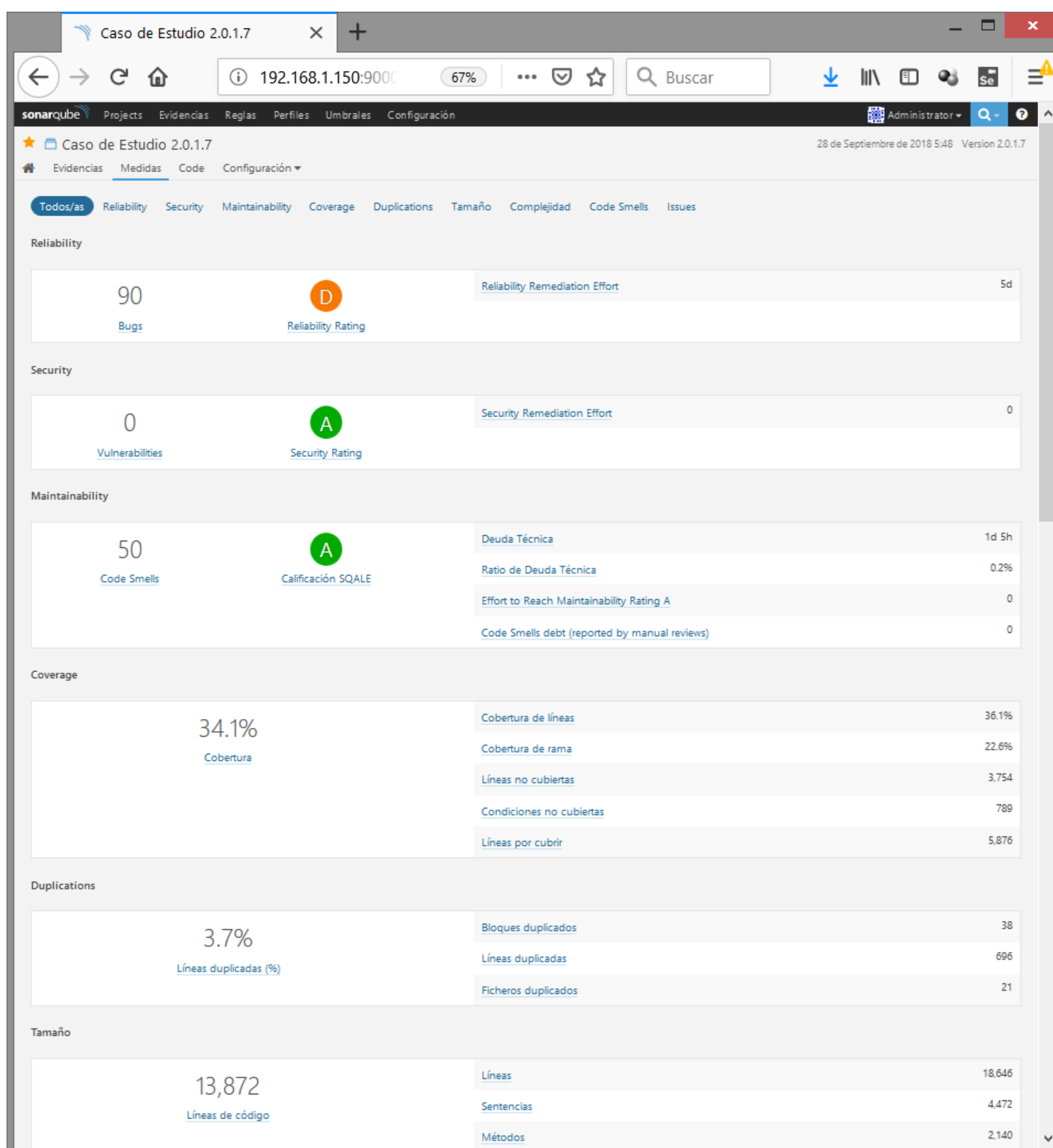


Elaboración: Propia.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

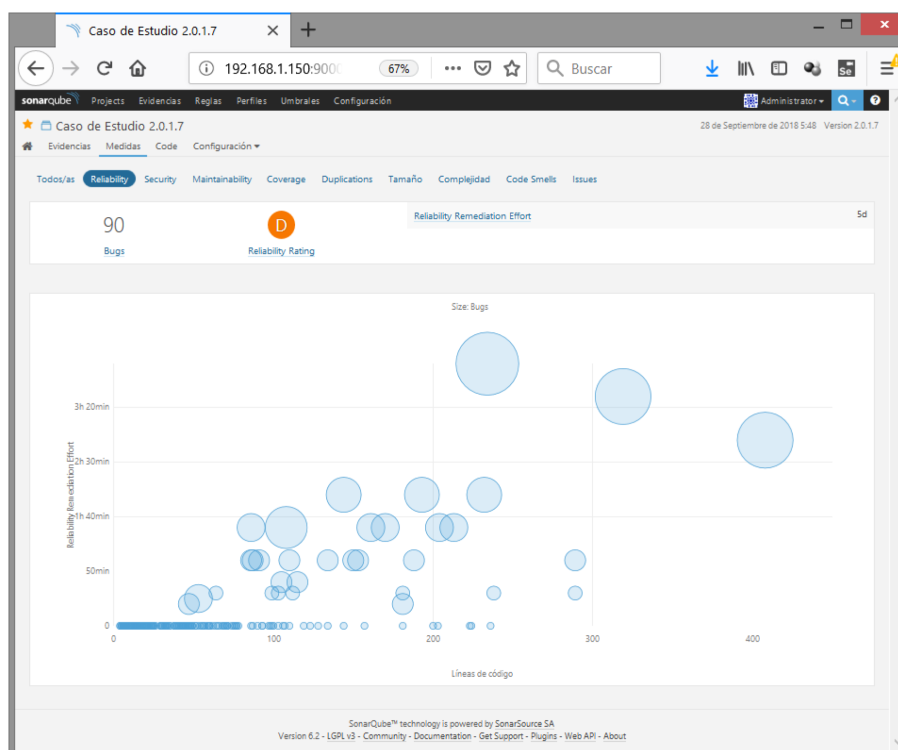
Figura N° G.2: Reporte de SonarQube, medidas halladas en el caso de estudio después de su refactoring.



Elaboración: Propia.

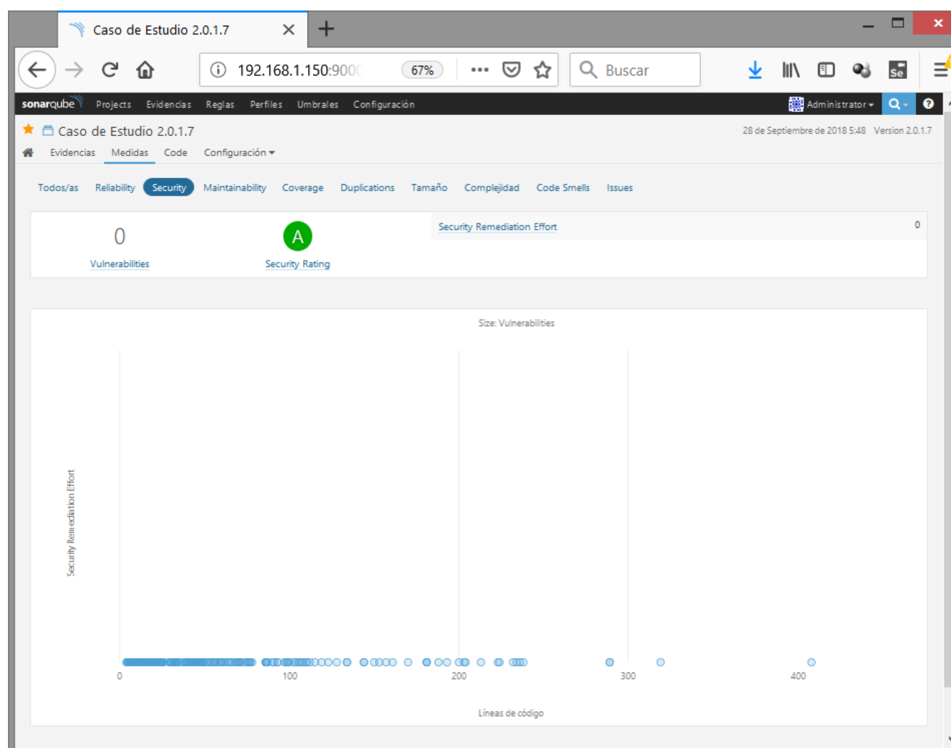
Repositorio Institucional UNA-PUNO
No olvide citar esta tesis

Figura N° G.3: Reporte de SonarQube, medida de la fiabilidad hallada en el caso de estudio después de su refactoring.



Elaboración: Propia.

Figura N° G.4: Reporte de SonarQube, medida de la seguridad hallada en el caso de estudio después de su refactoring.

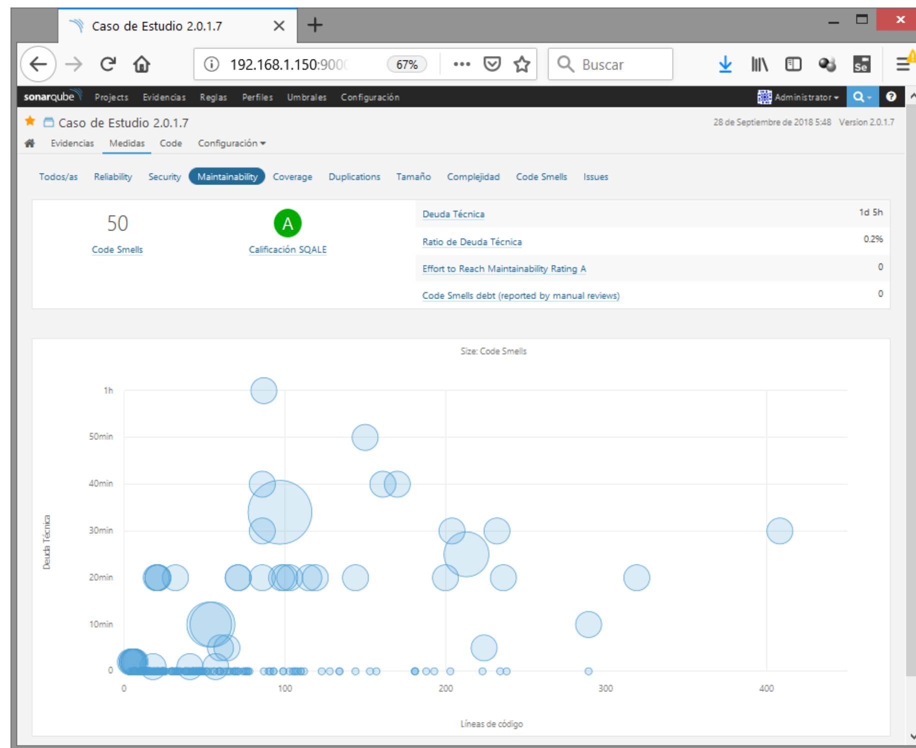


Elaboración: Propia.

Repositorio Institucional UNA-PUNO

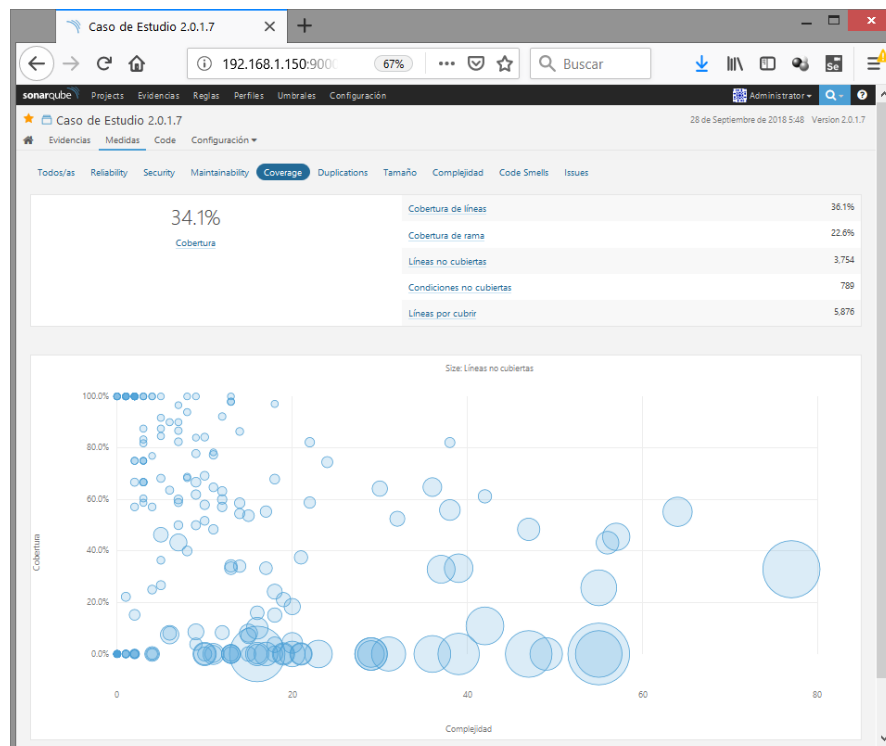
No olvide citar esta tesis

Figura N° G.5: Reporte de SonarQube, medida de la mantenibilidad hallada en el caso de estudio después de su refactoring.



Elaboración: Propia.

Figura N° G.6: Reporte de SonarQube, medida de la cobertura hallada en el caso de estudio después de su refactoring.

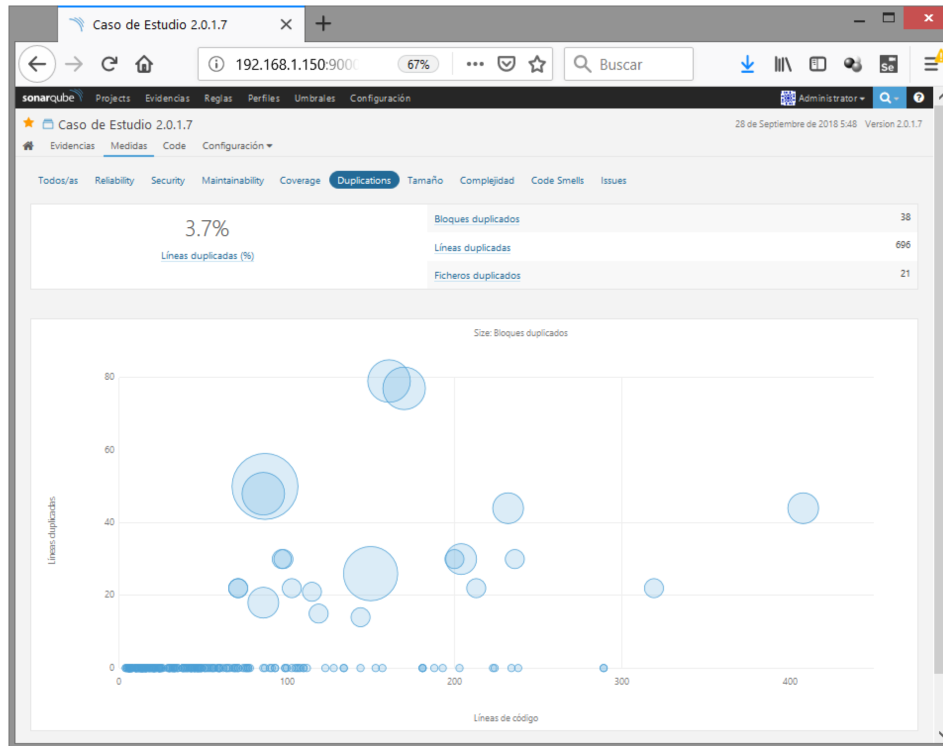


Elaboración: Propia.

Repositorio Institucional UNA-PUNO

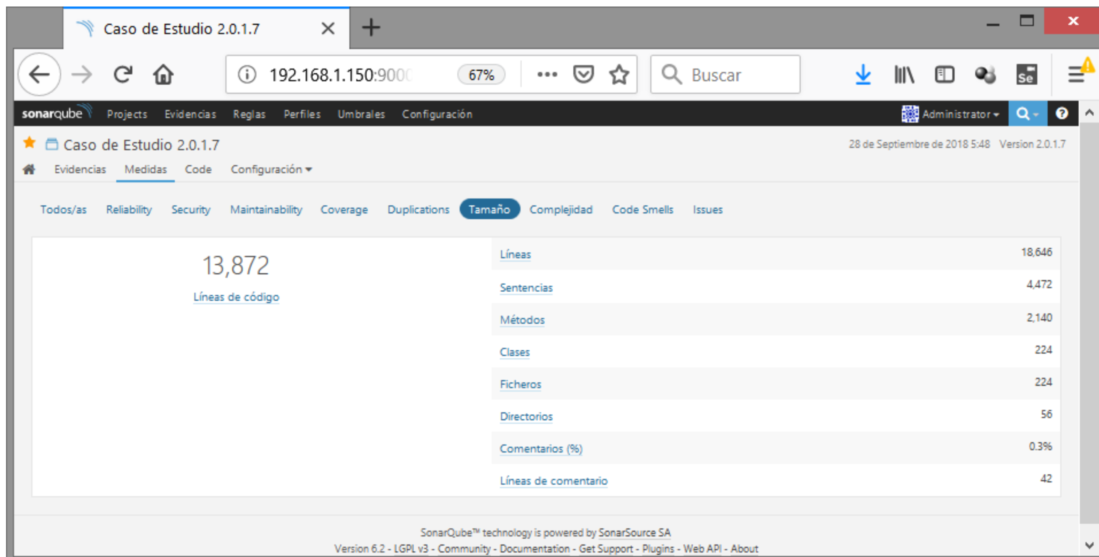
No olvide citar esta tesis

Figura N° G.7: Reporte de SonarQube, medida de la duplicación del caso de estudio después de su refactoring.



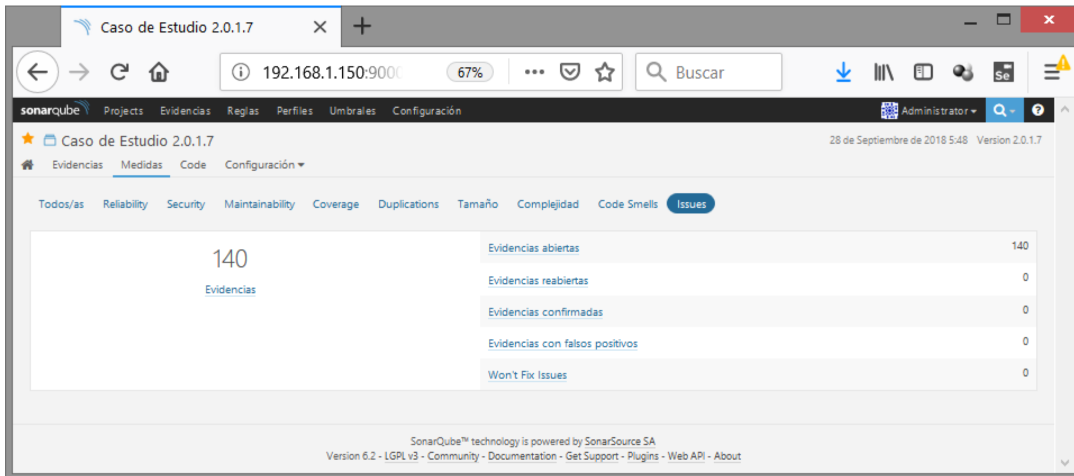
Elaboración: Propia.

Figura N° G.8: Reporte de SonarQube, medida del tamaño del caso de estudio después de su refactoring.



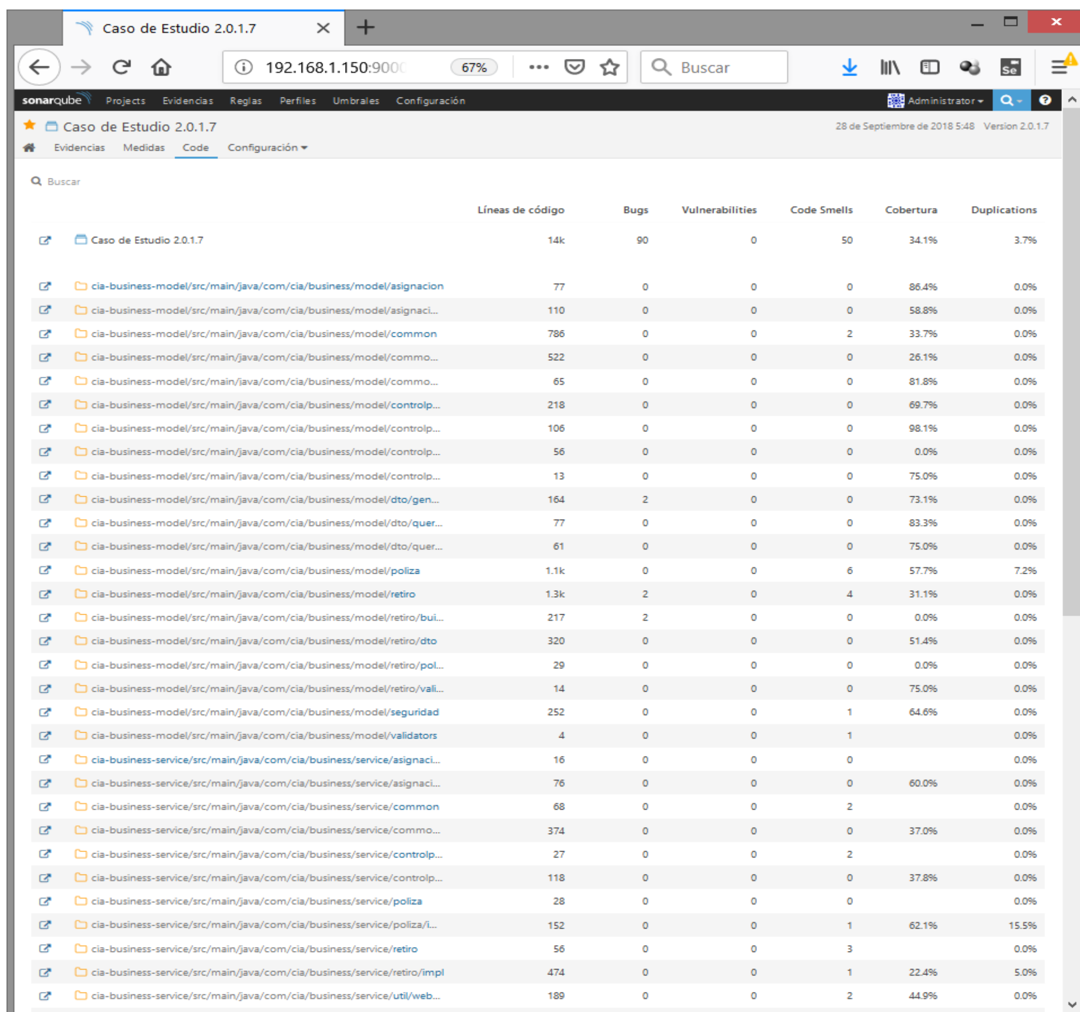
Elaboración: Propia.

Figura N° G.9: Reporte de SonarQube, medida de las evidencias halladas en el caso de estudio después de su refactoring.



Elaboración: Propia.

Figura N° G.10: Reporte SonarQube, código del caso de estudio después de su refactoring.



	Líneas de código	Bugs	Vulnerabilities	Code Smells	Cobertura	Duplications
Caso de Estudio 2.0.1.7	14k	90	0	50	34.1%	3.7%
cia-business-model/src/main/java/com/cia/business/model/asignacion	77	0	0	0	86.4%	0.0%
cia-business-model/src/main/java/com/cia/business/model/asignaci...	110	0	0	0	58.8%	0.0%
cia-business-model/src/main/java/com/cia/business/model/common	786	0	0	2	33.7%	0.0%
cia-business-model/src/main/java/com/cia/business/model/commo...	522	0	0	0	26.1%	0.0%
cia-business-model/src/main/java/com/cia/business/model/commo...	65	0	0	0	81.8%	0.0%
cia-business-model/src/main/java/com/cia/business/model/controlp...	218	0	0	0	69.7%	0.0%
cia-business-model/src/main/java/com/cia/business/model/controlp...	106	0	0	0	98.1%	0.0%
cia-business-model/src/main/java/com/cia/business/model/controlp...	56	0	0	0	0.0%	0.0%
cia-business-model/src/main/java/com/cia/business/model/controlp...	13	0	0	0	75.0%	0.0%
cia-business-model/src/main/java/com/cia/business/model/dto/gen...	164	2	0	0	73.1%	0.0%
cia-business-model/src/main/java/com/cia/business/model/dto/quer...	77	0	0	0	83.3%	0.0%
cia-business-model/src/main/java/com/cia/business/model/dto/quer...	61	0	0	0	75.0%	0.0%
cia-business-model/src/main/java/com/cia/business/model/poliza	1.1k	0	0	6	57.7%	7.2%
cia-business-model/src/main/java/com/cia/business/model/retiro	1.3k	2	0	4	31.1%	0.0%
cia-business-model/src/main/java/com/cia/business/model/retiro/bui...	217	2	0	0	0.0%	0.0%
cia-business-model/src/main/java/com/cia/business/model/retiro/dto	320	0	0	0	51.4%	0.0%
cia-business-model/src/main/java/com/cia/business/model/retiro/pol...	29	0	0	0	0.0%	0.0%
cia-business-model/src/main/java/com/cia/business/model/retiro/vali...	14	0	0	0	75.0%	0.0%
cia-business-model/src/main/java/com/cia/business/model/seguridad	252	0	0	1	64.6%	0.0%
cia-business-model/src/main/java/com/cia/business/model/validators	4	0	0	1		0.0%
cia-business-service/src/main/java/com/cia/business/service/asignaci...	16	0	0	0		0.0%
cia-business-service/src/main/java/com/cia/business/service/asignaci...	76	0	0	0	60.0%	0.0%
cia-business-service/src/main/java/com/cia/business/service/common	68	0	0	2		0.0%
cia-business-service/src/main/java/com/cia/business/service/commo...	374	0	0	0	37.0%	0.0%
cia-business-service/src/main/java/com/cia/business/service/controlp...	27	0	0	2		0.0%
cia-business-service/src/main/java/com/cia/business/service/controlp...	118	0	0	0	37.8%	0.0%
cia-business-service/src/main/java/com/cia/business/service/poliza	28	0	0	0		0.0%
cia-business-service/src/main/java/com/cia/business/service/poliza/i...	152	0	0	1	62.1%	15.5%
cia-business-service/src/main/java/com/cia/business/service/retiro	56	0	0	3		0.0%
cia-business-service/src/main/java/com/cia/business/service/retiro/impl	474	0	0	1	22.4%	5.0%
cia-business-service/src/main/java/com/cia/business/service/util/web...	189	0	0	2	44.9%	0.0%

Elaboración: Propia.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

ANEXO H. Evolución de la calidad interna del caso de estudio durante el desarrollo de su proceso de refactoring

Tabla N° H.1: Evolución de la calidad interna del caso de estudio durante el desarrollo su proceso de refactoring.

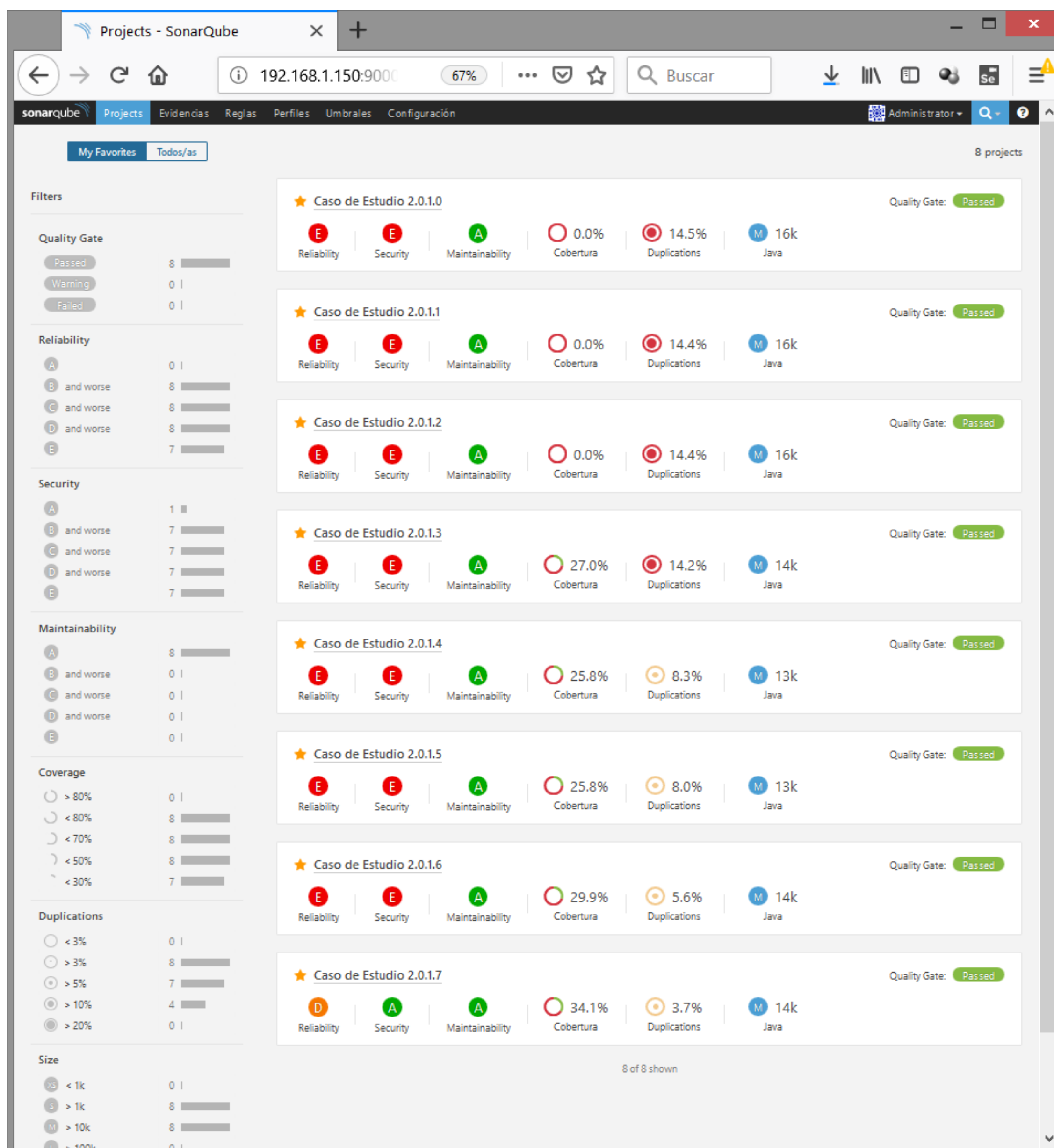
Iteración	Descripción	Resumen del análisis con SonarQube
-	Proyecto Incial	Caso de Estudio 2.0.1.0 — Quality Gate: Passed — E Reliability, E Security, A Maintainability, 0.0% Cobertura, 14.5% Duplications, M 16k Java
1	Restructuración del proyecto	Caso de Estudio 2.0.1.1 — Quality Gate: Passed — E Reliability, E Security, A Maintainability, 0.0% Cobertura, 14.4% Duplications, M 16k Java
2	Modularización del proyecto	Caso de Estudio 2.0.1.2 — Quality Gate: Passed — E Reliability, E Security, A Maintainability, 0.0% Cobertura, 14.4% Duplications, M 16k Java
3	Desarrollo de las pruebas unitarias y de integración en todo el proyecto	Caso de Estudio 2.0.1.3 — Quality Gate: Passed — E Reliability, E Security, A Maintainability, 27.0% Cobertura, 14.2% Duplications, M 14k Java
4	Refactoring de la capa de lógica de negocios -Módulo 1	Caso de Estudio 2.0.1.4 — Quality Gate: Passed — E Reliability, E Security, A Maintainability, 25.8% Cobertura, 8.3% Duplications, M 13k Java
5	Refactoring de la capa de lógica de negocios -Módulo 2	Caso de Estudio 2.0.1.5 — Quality Gate: Passed — E Reliability, E Security, A Maintainability, 25.8% Cobertura, 8.0% Duplications, M 13k Java
6	Refactoring de la capa de presentación	Caso de Estudio 2.0.1.6 — Quality Gate: Passed — E Reliability, E Security, A Maintainability, 29.9% Cobertura, 5.6% Duplications, M 14k Java
7	Refectoring de la capa de presentación. Proyecto Refactorizado	Caso de Estudio 2.0.1.7 — Quality Gate: Passed — D Reliability, A Security, A Maintainability, 34.1% Cobertura, 3.7% Duplications, M 14k Java

Elaboración: Propia.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

Figura N° H.1: Proyectos analizados en SonarQube, correspondientes a cada iteración del refactoring del caso de estudio.



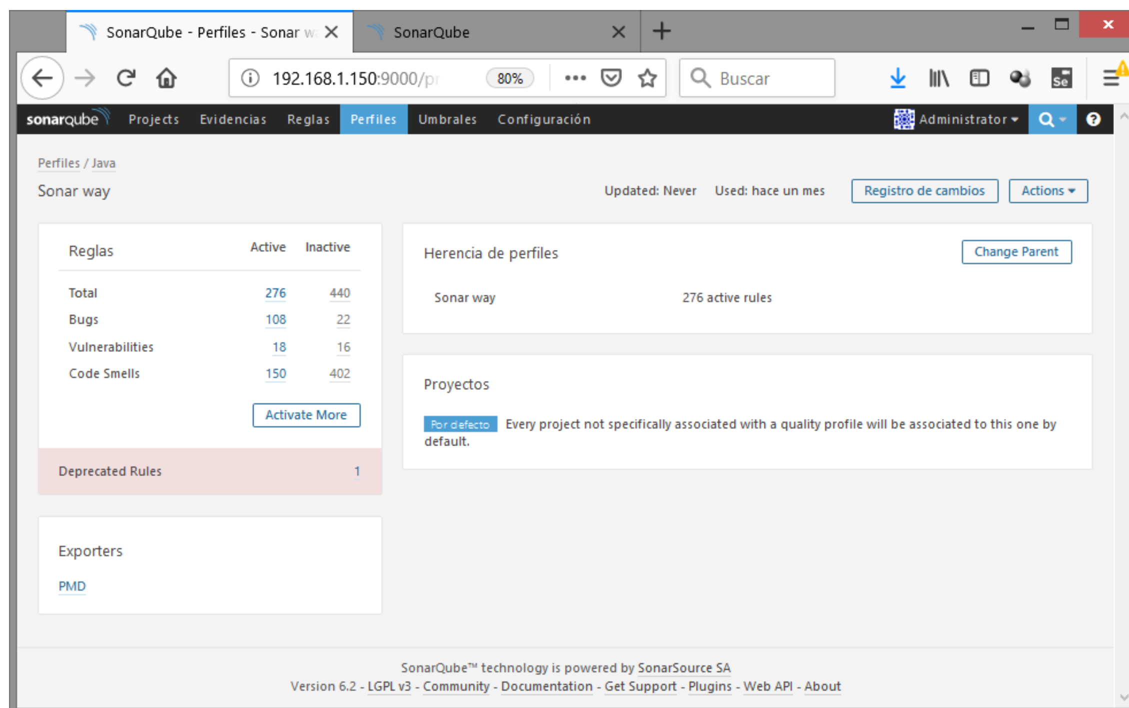
Elaboración: Propia.

Repositorio Institucional UNA-PUNO

No olvide citar esta tesis

ANEXO I. Perfil / Java de SonarQube usado para el análisis del caso de estudio

Figura N° I.1: Perfil / Java de SonarQube usado para el análisis del caso de estudio.



Elaboración: Propia.

Repositorio Institucional UNA-PUNO
No olvide citar esta tesis