Universidad de la República
Facultad de Ingeniería

Análisis de defectos de diseño

Grupo de Ingeniería de Software - Facultad de Ingeniería

Rodrigo Lago

Mario Nicolás Sánchez

Belén Silvotti

Informe de Proyecto de Grado presentado al Tribunal Evaluador como requisito de graduación de la carrera Ingeniería en Computación de la Universidad de la República

Tutores

Cecilia Apa ..... Universidad de la República
Vanessa Casella ..... Universidad de la República
Diego Vallespir ..... Universidad de la República

Montevideo, Uruguay
Marzo 2022

Página ha sido intencionalmente dejada en blanco.

Resumen

La calidad interna del software se puede ver afectada por las malas prácticas de diseño que pueden generar defectos de diseño. Estos defectos de diseño pueden aumentar el tiempo de desarrollo y mantenimiento del software, generando retrabajo, atrasos en las entregas, entre otros. El conocimiento e identificación de diferentes tipos de defectos de diseño contribuye a mejorar su calidad interna. Para esto, resulta importante apoyarse en herramientas automatizadas con el objetivo de optimizar el esfuerzo en las detecciones.

Para mejorar tanto las prácticas de diseño de software como la forma en la que se enseña a diseñar en la carrera Ingeniería en Computación de la Facultad de Ingeniería, en este trabajo se presenta la realización de un estudio exploratorio con el objetivo de conocer y analizar los defectos de diseño documentados como *Code Smells* y *AntiPatterns* en los que incurren los estudiantes de pregrado al desarrollar software. Para ello, se determina un conjunto de *Code Smells* y *AntiPatterns* vinculados a defectos de diseño, se define una taxonomía de clasificación de los mismos y se selecciona y configura un conjunto de herramientas para su detección, que luego se utiliza sobre el código de un producto de software construido por estudiantes.

Los resultados de nuestro trabajo muestran que la mayor cantidad de ocurrencias de *Code Smells* se observó en las categorías relacionadas a convenciones y tecnología Java, y que los *AntiPatterns* detectados fueron escasos y los tiempos estimados de refactorización imprecisos. Las principales clases afectadas fueron aquellas con alta complejidad y que concentraban la mayor parte de la lógica de negocio. Algunas debilidades y deficiencias fueron identificadas en la definición y utilización de estándares, así como en las revisiones de código realizadas por los estudiantes. En base a estos resultados, brindamos algunas sugerencias de mejora tanto a nivel del proceso de desarrollo de software como en la utilización de las herramientas automatizadas, para optimizar el tiempo dedicado a la detección y corrección de estos defectos de diseño.

Diversos productos generados para el desarrollo de este proyecto pueden ser utilizados a futuro por la industria para mejorar sus procesos de desarrollo de software, así como por la comunidad científica para futuros estudios relacionados a defectos de diseño.

Palabras clave: Defectos de diseño, *Code Smells*, *AntiPatterns*, calidad.

Página ha sido intencionalmente dejada en blanco.

Tabla de contenidos

Capítulo 1

Introducción

La actividad profesional del ingeniero del software abarca un amplio rango de tareas involucradas en el ciclo de vida de un sistema software: la obtención de requisitos, especificación del software, diseño e implementación del software, validación del software y evolución del software [32]. Estas actividades son necesarias para desarrollar un producto de software y realizarlas de forma adecuada es fundamental para obtener un producto de calidad.

En ingeniería de software, el término calidad refiere a la "concordancia con los requisitos funcionales y de rendimiento explícitamente establecidos con los estándares de desarrollo plenamente documentados y con las características implícitas que se espera de todo software desarrollado" [33]. Según el estándar ISO 9126 [19], cualquier componente de la calidad del software puede ser descrito en términos de una o más de seis características básicas: funcionalidad, confiabilidad, usabilidad, eficiencia, mantenibilidad y portabilidad. Cada una de estas características presenta a su vez un conjunto de subcaracterísticas que permiten profundizar en la evaluación de la calidad de productos de software.

Los productos de software son complejos en sí mismos. La complejidad inherente al software deriva, por ejemplo de: la complejidad del dominio del problema, la dificultad de gestionar el proceso de desarrollo de software, la existencia de múltiples factores que impiden la verificación en profundidad de todas las posibles ejecuciones del sistema, entre otros [7]. Adicionalmente, el hecho de que un producto sea intangible y abstracto hace que los requisitos sean difíciles de consolidar tempranamente [33]. Como consecuencia, los cambios en los requisitos son inevitables durante el proceso de desarrollo y posiblemente también después de entregado el producto.

El proceso de desarrollo de software tiene como propósito la producción eficaz y eficiente de un producto de software que reúna los requisitos del cliente [33]. Existen múltiples procesos de desarrollo de software donde ninguno es catalogado como el "ideal". A medida que los sistemas se vuelven más grandes y complejos, sumado a la complejidad del proceso de desarrollo, una preocupación frecuente es la existencia de problemas de diseño que afecten la calidad interna del software [22].

Los defectos de diseño hacen referencia a problemas en la estructura de un sistema, que no necesariamente producen errores de compilación o de ejecución,

pero que afectan negativamente a los factores de calidad interna del software [27]. Por lo general, estos defectos producen problemas durante el mantenimiento y evolución del sistema [14]. Estos problemas se pueden abordar, dejar postergados para una resolución posterior, o se pueden ignorar. Aquellos sistemas de software que contienen defectos de diseño en su código fuente pueden complejizar el desarrollo o incrementar el riesgo de defectos o fallos en el software a futuro.

Los defectos de diseño pueden surgir en diferentes niveles de granularidad, desde problemas de diseño de alto nivel (arquitectura), por ejemplo, *AntiPatterns*, hasta problemas de bajo nivel (diseño detallado), por ejemplo, *Code Smells* [9].

Los *Code Smells* y *AntiPatterns* documentan defectos comunes cometidos por los desarrolladores al construir sistemas de software, de forma similar a los patrones de diseño de software que documentan buenos diseños de software para abordar ciertos problemas comunes. A menudo, la presencia de defectos de diseño en el código indica que existen problemas de calidad interna de código, de diseño o de ambos [9]. Los *Code Smells* y *AntiPatterns*, además de documentar el problema, sugieren posibles soluciones. Estas soluciones sugieren normalmente la refactorización del software. Al refactorizar el software, se modifica la estructura interna del mismo, con la finalidad de que sea más fácil de entender y modificar, sin alterar su comportamiento externo [38]. Con la refactorización lo que se pretende es mejorar el código fuente, el diseño y/o la arquitectura del software, de manera que el costo y el esfuerzo de realizar modificaciones sobre el sistema sean menores respecto a hacerlo sobre el software original [18].

1.1. Motivación

La evolución de las tecnologías, como el uso del paradigma orientado a objetos, ayuda a que los sistemas de software sean más flexibles, extensibles y comprensibles, y por lo tanto más fáciles de mantener [20]. Sin embargo, a pesar del uso de las nuevas tecnologías que podrían llegar a facilitar el proceso de desarrollo de software, es muy común el uso de prácticas ineficientes durante dicho proceso [6]. Se listan a continuación ejemplos de "malas prácticas" de desarrollo llevadas a cabo en proyectos de software [2].

- Toma de decisiones de forma apresurada.
- Poca experiencia o ignorancia por parte de los líderes del proyecto.
- Apatía por resolver problemas conocidos.
- Rechazo a poner en práctica soluciones ampliamente conocidas por ser eficaces.
- No utilización ni integración de paquetes comerciales, software gratuito ni aplicaciones heredadas empaquetadas.
- Modelado excesivo de detalles resultando en una abstracción insuficiente.

Estas malas prácticas (y otras) pueden provocar debilidades en el diseño que a su vez pueden complejizar el desarrollo o incrementar el riesgo de errores o fallos en el software a futuro.

La identificación de *Code Smells* y *AntiPatterns* en el código es el primer paso hacia la práctica de refactorización [18]. El conocimiento de estos defectos de diseño puede ser utilizado, al igual que los patrones de diseño, para guiar el diseño de software [9].

Sin embargo, múltiples estudios indican que los estudiantes de pregrado presentan dificultades para diseñar adecuadamente un sistema de software [1,8,39]. Un estudio realizado en nuestra Facultad [25] señala que estudiantes de pregrado avanzados en la carrera Ingeniería en Computación dedican cuatro veces menos tiempo al diseño de software en comparación con el tiempo dedicado a la implementación. En este estudio se observa que en líneas generales los estudiantes no perciben las consecuencias negativas de no diseñar o no dedicar el tiempo suficiente a dicha disciplina.

Con el objetivo de mejorar tanto las prácticas de diseño de software como la forma en la que se enseña a diseñar en la carrera Ingeniería en Computación de Facultad de Ingeniería resulta de interés estudiar los *Code Smells* y *AntiPatterns* en los que incurren los estudiantes de pregrado de nuestra Facultad al desarrollar software.

1.2. Objetivos

El objetivo de este trabajo es conocer y analizar diferentes tipos de defectos de diseño, documentados en *Code Smells* y *AntiPatterns*. En particular, aquellos en los que incurren los estudiantes de grado avanzados de la Facultad de Ingeniería, UdelaR, al desarrollar software en el contexto de la asignatura Proyecto de Ingeniería de Software (PIS), que puedan ser detectados por herramientas automatizadas.

En este marco, para poder cumplir con el objetivo planteado es necesario definir las herramientas automatizadas a utilizar para la detección *Code Smells* y *AntiPatterns* a analizar. Adicionalmente, se necesita definir cuáles serán los *Code Smells* y *AntiPatterns* que interesa estudiar con las herramientas escogidas.

Del objetivo planteado se desprenden las siguientes preguntas de investigación (RQ):

- **RQ1: ¿En qué *Code Smells* incurren los estudiantes cuando realizan el curso PIS?**
  Esta pregunta intenta responder cuáles de los *Code Smells* son detectados automáticamente en el código fuente del producto entregado por estudiantes del PIS y cómo se categorizan.

- **RQ2: ¿Qué criticidad tienen los *Code Smells* en los que incurren los estudiantes?**
  En esta pregunta se pretende caracterizar los *Code Smells* detectados teniendo en cuenta su tiempo de refactorización y gravedad.

- **RQ3: ¿Cómo se distribuyen los *Code Smells* detectados en el código del proyecto?**
  Esta pregunta busca conocer la distribución de los *Code Smells* detectados en las clases Java de los proyectos a estudiar, las características de las clases donde se detectan *Code Smells*, y la densidad de ocurrencias de los

mismos. El término *densidad de Code Smells* asociado a una clase hace referencia al resultado del cálculo de la cantidad de ocurrencias del *Code Smell* por línea de código en la clase.

- **RQ4: ¿En qué *AntiPatterns* incurren los estudiantes?**
  Esta pregunta intenta responder cuáles *AntiPatterns* son detectados en el código fuente del producto entregado por estudiantes del PIS y cómo se distribuyen.

1.3. Trabajo realizado

Para conocer el estado del arte sobre defectos de diseño y herramientas de detección automatizadas sobre *Code Smells* y *AntiPatterns* realizamos una revisión de literatura existente, con el objetivo de profundizar en la terminología asociada a defectos de diseño, de poder categorizar los *Code Smells* y *Antipatterns* a analizar. La revisión se presenta en el capítulo 2.

Con el fin de poder determinar el conjunto de herramientas a utilizar para la detección de *Code Smells* y *AntiPatterns* a analizar en el proyecto PIS, definimos un proceso de selección de herramientas, detallado en el capítulo 4. Adicionalmente, en el capítulo también se incluye un proceso de selección del cual surge un conjunto de *Code Smells* y *Antipatterns* a estudiar. Dentro de este conjunto, analizamos aquellos *Code Smells* que cumplan con alguna de las siguientes características:

- Se relacionan con *AntiPatterns* de diseño, es decir, *Code Smells* que pueden dar indicios de un problema de diseño más profundo. Estos *Code Smells* dificultan la comprensión y mantenimiento del mismo, pudiendo advertir sobre problemas de desarrollo más amplios como elecciones de diseño incorrectas [34].

- Son de carácter crítico en cuanto a esfuerzo en su refactorización y criticidad establecida por las herramientas automatizadas a utilizar.

- Son los más incurridos en el código a analizar. Es decir, son aquellos *Code Smells* con mayor cantidad de ocurrencias.

A su vez, interesa profundizar en aquellos *AntiPatterns* que cumplen con alguna de las siguientes características:

- Surgen de detecciones de la herramienta *Code Smells and AntiPatterns detection*.

- Surgen a partir de revisiones manuales de código sobre ocurrencias de *Code Smells* que dan indicios de *AntiPatterns* a partir del mapeo manual de *Code Smells* y *AntiPatterns* realizado.

Luego de obtener el conjunto resultado de *Code Smells* y *AntiPatterns*, configuramos las herramientas para poder ejecutarlas sobre un proyecto realizado por estudiantes del curso Proyecto de Ingeniería de Software (PIS), de la carrera Ingeniería en Computación de la Facultad de Ingeniería, UdelaR.

Los resultados de nuestro trabajo muestran que el equipo de estudiantes incurrió en un conjunto de defectos de diseño documentados como *Code Smells* y *AntiPatterns*, donde existen defectos asociados a convenciones del lenguaje y aspectos específicos de la tecnología Java, defectos asociados a complejidad excesiva de métodos y clases, violación de principios de orientación a objetos, prácticas inadecuadas de codificación, entre otros.

1.4. Estructura del documento

El esquema general de este trabajo final se organiza de la siguiente manera:

En el capítulo 2, se define la terminología a utilizar en la investigación (defectos de diseño, *Code Smells*, *AntiPatterns*), se describen y categorizan los *Code Smells* básicos [17] según la taxonomía Mäntylä del año 2003 [28] y se determina el conjunto de *AntiPatterns* a utilizar posteriormente en la investigación.

En el capítulo 3, se detallan las características del estudio experimental exploratorio realizado para llevar adelante la investigación. Se define el objeto de estudio y se contextualiza el mismo.

En el capítulo 4, se describe en detalle el proceso de selección de herramientas de detección automatizadas de *Code Smells* y *AntiPatterns*. Asimismo, se detalla el proceso de selección de *Code Smells* a tener en cuenta para la investigación, y se describe la interrelación entre ambos procesos. Además, en la última sección del capítulo se detalla la configuración de las herramientas seleccionadas para la detección de los *Code Smells* y *AntiPatterns* seleccionados.

En el capítulo 5, se presenta el análisis realizado de *Code Smells* y *AntiPatterns* incurridos por los estudiantes del proyecto PIS a analizar, respondiendo a las preguntas de investigación.

Finalmente, el capítulo 6, resume las conclusiones, principales contribuciones, y posibles trabajos futuros.

Página ha sido intencionalmente dejada en blanco.

Capítulo 2

Antecedentes: una aproximación a los Defectos de Diseño

El presente trabajo de investigación toma como antecedente fundamental la Tesis de Maestría "Detectando y Evitando Defectos de Diseño de Software - Un catálogo de *AntiPatterns* y un análisis de los *Code Smells* en los que incurren estudiantes de grado" [9] presentada por Vanessa Casella en 2021, en el marco de la culminación de su programa de Maestría en Informática de el Programa de desarrollo de las Ciencias Básicas (PEDECIBA Informática), de la Facultad de Ingeniería de la Universidad de la República (UdelaR). Según la autora, los factores de calidad del software se ven afectados por las malas prácticas llevadas a cabo en la actividad de diseño del software. Los defectos de diseño se definen como problemas en la estructura de un sistema. Así como la creación de un diseño de software simple y eficiente puede ser una tarea compleja, lo mismo sucede al intentar identificar y corregir los defectos de diseño. En este marco, Casella propone analizar desde diversas perspectivas diferentes tipos de defectos de diseño, en particular los denominados *Code Smells* y *AntiPatterns* en código desarrollado por estudiantes avanzados de pregrado de la Facultad de Ingeniería.

En esta línea, se toma como base la propuesta de diferentes autores que profundizan en el análisis de la afectación de los sistemas en base a dichos defectos de diseño. El fin principal que persigue este enfoque se relaciona con investigar, identificar y caracterizar los defectos de diseño en los que incurren estudiantes de pregrado al desarrollar software en el contexto de asignaturas de la carrera. Más específicamente, en el presente capítulo se destacan trabajos relacionados al estudio de los defectos de diseño, análisis y detección de *Code Smells* y *AntiPatterns*.

2.1. Terminología

Según el IEEE[1], el término "calidad" se define como "el grado con el que un sistema, componente o proceso cumple los requerimientos especificados y las necesidades o expectativas del cliente o usuario" [30]. Desde esta perspectiva,

[1] Instituto de Ingenieros Eléctricos y Electrónicos.

la calidad del software no refiere únicamente a obtener un producto final con la menor cantidad posible de errores sino, también, a que el código fuente del producto sea mantenible, extensible y modificable, de manera de evitar problemas con el paso del tiempo [10]. La calidad se obtiene mejorando día a día el proceso de producción, mantenimiento y gestión del software [35].

La calidad del software no solo busca cumplir con las expectativas del cliente sino también mejorar los procesos internos de elaboración de un producto de software. La misma puede ser analizada desde dos perspectivas: la interna y la externa. Según la norma ISO/IEC 25010:2011 [37], la calidad interna refiere a "las características y detalles de calidad del producto software que pueden ser mejoradas durante la implementación, revisión y prueba del código fuente" [37]. La calidad interna del producto permanece inalterada siempre y cuando el software no sea rediseñado. Desde la perspectiva externa, "la calidad se estudia cuando el software es ejecutado, la cual es típicamente medida y evaluada mientras se prueba en un ambiente simulado con datos simulados y usando métricas externas" [37].

Asimismo, a nivel interno la expresión "código de mala calidad" es generalmente asociada a software a distintos tipos de errores en el código fuente. La mala calidad de código implica problemas en la comprensión [21] y estructuración del mismo. Por otra parte, el código de mala calidad suele ser propenso a fallos al enfrentarse a eventuales cambios relacionados a la evolución del software [15]. Como consecuencia, diversos estudios indican que hay un impacto negativo directo en la mantenibilidad del software y los costos asociados [14].

Los errores en el código fuente pueden ser indicios de problemas más profundos vinculados a la operación, la evolución y el mantenimiento del software. Dichos errores de código han sido denominados como "*Code Smells*" [17], aludiendo a una evaluación de la estructura y del diseño del software, y a su potencial impacto en la presencia de errores y/o problemas del sistema. Posteriormente, el desarrollo y la profundización de la investigación sobre calidad de código fue enfocándose hacia el estudio de los errores generados por los *Code Smells* en el código fuente del sistema. De estos análisis surgen conceptos relacionados a los problemas que se presentan en la estructuración de un sistema, en su evolución, mantenimiento, entre otros. A este tipo de problemas se los denomina "defectos de diseño". En el libro "*Design Patterns*" [13] publicado en 1994. los autores Gamma, Helm, Johnson y Vlissides establecen la noción de "patrón de diseño", mediante la presentación de guías basadas en la identificación del comportamiento disfuncional del patrón, y en la refactorización de la solución. Fue en 1995 que el autor Webster [43] introduce el concepto de *AntiPatterns*. Posteriormente, en 1998 Fowler [16] introduce a los *Code Smells*.

Los **defectos de diseño** son una desviación entre el resultado esperado y el resultado obtenido en el diseño del software [34]. Esa desviación tiene un impacto negativo en la estructura general del sistema. Asimismo, los defectos de diseño no producen errores de compilación o de ejecución, pero afectan negativamente a los atributos de calidad interna del software [34], impidiendo su evolución. De esta manera, el código que contiene defectos de diseño tiene una mayor tendencia a fallar en relación con el resto del código del sistema [15].

Más específicamente, los defectos de diseño surgen como consecuencia de malas prácticas durante la fase de diseño del software, y/o de la ausencia de

dicha fase en el proceso de desarrollo del software. La repetición de código y no reutilización del conocimiento, o la urgencia en un *release* motivado a un ajustado cronograma de planificación del proyecto de software, son ejemplos frecuentes de malas prácticas de diseño [24].

Por otra parte, los ***Code Smells*** (olores de código en español) son un tipo de defecto de diseño de bajo nivel. Refieren a síntomas que indican que el código fuente presenta problemas de diseño [17]. Este tipo de defecto documenta errores comunes cometidos por los desarrolladores en la construcción del sistema. Surgen, principalmente, como consecuencia de malas prácticas empleadas durante el proceso de desarrollo de software. Entre ellas, podemos nombrar al código duplicado, a los métodos cuando son muy largos o cuando engloban funcionalidades en exceso, a las clases muy grandes, entre otros [40].

Por su parte, los ***AntiPatterns*** (antipatrones en español), son un tipo de defecto de diseño de alto nivel. Dependen principalmente del diseño de la arquitectura del sistema, el tipo de desarrollo del software y la tecnología utilizada [36]. Por sus características, los *AntiPatterns* se pueden considerar como complementarios a los patrones de diseño, ya que reflejan cómo los patrones de diseño están siendo mal utilizados [17]. Por ejemplo, mediante la aplicación de un patrón de diseño en un contexto distinto al descrito, o por la ausencia de un patrón en la solución otorgada. Asimismo, a medida que las tecnologías evolucionan, algunos patrones pueden volverse obsoletos [12], es decir, lo que alguna vez fue una buena solución puede convertirse en una mala solución.

2.2. Taxonomía de *Code Smells*

En 1999 Martin Fowler presenta un catálogo de 22 *Code Smells* en su libro "*Refactoring*" [17] en el contexto de programación orientada a objetos, que se describen en la tabla A.1. Posteriormente, estos *Code Smells* fueron referenciados como "*Code Smells* básicos" por otros autores que estudian a estos defectos de diseño. En la segunda edición de este libro ("*Refactoring*", 2018), se añade conocimiento sobre el estado actual de las técnicas de análisis de código, enfocado a la anticipación a problemas más profundos a nivel estructural, con el objetivo de proporcionar una guía a los desarrolladores de software para mejorar sus prácticas de codificación.

En 2003, en el estudio empírico "*Bad smells in software - a taxonomy and an empirical study*" [28], Mika V. Mäntylä presenta una taxonomía para la clasificación de los 22 *Code Smells* básicos según siete categorías principales. Esta taxonomía constituye un aporte académico fundamental en el desarrollo del estudio sobre la calidad interna del software dentro del desarrollo de la ingeniería de software. Asimismo, el estudio identifica las distintas relaciones entre los *Code Smells* básicos, teniendo en cuenta el contexto en el cual se incurren. De esta forma, se logra reconocer, categorizar, y aumentar la comprensión de los diferentes *Code Smells*, siempre teniendo en cuenta el mantenimiento de la calidad de código en el software.

A continuación se presentan las siete categorías que propone Mäntylä (2003) en su taxonomía:

- ***Bloaters***: *Code Smells* que representan alguna entidad que ha crecido

demasiado y se torna muy difícil trabajar con ella. Este suele constituir un problema de diseño que genera una acumulación con el paso del tiempo y la consecuente evolución del código. Se cree que los *Bloaters* crecen en pequeños pasos, hasta que pueden ser identificados. Es decir, no pueden ser detectados hasta que el programa evolucione.

- ***OOAbusers (OOA)***: *Code Smells* que involucran una aplicación errónea de los principios y posibilidades de diseño que ofrece la orientación a objetos. A nivel genérico, son clases con un mal manejo/diseño de herencias o de información, no respetando la privacidad de los atributos y datos.
- ***Change preventers***: *Code Smells* que dificultan el desarrollo o cambio del software. La refactorización de estos *Code Smells* suele implicar grandes cambios en el código, dificultando y encareciendo el desarrollo y/o evolución del software.
- ***Dispensables***: *Code Smells* que representan elementos innecesarios en el código, por falta de actividad, responsabilidad o redundancia. Estos defectos de diseño deben ser eliminados del código fuente. La ausencia de ellos, convierte al código del programa más entendible y eficiente.
- ***Encapsulators***: *Code Smells* relacionados con los principios de encapsulación (privacidad de datos) y/o con el mecanismo de comunicación (mediante una delegación excesiva).
- ***Couplers***: *Code Smells* que contribuyen a un acoplamiento excesivo entre las distintas clases del programa. Por lo general, violan el principio de bajo acoplamiento.
- ***Others***: *Code Smells* que no pertenecen a ninguna de las categorías anteriores. Estos *Code Smells* podrían tener o no puntos en común entre ellos.

En la figura 2.1 se diagrama la categorización de los *Code Smells* básicos según la taxonomía de Mäntylä utilizando la técnica de diagramas de Venn.

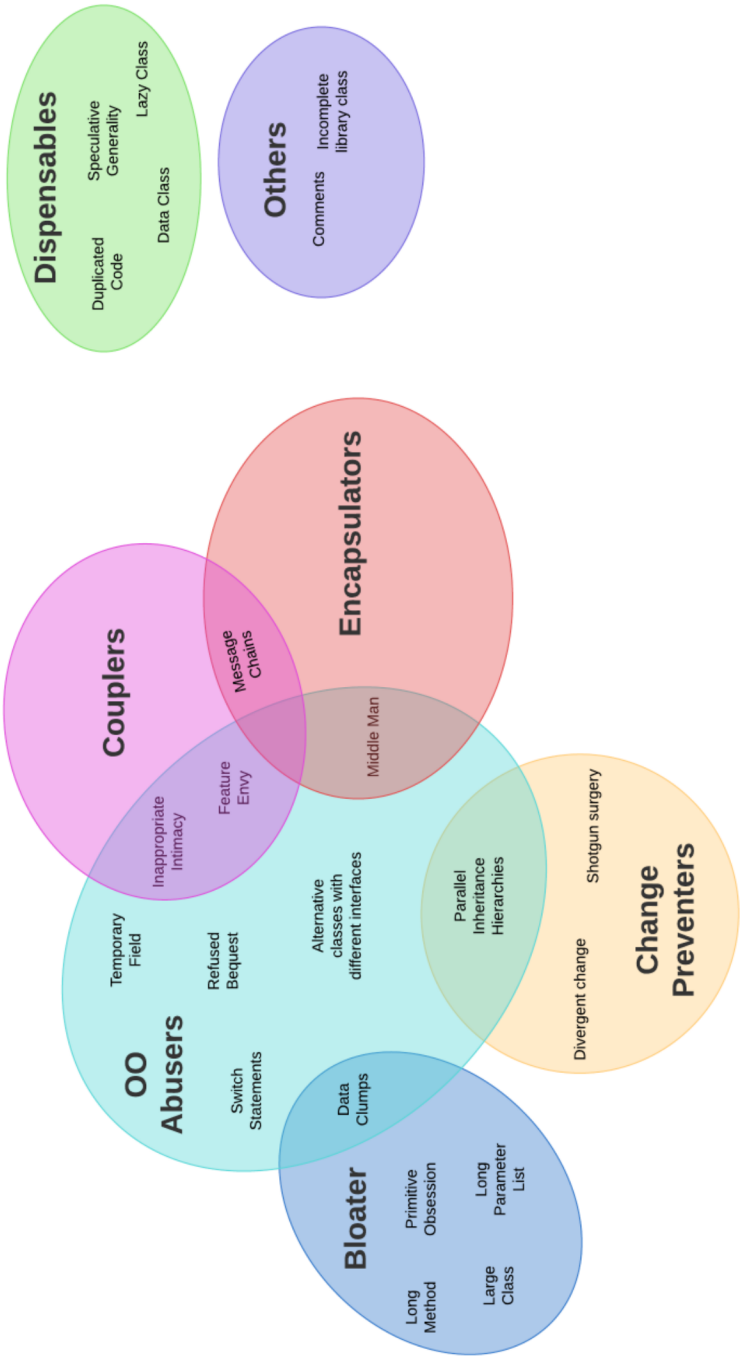

Figura 2.1: *Code Smells* básicos categorizados según la taxonomía Mäntylä [28].

2.3. Conjunto de *AntiPatterns*

Previamente se definió a los *AntiPatterns* como defectos de diseño de alto nivel. Son una "forma literaria que describe una solución común a un problema que genera consecuencias negativas" [42]. Sin embargo, resolverlos puede resultar en una mejora en la calidad interna del software [42].

Los patrones de diseño documentan soluciones de desarrollo de software recurrentes. La esencia de un patrón de diseño es un problema y una solución. Los patrones de diseño al ser utilizados en un contexto inapropiado (distinto al descrito en el patrón), "se convierten" en *AntiPatterns* [42], resultando en proyectos de software fallidos, costosos, retrasados en el cronograma, o con necesidades comerciales insatisfechas [26].

El estudio de AntiPatterns es una actividad de investigación fundamental, ya que la presencia de patrones de diseño en un software funcional no es suficiente para asegurar la calidad interna del mismo. Por otra parte, también se debe demostrar que esos patrones están ausentes en software fallidos.
—Jim Coplien

Para la definición del conjunto de *AntiPatterns* a utilizar en la investigación, se estudiaron los *AntiPatterns* pertenecientes al "Catálogo de *AntiPatterns*" de la Tesis de Maestría de Casella [9] y aquellos presentados en el libro *"AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis"* [42]. Sobre estos últimos, se decide no estudiar aquellos *AntiPatterns* clasificados como "de gestión", ya que no son de interés ni aplican al estudio de defectos de diseño en el marco de nuestra investigación. Estos *AntiPatterns* identifican algunos de los escenarios clave en los que se involucran los recursos humanos y la gestión de los mismos, principalmente en la resolución de conflictos entre miembros del equipo. Análogamente, considerando el catálogo de Vanessa Casella [9], se excluyen aquellos *AntiPatterns* clasificados como *Big Data*, ya que este tipo de *AntiPatterns* no son de interés por no estar relacionados a defectos de diseño de software o no son aplicables el análisis a realizar en nuestra investigación.

En la tabla A.2, se presenta brevemente los *AntiPatterns* que serán considerados en la investigación, nuestro conjunto de *AntiPatterns*.

Capítulo 3

Diseño experimental

La investigación implicó la realización de un estudio experimental de carácter exploratorio para el cual se usa como objeto de estudio un proyecto de estudiantes de la asignatura Proyecto de Ingeniería de Software (PIS) codificado en Java. Más precisamente, se busca explorar y observar cómo se distribuyen los *Code Smells* y *AntiPatterns* detectados, y a futuro ampliar la investigación con otros códigos, clientes, requerimientos funcionales u otros tipos de productos. Este capítulo presenta brevemente la toma de decisiones relacionadas al diseño experimental, y se describe de las características del código analizado.

Nuestro proyecto de grado busca conocer e investigar los defectos de diseño, *Code Smells* y *AntiPatterns*, en los que incurren estudiantes avanzados de la carrera. En esta línea, se detecta ausencia de antecedentes de investigación en cuanto a estos defectos de diseño incurridos por estudiantes de nuestra facultad. Como consecuencia, realizamos un estudio experimental de carácter exploratorio. Por cuestiones de tiempo y alcance del proyecto de grado, se decidió reducir el conjunto de proyectos PIS a analizar a un único proyecto. En este sentido, dada nuestra experiencia en tecnologías y lenguajes de programación, sumado a las limitaciones encontradas en las herramientas automatizadas de detección disponibles, y las capacidades de detección de las herramientas disponibles, se determina *Java* como la tecnología de preferencia. Adicionalmente, como consecuencia de la disponibilidad y rápido acceso a documentación proporcionada [11], se define utilizar un proyecto PIS desarrollado en Java del año 2017. Posteriormente, utilizaremos el término PIS_N para referirnos a dicho proyecto y al equipo de desarrollo que lo llevó adelante.

3.1. Contexto experimental

PIS es una asignatura obligatoria dictada en el octavo semestre de la carrera Ingeniería en Computación de Fing, UdelaR, lo cual nos permite pensar que los estudiantes cuentan con experiencia a nivel académico. La metodología de enseñanza del PIS implica la puesta en marcha de un proyecto de software guiado y controlado, sometido a diversos tipos de restricciones análogas a las que son comunes en la industria de software para un proyecto de mediano porte. El PIS logra un acercamiento entre la academia y la industria en los estudiantes

avanzados de la carrera, lo que se considera una ventaja en cuanto a realismo que se aporta al proyecto construido y por ende aporta valor a nuestro análisis.

El propósito del sistema construido por el equipo PIS_N fue crear un módulo de gestión de actividades de centros hospitalarios. En particular, se implementó un motor de *workflow* capaz de integrarse a una herramienta de administración de tareas y mensajería para empresas y organizaciones. En este motor se definen flujos de trabajo que reaccionan ante determinados eventos, ejecutando una secuencia de actividades. También se puede ejecutar una secuencia de actividades predefinida independientemente de estos flujos.

El proyecto PIS_N utiliza las tecnologías *React JS* en el *frontend* y *Java* en el *backend*. A modo de modularizar el sistema en base a las responsabilidades, la arquitectura del producto generado es una conjunción de tres patrones de diseño: *arquitectura publicar y suscribir*, *arquitectura tuberías y filtros*, *arquitectura en capas*. Las capas definidas por el equipo son: presentación, servicio (proporciona una API REST por requisito del cliente), negocios (contiene el motor de *workflows*), acceso a datos, transversal (define *datatypes* y enumerados).

Para llevar a cabo el proyecto PIS_N se utilizó el modelo de proceso iterativo e incremental MUM[1]. MUM está basado en RUP[2] que proporciona un enfoque de proceso de desarrollo iterativo e incremental, dividido en 4 fases (inicial, elaboración, construcción y transición) y adaptado a las 14 semanas de duración del PIS_N. El objetivo de MUM es asegurar la producción de software de calidad, que satisfaga las necesidades de los usuarios dentro de un cronograma y un presupuesto predecible [11]. La figura 3.1 muestra cómo varía el énfasis o dedicación con el paso del tiempo según RUP. La dimensión tiempo, expresada en términos de fases, iteraciones e hitos, muestra los aspectos de ciclo de vida del proceso y representa el aspecto dinámico del mismo.

[1] Modelo de Proceso Modularizado Unificado y Medible [11]
[2] *Rational Unified Process* [11]

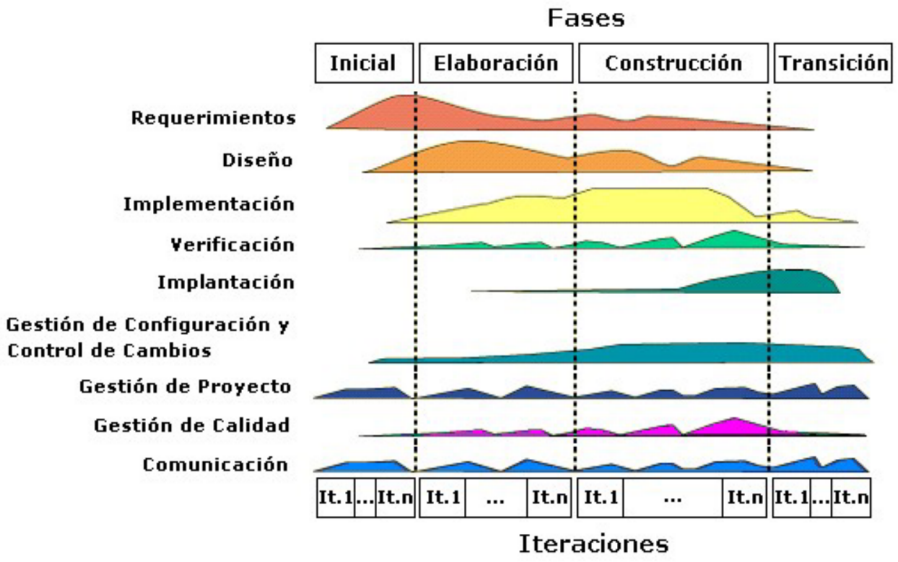

Figura 3.1: Dimensiones del Modelo del Proceso (Fases e Iteraciones y Disciplinas)

Las primeras cuatro semanas de PIS_N constituyeron la fase inicial. Desde la semana cinco a la semana nueve el proyecto se encontró en fase de elaboración. La tercera fase, la de construcción, se desarrolló entre las semanas 10 y 13. La última semana estuvo dedicada a realizar la fase de transición.

En la figura 3.2 se visualizan las disciplinas del proceso MUM y las horas de trabajo que se dedicó a cada una de las disciplinas en el proyecto PIS_N.

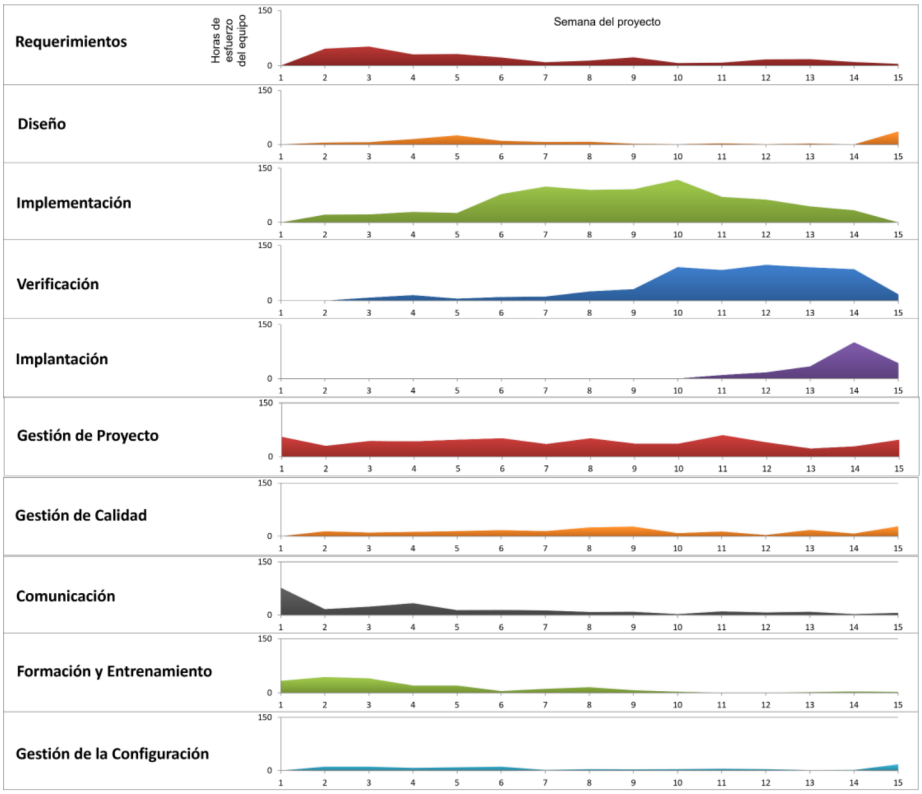

Figura 3.2: Horas dedicadas por disciplina en el proyecto PIS_N

En relación a los recursos humanos, el equipo que desarrolló el proyecto estuvo integrado por 11 estudiantes, 1 docente de la asignatura PIS con el rol de director de proyecto, y el cliente que solicita el sistema especificado. En la documentación de PIS_N se menciona que el cliente hizo hincapié principalmente en la *performance* del producto final.

En relación a la calidad del proyecto, el equipo realizó dos revisiones de código por parejas al *backend* siguiendo un estándar de implementación *Java* definido por ellos. Las revisiones estuvieron acotadas a algunas funcionalidades del sistema por cuestiones de tiempo y esfuerzo de los miembros del equipo. En la primera revisión se encontraron 47 errores y en la segunda 18, todos ellos corregidos según el equipo PIS_N. Analizando los resultados de estas revisiones, podemos observar que:

- los defectos encontrados fueron mayoritariamente sobre comentarios faltantes en el código.
- se detectaron múltiples porciones de código comentadas.
- en la capa de acceso a datos, se observaron varias clases vacías y con importaciones sin utilizar.
- se muestran mensajes de error en consola, en lugar de hacerlo en un archivo de registro de errores como fue requerido.

- se atrapan múltiples excepciones de manera genérica.

Con respecto al estándar de implementación definido por los estudiantes para la realización de las revisiones de código por parejas, se observa que la mayoría de las reglas incluidas en el estándar son convenciones, se asocian al buen uso de la tecnología o se relacionan con "buenas prácticas de programación".

A raíz del objetivo del proyecto de grado, se decide evaluar en profundidad la disciplina de diseño. A partir de la figura 3.2, se concluye que el diseño fue una de las disciplinas con menor dedicación semanal en PIS_N. Este resultado está alineado con el estudio realizado en 2018 por Moreno y Vallespir [25] en la Facultad de Ingeniería, UdelaR, el cual señala que estudiantes de pregrado avanzados en la carrera Ingeniería en Computación dedican cuatro veces menos tiempo al diseño de software en comparación con el tiempo dedicado a la implementación. En este sentido, interesa observar la relación del esfuerzo del equipo PIS_N, medido en horas de dedicación semanal, en las disciplinas de diseño e implementación. Esta relación se presenta en la figura 3.3

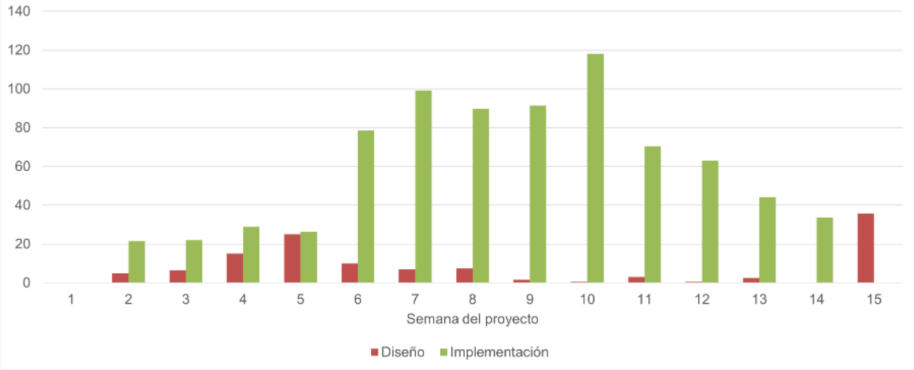

Figura 3.3: Horas dedicadas a diseño en comparación con las horas dedicadas a implementación en PIS_N

Observando y comparando la relación entre esfuerzo dedicado a implementación y diseño, con el esfuerzo sugerido para cada disciplina del MUM, nos preguntamos sobre las posibles razones por las cuales el esfuerzo dedicado a diseño fue tan bajo. Una de las causas de este resultado podría ser que haya un registro incorrecto del esfuerzo realizado por parte del equipo, por desconocer qué tareas implica esta actividad.

Al comparar la dedicación que brinda el MUM según el RUP contra la dedicación del proyecto PIS_N se pueden observar algunos desfases en las distintas etapas en cuanto al esfuerzo del equipo PIS_N. En especial, la fase de transición es la fase en la que más horas se dedicó al diseño. En principio esta particularidad captura nuestra atención ya que - según el MUM - en esta fase, la disciplina de diseño debería disminuir considerablemente hasta llegar a cero.

Página ha sido intencionalmente dejada en blanco.

Capítulo 4

Procesos de selección de herramientas y *Code Smells*

Con el fin de conocer los defectos de diseño, *Code Smells* y *AntiPatterns* en los cuales incurren los studiantes, se toma como base el conjunto de *Code Smells* detectados por *SonarQube* y los *plugins* seleccionados para la investigación. En especial, para la determinación de los *Code Smells* a investigar se definen ciertos criterios particulares. Se estudiarán aquellos *Code Smells* que cumplen con alguno de los siguientes criterios:

- Se relacionan con *AntiPatterns* de diseño.
- Son de carácter crítico en cuanto a esfuerzo requerido en su refactorización y nivel de gravedad establecidos por *SonarQube*.
- Son los más incurridos en el código a analizar. Es decir, son aquellos *Code Smells* con mayor cantidad de ocurrencias.

Con el objetivo de poder determinar por un lado las herramientas a utilizar, y por otro, el conjunto de *Code Smells* a analizar en el proyecto PIS_N, se definen dos procesos de investigación: el proceso de **selección de *Code Smells*** y el proceso de **evaluación y selección de herramientas**. Ambos procesos toman como base las preguntas de investigación, los *Code Smells* básicos de Fowler [16] y el conjunto de *AntiPatterns* que se presenta en el capítulo 2.

En la figura 4.1, se ilustran los procesos de investigación. En la parte superior del diagrama se describe el proceso de *evaluación y selección de herramientas*. El mismo consta de dos actividades principales; y retorna las herramientas a utilizar en la investigación y el conjunto de *Bugs, Vulnerabilities, Security Hotspots y Code Smells* detectados por ellas. A consecuencia del alcance de la investigación, el conjunto resultado obtenido es depurado. Solo se tendrán en cuenta los *Code Smells* detectados. En la sección inferior del diagrama 4.1 se observa el proceso de *selección de Code Smells*. Este proceso también consta de dos actividades fundamentales. A partir del conjunto de *Code Smells* detectados retornado por el proceso de *evaluación y selección de herramientas* y los criterios de selección de *Code Smells* establecidos, el proceso de *selección de Code Smells* retorna un conjunto reducido de *Code Smells* a estudiar en la investigación.

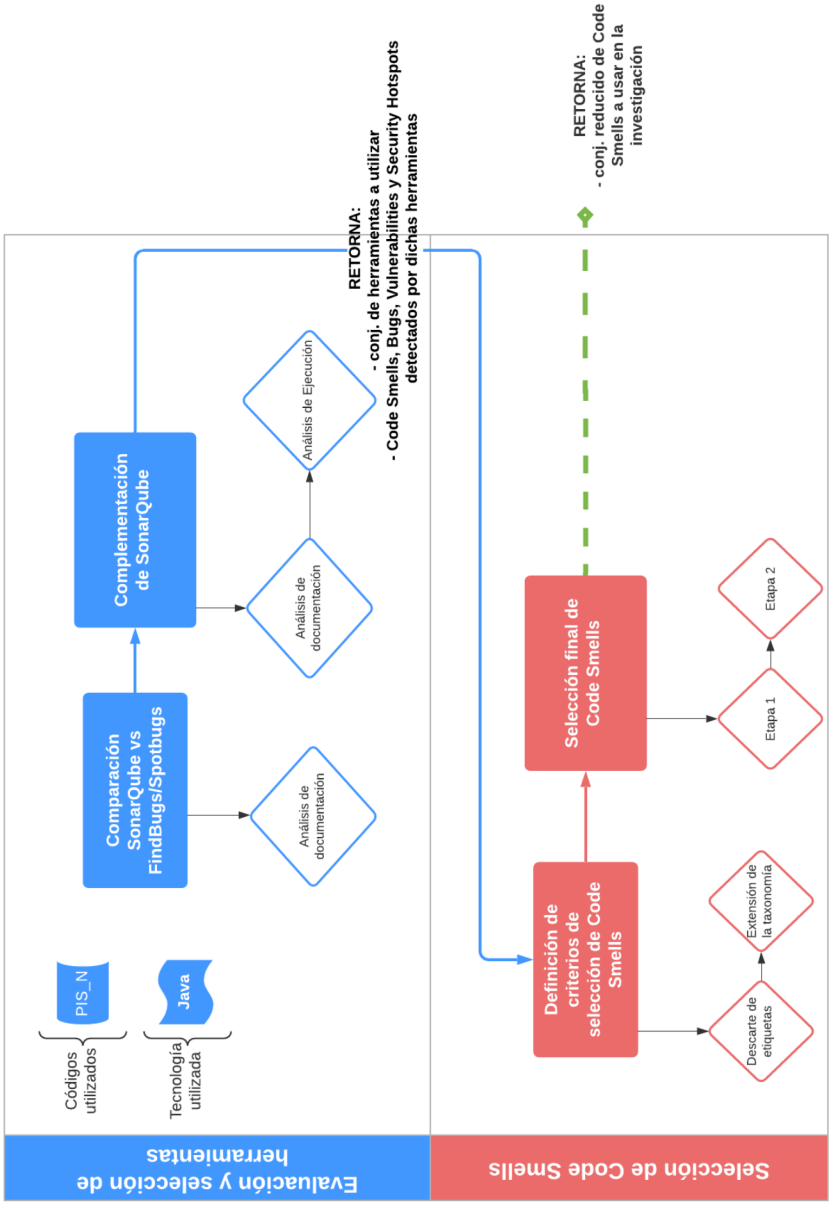

Figura 4.1: Los procesos definidos para la selección de *Code Smells* y herramientas para su detección.

En las siguientes secciones, se describe en detalle los dos procesos mencionados anteriormente y cómo éstos se relacionan entre sí. En la última sección del capítulo, bajo el subtítulo *proceso de ejecución*, se encuentran detallados los pasos a seguir para llevar a cabo la ejecución de las herramientas seleccionadas en base al conjunto resultado de *Code Smells* obtenido.

4.1. Evaluación y selección de herramientas

Para la realización del proceso de *evaluación y selección de herramientas* y un correcto análisis posterior, se debe comprender con cierto grado de profundidad el lenguaje de programación a utilizar y el contexto en el cual se incurre en los defectos de diseño a estudiar. Dados estos requisitos, considerando nuestra experiencia a nivel laboral y académica, se define el lenguaje Java[1] como la tecnología a utilizar. Por otra parte, se decide utilizar el proyecto PIS_N codificado en el lenguaje mencionado.

El proyecto de grado plantea como requisito el uso de herramientas automatizadas disponibles en el mercado para la detección de *Code Smells* y *AntiPatterns*. En este marco, se realizan revisiones de literatura donde *SonarQube*[2] y en segundo lugar *FindBugs*[3] (posteriormente renombrada como *SpotBugs*) resultan ser las herramientas más utilizadas para el análisis de código estático tanto en la industria como en la academia [40, 44].

SonarQube y *SpotBugs*, cuentan con disponibilidad de documentación técnica, investigaciones empíricas y revisiones sistemáticas de la literatura [41, 44]. La tabla comparativa A.3 resume las características más relevantes de ambas herramientas en relación a nuestra investigación. La última fila de la tabla A.3 muestra los resultados de las ejecuciones de *SonarQube* y *SpotBugs* en el proyecto PIS_N. Al comparar los resultados obtenidos se observa que *SonarQube* detecta aproximadamente tres veces más *Code Smells* que *SpotBugs*. Adicionalmente, el análisis en detalle de las ocurrencias de algunos *Code Smells* según la herramienta de detección demuestra que existen ocurrencias detectadas por *SonarQube* y no por *SpotBugs*. Ejemplos de *Code Smells* detectados por *SonarQube* y no por *SpotBugs* son:

- *"Cognitive Complexity of methods should not be too high"*: refiere a la complejidad cognitiva de un método. Múltiples clases contienen ocurrencias de este *Code Smell*, por ejemplo, *ForEachBlockActivity.java y SendMailActivity.java*.

- *"Source files should not have any duplicated blocks"*: detecta bloques de códigos duplicado en una misma clase. Múltiples clases contienen ocurrencias de este *Code Smell*, por ejemplo, *EventListener.java y ConditionalActivity.java*.

En la actualidad, *SonarQube* es utilizado por más de 200.000 organizaciones que, a cambio, brindan comentarios y contribuciones regulares[4]. Este punto,

[1] https://www.java.com/es/
[2] https://www.sonarqube.org/
[3] http://findbugs.sourceforge.net/
[4] Datos extraídos de https://community.sonarsource.com

sumado a la comunidad, la documentación de la herramienta y la experiencia laboral de los miembros del equipo del proyecto de grado, motivaron el uso de *SonarQube* como herramienta principal en la investigación. Por otra parte, luego del análisis y comparación de ambas herramientas, se concluye que para nuestro trabajo *SpotBugs* es notoriamente inferior en los aspectos analizados y no complementa en ningún sentido a *SonarQube*.

4.1.1. Complementación de *SonarQube*

Luego de determinar a *SonarQube* como la herramienta principal a utilizar en el proyecto de grado, se busca complementar a *SonarQube* con el objetivo de detectar la mayor cantidad de *Code Smells* y *AntiPatterns* en el proyecto PIS_N. En esta línea, surgen de la revisión de literatura otras herramientas menos populares que podrían ser complementarias a *SonarQube*. Con el objetivo recién mencionado, luego de consultar múltiples fuentes bibliográficas, se analizan las herramientas *JDeodorant, JCodeOdor y JSpirit*. Sin embargo, a partir del análisis de la documentación proporcionada por las herramientas, se descartan como complementarias a *SonarQube*. Para ejemplificar, se cita como ejemplo los motivos que llevaron a descartar el *plugin JSpirit*[5]. Por un lado, *JSpirit* no se encuentra disponible en *Eclipse Marketplace*[6], a pesar de ser por definición un *plugin* para este IDE[7]. Adicionalmente, no se encuentra disponible el enlace de descarga proporcionado en la escasa documentación oficial de la herramienta. Como consecuencia, el *plugin JSpirit* se descarta.

De forma similar, se intenta complementar *SonarQube* con los *plugins* oficiales a sugerencia de las tutoras (PMD [8] y *CheckStyle* [9]). Con el objetivo de complementar a *SonarQube* en niveles más abstractos de *Code Smells* enfocados en la detección de *AntiPatterns*, se estudia el *plugin Code Smells and AntiPatterns detection*[10]. En cuanto a los tres *plugins* a considerar, se analiza la documentación y además se ejecutan en el código PIS_N. En la tabla A.4, a modo de ejemplo, se detallan algunos *Code Smells* detectados que complementan a *SonarQube*.

Analizando en detalle los *Code Smells* detectados en PIS_N por los *plugins*, se observa que existen múltiples *Code Smells* que son de utilidad para responder a las preguntas de investigación del proyecto de grado. Como consecuencia, se decide incluir los tres *plugins* estudiados como complemento a *SonarQube*.

En resumen, el proceso de *evaluación y selección de herramientas* permite:

1. definir la tecnología y el conjunto de herramientas a utilizar en el proyecto de investigación. Se utiliza el lenguaje de programación *Java*, la herramienta *SonarQube* y los *plugins*: *PMD*, *CheckStyle*, *Code Smells and AntiPatterns detection*.

2. conocer el conjunto de *Code Smells, Bugs, Vulnerabilities, Security Hotspots* de las herramientas seleccionadas. En el marco de nuestro proyecto

[5] https://sites.google.com/site/santiagoavidal/projects/jspirit
[6] https://marketplace.eclipse.org/
[7] entorno de desarrollo integrado
[8] https://pmd.github.io/
[9] https://CheckStyle.sourceforge.io/
[10] https://github.com/davidetaibi/sonarqube-anti-patterns-code-smells

de grado, sólo se consideran los *Code Smells* de este conjunto.

Cabe destacar que con el fin de visualizar los resultados de las ejecuciones de las herramientas de forma centralizada se realizó la instalación de *SonarQube* en un servidor proporcionado por la FIng, UdelaR. El detalle de los pasos de instalación se encuentra en la sección A.2.

4.2. Selección de Code Smells

El proceso de *selección de Code Smells* tiene como objetivo acotar el conjunto de *Code Smells* a estudiar, descartando aquellos *Code Smells* que no sean de interés para la investigación. El proceso de selección de *Code Smells* cuenta de dos actividades:

1. la definición de nuestros criterios de selección de *Code Smells*.
2. la selección de *Code Smells*. Se consideran aquellos disponibles en las herramientas de detección a utilizar.

4.2.1. Definición de los criterios de selección de *Code Smells*

Previo a la definición de los criterios de selección de *Code Smells*, se realiza un estudio detallado de la literatura existente para comprender los conceptos teóricos, definiciones, terminología y posibles categorizaciones relacionadas a los defectos de diseño; descrito en el capítulo 2.

SonarQube ejecuta reglas sobre el código fuente. Utiliza etiquetas como una forma de clasificar las reglas y ayudar a describirlas más fácilmente. Un gran porcentaje de las reglas definidas tienen múltiples etiquetas asociadas. Con el fin de poder establecer un primer filtro sobre el conjunto de *Code Smells*, se analiza en detalle estas etiquetas y las características de los *Code Smells* pertenecientes a ellas. En una primer instancia, se descartan los *Code Smells* asociados exclusivamente a etiquetas que no son de interés para la investigación. En la tabla A.5 se detallan las etiquetas de *SonarQube* descartadas.

Dentro de las múltiples etiquetas definidas por *SonarQube*, se analizan en profundidad dos de ellas dado que su aplicabilidad a la investigación, o relación con defectos de diseño, no resulta tan clara como con otras etiquetas.

- Expresiones regulares. El uso de expresiones regulares depende del tipo de software a construir, es decir, su aplicabilidad en el software que se implementa. Por ejemplo, las expresiones regulares resultan de gran utilidad para realizar la comprobación de la correctitud de los datos rellenados por los usuarios en un formulario. Se observa que al realizar búsquedas de ciertas palabras clave asociadas al uso de expresiones regulares en el código fuente, como *"java.util.regex"*, no se detectan coincidencias. Se concluye que el proyecto PIS_N no utiliza expresiones regulares. Adicionalmente, se ejecuta *SonarQube* sobre el código PIS_N únicamente con las reglas que contienen la etiqueta de expresiones regulares para analizar ocurrencias. Como resultado, no se encuentran ocurrencias.

- Hilos. Análogamente, se ejecuta *SonarQube* sobre el código PIS_N con las reglas que contienen etiquetas relacionadas a hilos (*threading, multi-threading*) para analizar ocurrencias. El resultado arroja que sólo hay un *Code Smell "Instance methods should not write to "static"fields"* (de un total de 30 *Code Smells* para esta etiqueta) relacionado al uso de hilos, detectando siete ocurrencias del mismo en el proyecto PIS_N. Analizando sus ocurrencias, se observa que está relacionado al buen uso de la tecnología, por lo que se decide no incluir la etiqueta, pero si incluir el *Code Smell* asociándolo a la categoría *Java Technology*. Por estos motivos, se descarta incluir *Code Smells* de hilos en el conjunto de *Code Smells* a detectar.

Posterior al descarte de etiquetas, se intenta clasificar uno a uno los *Code Smells* seleccionados usando la taxonomía propuesta por Mäntylä [28]. Durante la clasificación surge la necesidad de incluir nuevas categorías a la taxonomía de Mäntylä, ya que existen algunos *Code Smells* que no se relacionan a las categorías allí definidas. Por ejemplo, el *Code Smell "Method names should comply with a naming convention"* no pertenece a ninguna de las categorías de la taxonomía. No es considerado complejo (*Bloaters*), no está asociado a elementos innecesarios en el código (*Dispensables*), no está asociado a principios de orientación a objetos (*OOAbusers*) ni acoplamiento (*Couplers*), tampoco está relacionado a encapsulamiento ni comunicación (*Encapsulators*) y por último no es un *Code Smell* que dificulte el cambio (*Change preventers*). Cabe destacar que la categoría *Others* de la taxonomía de Mäntylä no resulta relevante a nuestra investigación por ser una categoría muy general que no incluye características particulares asociadas a los *Code Smells*. Como consecuencia, *Others* no será tenida en cuenta en la taxonomía extendida.

A continuación se detallan las nuevas categorías a considerar en la extensión de la taxonomía.

- ***Conventions***. Hace referencia a aquellas convenciones generales establecidas para *Java* y otros lenguajes de programación. Algunos ejemplos de *Code Smells* detectados pertenecientes a esta categoría son aquellos asociados a convenciones respecto a nombres de clases, métodos y variables.

- ***Performance***. *Code Smells* relacionados a la eficiencia en *Java*. En particular, se vincula con los tiempos de respuesta y de procesamiento del sistema.

- ***Exceptions***. *Code Smells* relacionados al manejo de excepciones en la tecnología *Java*. Se cree que este tipo de *Code Smells* pueden estar relacionados a la experiencia del programador en la industria.

- ***Java Technology***. Engloba aquellos *Code Smells* que tienen que ver directamente con el buen uso de la tecnología. Resulta de utilidad el conocimiento de los avances en las distintas versiones de *Java*, para poder aprovechar las mejoras que estos *releases* proveen. Creemos que este tipo de *Code Smells* están relacionados a la experiencia del programador en el uso de la tecnología. Por lo general, no son simples de corregir ya que requieren un conocimiento en profundidad del lenguaje en sí para hacer un buen uso de los recursos y posibilidades que brinda. Como ejemplo

de esta categoría esta el *Code Smell "try with resources"*. Es detectado para versiones de *Java* superiores a la versión 6 y es una "mejora" a la utilización de *"try/catch/finally"*.

- ***Good coding practices***. Abarca todos aquellos *Code Smells* que evidencian malas prácticas de programación.
- ***Clarity and readability***. Categoría que hace referencia a aquellos *Code Smells* que evidencian la ausencia de una solución comprensible, descriptiva, bien indentada. Son *Code Smells* que tienden a ser simples de encontrar y corregir.
- ***Security***. *Code Smells* relacionados al manejo de elementos de seguridad en la tecnología *Java*.

Utilizando la técnica de diagramas de Venn, en la figura 4.2 se visualiza la taxonomía extendida a utilizar en la investigación. En color rojo se visualizan los nombres de las nuevas categorías a considerar en la taxonomía.

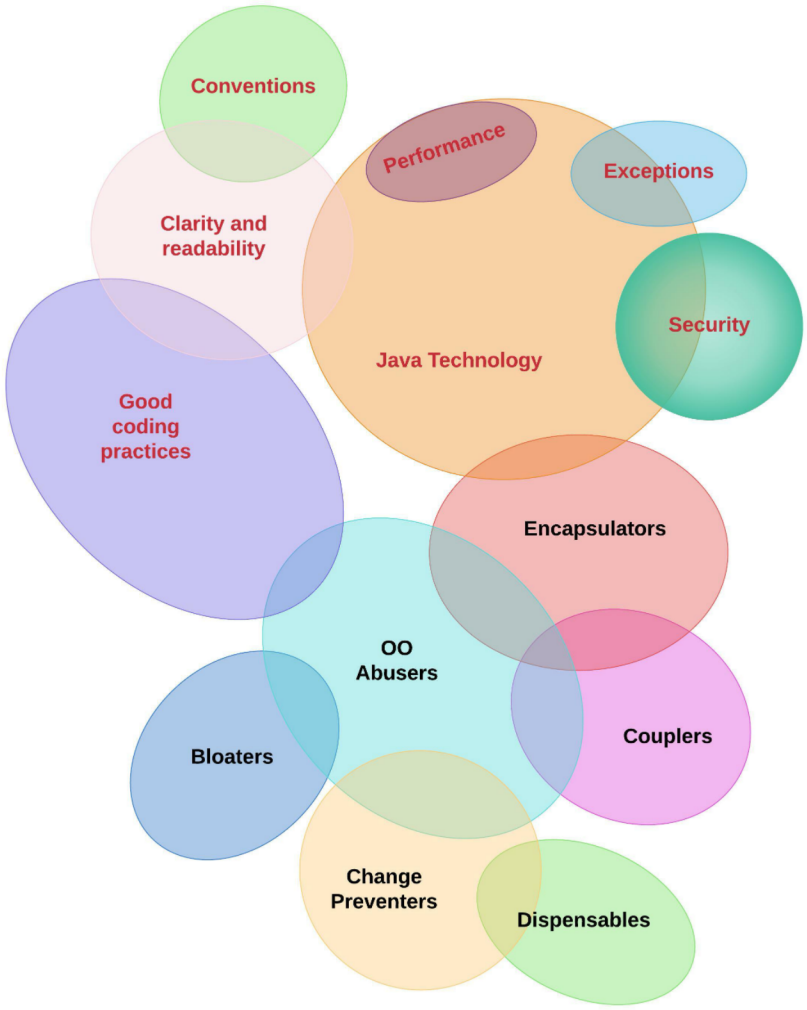

Figura 4.2: Taxonomía extendida.

En la categorización individual que se hizo de los *Code Smells* detectados por las herramientas, se analizaron las características de cada uno. A raíz de este análisis no solo surge la extensión de la taxonomía, sino que también los criterios específicos a considerar para la selección de los *Code Smells*. A continuación se detallan dichos criterios.

- *Code Smells* que **no** contienen etiquetas de *SonarQube* que **no** nos interesa analizar. Las etiquetas descartadas son: expresiones regulares, hilos, y aquellas pertenecientes a la tabla A.5.
- *Code Smells* que pertenezcan a alguna de las categorías de la taxonomía extendida, figura 4.2.
- En caso de *Code Smells* detectados por los *plugin* seleccionados para *SonarQube*, nos interesan solo aquellos que efectivamente tengan relación con los *AntiPatterns* definidos en la tabla A.2.
- *Code Smells* complementarios. Para los *Code Smells* de los *plugins* que sean similares a los de *SonarQube* se estudia si son repetidos o complementarios a partir de las ocurrencias, el algoritmo de detección y falsos positivos.

4.2.2. Selección del conjunto final de *Code Smells* a partir de aquellos disponibles en las herramientas de detección escogidas

La actividad de selección final de *Code Smells*, figura 4.3, consiste principalmente en la aplicación de los criterios de selección recientemente mencionados. La actividad consta de dos etapas, la **etapa uno** se basa en la aplicación de la mayoría de los criterios de selección definidos, obteniendo un conjunto reducido de *Code Smells*. La **etapa dos**, toma como entrada al conjunto reducido y analiza aquellos *Code Smells* que sean similares entre sí. En caso de que no sean complementarios, se descarta el *Code Smell* que menos aporte. De esta forma, al finalizar el proceso de selección de *Code Smells* se obtiene el conjunto reducido final de *Code Smells* a usar en la investigación. Esos *Code Smells* serán los que se incluyen en el *Quality Profile*[11] a utilizar en la investigación.

[11] Componente fundamental de *SonarQube* donde se definen las reglas que se considerarán *issues* en el código analizado. Se reporta un *issue* cuando un fragmento del código analizado rompe alguna regla de codificación del *Quality Profile* asociado al proyecto.

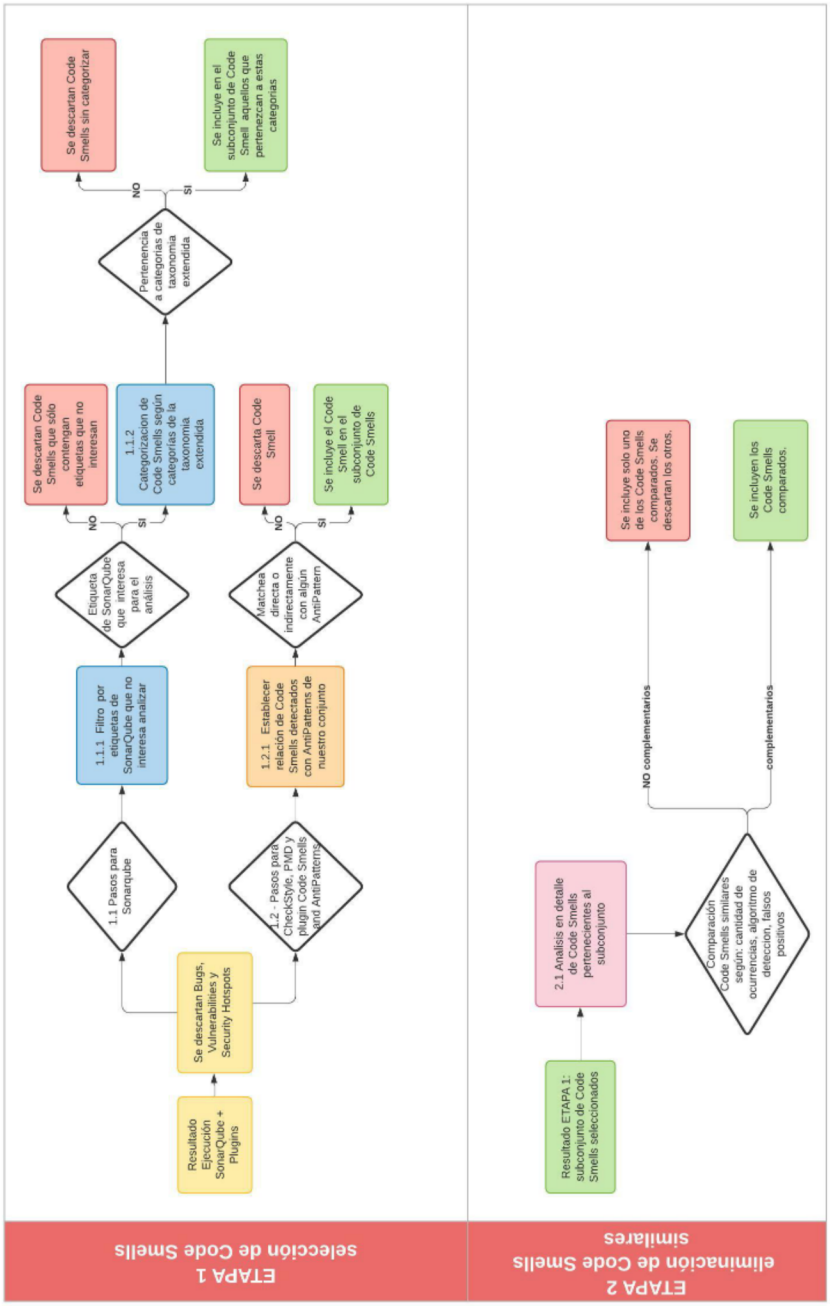

Figura 4.3: Proceso de selección de *Code Smells*.

A continuación se detalla cada una de las etapas del proceso de *selección de Code Smells*.

1. **Etapa uno**: selección de *Code Smells*. Tomando como entrada el conjunto resultado del proceso de *evaluación y selección de herramientas* (sección 4.1), se descartan *Bugs, Vulnerabilities, Security Hotspots* del conjunto resultado. Como consecuencia, se obtiene un conjunto únicamente conformado por los *Code Smells* detectados por las herramientas seleccionadas.

   1.1 **Pasos a seguir en SonarQube**.

   1.1.1 Filtro por etiquetas no relevantes para el análisis.
   A modo de establecer un primer filtro, se descartan del subconjunto de *Code Smells* aquellos que únicamente contengan etiquetas que **no** interesan a la investigación. Cabe destacar que aquellos *Code Smells* que poseen múltiples etiquetas, entre ellas alguna de interés, no quedarán descartados.

   1.1.2 Categorización de *Code Smells* según categorías de la taxonomía extendida.
   Se categoriza cada *Code Smells* según la taxonomía extendida. Se descarta del subconjunto de *Code Smells* aquellos que **no** pertenecen a categorías de la taxonomía extendida.

   1.2 **Pasos a seguir para *PMD, CheckStyle, Code Smells and AntiPatterns detection***.
   El objetivo principal de incluir extensiones a la herramienta *SonarQube* es buscar *Code Smells* que se vinculen con *AntiPatterns*.

   1.2.1 Establecer relación con *AntiPatterns*.
   Tomando como base el conjunto de *AntiPatterns* definido en la sección 2.3, se busca establecer una relación de cada *Code Smell* con *AntiPatterns*. Para estudiar la relación entre los *Code Smells* detectados y *AntiPatterns* de nuestro conjunto, se define una escala de tres valores y luego se realiza el mapeo.
   - *Match* directo.
   - *Match* Indirecto. En este caso en particular, se determina que la vinculación indirecta es cuando existe algún nivel de relación entre el *Code Smell* y un *AntiPattern*.
   - Ausencia de *match*.

2. **Etapa dos**: eliminación de *Code Smell* similares.

   Se analiza en detalle los *Code Smells* similares comparando la cantidad de ocurrencias, algoritmo de detección utilizado y la detección de falsos positivos. En base a estos tres puntos se determina si los *Code Smells* son complementarios o no. En caso de no ser complementarios, solo uno de ellos pasa a formar parte del subconjunto de *Code Smells* a utilizar en la investigación. El *Code Smell* repetido es descartado. A modo de ejemplo, el *Code Smell* de *PMD "NPath complexity"* fue descartado ya que sus 11 ocurrencias están incluidas en las 14 ocurrencias del *Code Smell "NPath complexity"* de *CheckStyle*.

En la tabla *Code Smells* seleccionados, pertenecientes al *Quality Profile* en *SonarQube* [3], se detallan todos los *Code Smells* seleccionados a utilizar en la investigación. Adicionalmente, la tabla *Code Smells* seleccionados, pertenecientes al *Quality Profile* en *SonarQube* categorizados [4], categoriza dichos *Code Smells*. Por último, la tabla *Code Smells* seleccionados, pertenecientes al *Quality Profile* en *SonarQube* mapeados con los *AntiPatterns* del conjunto [5], detalla el resultado del mapeo realizado entre *Code Smells* seleccionados y los *AntiPatterns* del conjunto a considerar.

4.3. Interrelación entre los procesos de investigación

El proceso de *evaluación y selección de herramientas*, se da en simultáneo con el proceso de *selección de Code Smells*. Además de ejecutarse en paralelo, ambos procesos se interrelacionan en mayor o menor medida en distintas instancias.

Para la realización de la extensión de la taxonomía propuesta por Mäntylä (figura 4.2), fue fundamental utilizar los resultados del análisis de ejecución de las herramientas seleccionadas del proceso de *evaluación y selección de herramientas*. De esta forma, se tiene en cuenta las etiquetas de *SonarQube* que se podrían considerar como categorías de la extensión e identificar categorías faltantes que resulten de interés según los *Code Smell* detectados.

Aplicando un razonamiento análogo al del punto anterior, se utiliza el conjunto de *Code Smells, Bugs, Vulnerabilities, Security Hotspots* resultado del proceso de *evaluación y selección de herramientas*. Dicho conjunto se refina quedándonos únicamente con los *Code Smells*. Para la selección final de *Code Smells* se aplican los criterios de selección definidos. De esta forma, se obtiene el subconjunto final de *Code Smells* a utilizar.

En conjunto, los procesos de investigación detallados en este capítulo retornan las herramientas y el conjunto final de *Code Smells* a utilizar en la investigación. Este conjunto final de *Code Smells* determinará la configuración del *Quality Profile* a aplicar en la investigación.

4.4. Configuración de las herramientas de detección

El *Quality Profile* es uno de los componentes principales de *SonarQube*, allí se definen las reglas que se deben cumplir en el código fuente a analizar. Cuando estas reglas son violadas se producen *issues*, clasificados en los siguientes tipos:

- *Bug*: error de codificación que debe corregirse inmediatamente. Puede que el error no haya ocurrido aún, pero en algún momento lo hará. Un ejemplo claro es *Null pointers should not be dereferenced*, una referencia a *null* nunca debería ser accedida ya que determinaría un *NullPointerException*.

- *Vulnerability*: punto inseguro del código, que está predispuesto a ser atacado, es necesario aplicar un cambio de forma inmediata. Estos *issues* son definidos y contabilizados bajo la métrica de *SonarQube security*.

- *Security Hotspots*: es similar a una vulnerabilidad ya que también hace referencia a la seguridad. La diferencia es la gravedad del *issue*. Un *Security*

Hotspot necesita ser analizado por el desarrollador para determinar si es necesario realizar algún cambio.

- *Code Smell*: refiere a una sección del código que tiene principalmente problemas de mantenimiento, hace que el código sea confuso y difícil de mantener.

Por otra parte, *SonarQube* brinda la posibilidad de definir etiquetas y asociarlas a los *issues*, como una forma de poder categorizarlos. Por defecto, *SonarQube* define un conjunto de categorías y establece una categorización, la cual puede ser modificada por los usuarios.

4.4.1. Descripción del *Quality Profile* resultado

En la tabla 4.1 se presenta la distribución en categorías de los *Code Smells* asociados a *SonarQube* luego del proceso de selección. Dentro del *Quality Profile* resultado se incluyen 280 *Code Smells* de *SonarQube*, de los cuales una parte importante refiere a *Code Smells* relacionados al correcto uso de la tecnología, al correcto uso de estándares, el uso de convenciones y de claridad y legibilidad del código.

Categorías	**Cantidad de *Code Smells* seleccionados por categorías**
Java Technology	122
Good coding practices	36
Conventions	35
Clarity and readability	31
Dispensables	28
Performance	23
Exceptions	22
Bloaters	16
OOAbusers	10
Encapsulators	8
Change Preventers	6
Couplers	1
Security	1

Tabla 4.1: Distribución de los *Code Smells* de *SonarQube* por categorías de la Taxonomía extendida, pertenecientes al *Quality Profile*.

Cabe destacar que si bien la suma de la cantidad de *Code Smells* de *SonarQube* incluidos en el *Quality Profile* por categoría es 339, la cantidad de *Code Smells* de *SonarQube* incluidos en el *Quality Profile* es 280. Esta diferencia se debe a que existen *Code Smells* categorizados en más de una categoría de la taxonomía extendida.

Por otra parte, en la tabla 4.2 se presenta la distribución de *Code Smells* asociados a los *plugins* durante el proceso de selección. En esta tabla se detalla la cantidad de *Code Smells* al inicio del proceso, la cantidad de *Code Smells*

relacionados a *AntiPatterns*, la cantidad de *Code Smells* repetidos y la cantidad de *Code Smells* incluidos en el *Quality Profile*.

Plugin	**Cantidad de *Code Smells* considerados al inicio**	**Cantidad de *Code Smells* relacionados a *AntiPatterns***	**Cantidad de *Code Smells* repetidos**	**Cantidad de *Code Smells* incluidos en el *Quality Profile* resultado**
Code Smells and AntiPatterns detection	17	17	5	12
PMD	91	8	5	3
CheckStyle	161	15	13	2

Tabla 4.2: Distribución de *Code Smells* asociados a los *plugins* luego del proceso de selección

Es preciso mencionar que los *Code Smells* cuantificados en la columna de relación a *AntiPatterns* son aquellos que se relacionan directa e indirectamente. La columna *Code Smells* repetidos informa acerca de aquellos *Code Smells* para los cuales se encontró un *Code Smell* similar en *Sonarqube*, tal como se detalló en la sección 4.2.

A diferencia de la tabla anterior, la tabla 4.3 detalla la distribución de *Code Smells* por categorías incluidos en el *Quality Profile* para cada *plugin*.

Plugin	**Categorías**	**Cantidad de *Code Smells* seleccionados por categorías**
Code Smells and AntiPatterns detection	*OOAbusers*	5
	Dispensables	3
	Bloaters	2
	Change Preventers	2
	Couplers	2
	Good coding practices	2
	Encapsulators	1
PMD	*OOAbusers*	2
	Good coding practices	2
	Encapsulators	1
CheckStyle	*Bloaters*	1
	Exceptions	1

Tabla 4.3: Distribución de *Code Smells* por categorías de la Taxonomía extendida de los *Code Smells* de los *plugins* pertenecientes al *Quality Profile*

4.4.2. Ejecución del *Quality Profile* resultado

Se obtiene como resultado del proceso de selección de *Code Smells* un nuevo *Quality Profile* compuesto por *Code Smells* de *SonarQube* y de los *plugin* asociados como complementarios a la herramienta de referencia (*SonarQube*) [3].

SonarQube permite la definición de nuevas etiquetas. Haciendo uso de esta funcionalidad, se configura *SonarQube* de forma de etiquetar los *Code Smells* a sus correspondientes categorías de la taxonomía extendida de Mäntylä (figura 4.2). De esta forma, se facilita el análisis de los resultados de la ejecución del proyecto PIS_N con el *Quality Profile* configurado. Al ejecutar la herramienta se obtiene información de interés agrupada por etiquetas, en este caso, categorías de la taxonomía extendida. Entre los datos de interés se destacan, por ejemplo, la cantidad de *Code Smells* asociados a una categoría y la deuda técnica asociada a una categoría.

Como se mencionó anteriormente, en la tabla 4.2 se visualiza una columna que detalla la relación directa e indirecta entre *Code Smells* detectados y *AntiPatterns*. Con el fin de obtener información categorizada de interés en este sentido, se realiza el mismo proceso de etiquetado categorizando los *Code Smells* en *SonarQube* según su relación directa/indirecta/ausente con *AntiPatterns*.

Página ha sido intencionalmente dejada en blanco.

Capítulo 5

Resultados, análisis y discusión

En este capítulo se presenta el análisis realizado y los resultados obtenidos en la detección de defectos de diseño, *Code Smells* y *AntiPatterns*, incurridos por estudiantes de PIS_N, detectados por el conjunto de herramientas seleccionadas. Los resultados se presentan respondiendo a las diferentes preguntas de investigación, capítulo 1, planteadas para el proyecto de grado.

A continuación se especifican algunos términos y expresiones a los que se hará referencia a lo largo del capítulo.

- Total de *Code Smells*: conjunto de *Code Smells seleccionados para la investigación* [3].
- *Code Smell* detectado: un *Code Smell* se detecta cuando es reportado por las herramientas seleccionadas en el proyecto PIS_N. Todo *Code Smell* detectado tiene al menos una ocurrencia en el proyecto analizado.
- Densidad de *Code Smell*: en una capa de la arquitectura o en una clase. Hace referencia al resultado del cálculo de la cantidad de ocurrencias de *Code Smells* sobre líneas de código (*LOCs*, del inglés *lines of code*) de la capa o clase (CS/LOC).

5.1. Respuestas a las preguntas de investigación

5.1.1. RQ1: ¿En qué *Code Smells* incurren los estudiantes?

Para responder esta pregunta analizaremos la distribución de las ocurrencias de *Code Smells* por categorías, los *Code Smells* con mayor cantidad de ocurrencias, y la relación entre la cantidad de *Code Smells* detectados por categorías y su cantidad de ocurrencias.

Code Smells detectados en el proyecto PIS_N distribuidos por categorías.

En la tabla "*Code Smells* seleccionados para la investigación categorizados" [4] se observan los *Code Smells* seleccionados, su cantidad de ocurrencias y su clasificación en la taxonomía extendida.

La tabla 5.1 detalla para cada categoría la cantidad de *Code Smells* incluidos en el *Quality Profile*, la cantidad de *Code Smells* detectados, los porcentajes de *Code Smells* detectados del total de *Code Smells* y de *Code Smells* detectados del total de detecciones.

Categoría	**Code Smells incluidos en el *Quality Profile***	***Code Smells* detectados**	**% *Code Smells* detectados del total de *Code Smells***	**% *Code Smells* detectados del total de *Code Smells* detectados**
Java Technology	122	25	20,4 %	25,8 %
Conventions	35	19	54,3 %	19,6 %
Dispensables	31	16	51,6 %	16,5 %
Good Coding Practices	40	12	30 %	12,4 %
Exceptions	23	11	47,8 %	11,3 %
Clarity and Readability	31	8	25,8 %	8,2 %
Bloaters	19	8	42,1 %	8,2 %
Change Preventers	8	4	50 %	4,1 %
Encapsulators	10	3	30 %	3,1 %
OOAbusers	17	3	17,6 %	3,1 %
Performance	23	2	8,7 %	2,1 %
Couplers	3	1	33,3 %	1 %
Security	1	1	100 %	1 %

Tabla 5.1: Relación entre *Code Smells* seleccionados en el *Quality Profile* y los *Code Smells* detectados en el proyecto PIS_N

Se observa que *Java Technology* es la categoría con más *Code Smells* detectados. Sin embargo, teniendo en cuenta el total de *Code Smells* para esta categoría, solo se detecta un 20,4 %. Existe un 79,6 % de *Code Smells* no detectados en esta categoría, cabe destacar que existen varios motivos por los cuales puede no detectarse un *Code Smell* en el código. Por ejemplo, puede que no se incurra en el *Code Smell* ya sea por haber aplicado buenas prácticas de programación o que el código fuente no sea propenso a contenerlo. El *Code Smell "deleteOnExit should not be used"* es un ejemplo del segundo caso. Este *Code Smell* se enfoca en el uso de una función asociada a la manipulación de archivos. Si en el código fuente no se manipulan archivos, el código no será propenso a contener este *Code Smell* en particular.

La categoría *Conventions*, es la segunda categoría con mayor cantidad de *Code Smells* detectados, con un 54,3 %. PIS_N siguió un estándar de codificación definido por el equipo, donde se observan reglas sobre cómo y dónde definir comentarios, convenciones de nombres para variables y constantes, indentación, uso de excepciones y ciclos (*while, for*). Creemos que el estándar definido es acotado en cuanto a alcance y cantidad de reglas en comparación con el es-

tándar oficial proporcionado por la documentación de la tecnología *Java* [31]. Adicionalmente, se desconoce el criterio utilizado para la selección de las reglas definidas en el estándar. Creemos que este punto podría ser una de las razones por las cuales se observa una alta tasa de ocurrencias dentro de esta categoría. A su vez, por más que se haya definido el estándar para ser utilizado por el equipo al momento de implementar, las revisiones de código por parejas muestran que se incurrió en *Code Smells* que fueron reportados por el equipo PIS_N. Verificamos que efectivamente algunos de los defectos de diseño reportados fueron corregidos por el equipo mientras que otros no se corrigieron o se volvió a incurrir en ellos. Esto denota algún tipo de dificultad en la utilización o apego al estándar definido.

El porcentaje de *Code Smells* detectados para la categoría *OOAbusers* frente al total de *Code Smells* es de 17,6 %. La categoría *OOAbusers* contiene *Code Smells* relacionados a la violación de los principios de orientación a objetos. Dado que PIS es una asignatura del octavo semestre de la carrera, destacamos que los estudiantes que cursan PIS han estudiado y aplicado conceptos relacionados a principios de orientación a objetos en asignaturas obligatorias de semestres anteriores. Creemos que a raíz de este punto es posible justificar el bajo porcentaje de detecciones para la categoría en cuestión, considerando que la mayoría de los principios de orientación a objetos aplican al proyecto PIS_N.

La cantidad de *Code Smells* detectados para la categoría *Performance* representa el 2,1 % del total de *Code Smells* detectados. A su vez, de los 23 *Code Smells* que se incluyen en el *Quality Profile* para esta categoría, únicamente se detectan dos, lo que representa el 8,7 %. Este resultado puede deberse a que el cliente del proyecto PIS_N hizo énfasis en la necesidad de que el producto a construir tenga un buen nivel de *performance*, requisito que entendemos se cumplió luego de analizar la documentación del producto final y los resultados de las pruebas de aceptación.

Por último, mencionar que el único *Code Smell* de la categoría *Security*, *Standard outputs should not be used directly to log anything*, fue detectado en PIS_N. Este *Code Smell* detecta el registro de mensajes sobre la salida estándar.

Principales ocurrencias de *Code Smells* en el proyecto PIS_N

La tabla 5.2 contiene los diez *Code Smells* con mayor cantidad de ocurrencias.

Code Smell	**Cantidad de ocurrencias**
String literals should not be duplicated	126
Magic numbers should not be used	91
Exception handlers should preserve the original exceptions	88
Static non-final field names should comply with a naming convention	88
"public static"fields should be constant	83
Class variable fields should not have public accessibility	83

Sigue en la página siguiente.

Code Smell	**Cantidad de ocurrencias**
Exceptions should not be thrown in finally blocks	67
Private fields only used as local variables in methods should become local variables	56
Lines should not be too long	52
Generic exceptions should never be thrown	45

Tabla 5.2: *Code Smells* con mayor cantidad de ocurrencias en el proyecto PIS_N

A continuación se analizan los primeros cinco *Code Smells* de la tabla 5.2 por razones de tiempo y alcance del proyecto de grado. Para este análisis se tienen en cuenta los siguientes puntos:

- Descripción del *Code Smell*.
- Categorización del *Code Smell* según la taxonomía extendida.
- Ejemplos de ocurrencias del *Code Smell* en PIS_N.
- Sugerencias para la resolución del *Code Smell*.
- Reglas del estándar de codificación definido en PIS_N relacionados al *Code Smell*.
- Errores reportados en la revisión por parejas de PIS_N relacionados al *Code Smell*.
- Observaciones.

1- ***String literals should not be duplicated***

- Descripción: las cadenas de texto no deben estar duplicadas. Se recomienda utilizar constantes para estos casos. El uso de constantes permite definir el valor de la misma una única vez, facilitando el mantenimiento y mejorando la legibilidad del código.
- Categoría: *Change Preventers, Dispensables*.
- Ejemplos de ocurrencias: las ocurrencias se encuentran distribuidas en varias clases del proyecto. Existen dos tipos de ocurrencias que se repiten sistemáticamente. Por un lado, existen ocurrencias asociadas a la utilización de *Strings* para obtener un valor de un *HashMap*. Un ejemplo de este caso se encuentra en la clase *CreateKezmoFlowItemActivity.java*:

```
if (parsedParams.get("createdByName") != null) {}
```

El otro caso está asociado a clases que contienen pruebas unitarias. En estas clases se definen los parámetros de entrada como *Strings*. Un ejemplo de este caso se encuentra en la clase *ExecuteResultSetQueryActivityTest.java*:

```
params1.put("title", "Actividad query sql
de prueba, tests unitarios");
```

- Sugerencias de resolución: creación de constantes y sustitución de todas las ocurrencias de estos *Strings* en el código fuente. Los entornos de desarrollo brindan la posibilidad de realizar una refactorización sustituyendo todas las ocurrencias de un *String* de forma automática. Cuando existe una gran cantidad de constantes, puede ser apropiado definir clases de constantes por conceptos.

- Reglas del documento de estándar de *Java*: existe una regla definida en el documento de estándar asociada a este *Code Smell*: "No se permiten valores de tipo *String hardcoded* (escritos a fuego en el código)" [11].

- Errores reportados: el equipo del proyecto PIS_N reportó los siguientes 2 errores que de haberse corregido podrían haber solucionado ocurrencias de este *Code Smell*:

  - El primero de los errores refiere a que *"Los mensajes de las excepciones deben declararse como constantes"* para mejorar la legibilidad y mantenibilidad del código. Se observa que en algunas clases de PIS_N los mensajes no están declarados como tal y han quedado duplicados.
  - El segundo error hace referencia a que *"No se agregan constantes"* en algunas clases de PIS_N. Presentan cadenas de texto duplicadas.

- Observaciones: existe una regla específica en el documento de estándar definido asociada directamente al *Code Smell* detectado. Creemos que si se hubiera seguido correctamente el estándar durante la implementación, no existirían (o se minimizarían) las detecciones de este *Code Smell*.

2- ***Magic numbers should not be used***

- Descripción: se debe evitar el uso de "números mágicos" directamente en el código. Los números mágicos son aquellos que aparecen en medio del código y no se sabe muy bien por qué o de dónde vienen exactamente, son valores constantes que no están declarados como tales. La sustitución de estos números por constantes hace que el código sea más legible y claro.

- Categoría: *Change Preventers*.

- Ejemplos de ocurrencias: el *Code Smell* se encuentra distribuido en varias clases del proyecto, por ejemplo en la clase *Persistence.java* con la siguiente ocurrencia:

```
String idact1 = String.valueOf(122222 + iter);
```

En este caso se hace uso de un número fijo para generar un identificador. Este número debería estar definido como una constante y el nombre de la constante debería ser descriptivo.

- Sugerencia de resolución: sustituir el número por una constante.
- Reglas del documento de estándar de *Java*: No existen reglas específicas asociadas a este *Code Smell*.
- Errores reportados: no existen errores reportados por el equipo PIS_N asociados a este *Code Smell*.
- Observaciones: existe una similitud entre el primer *Code Smell* analizado (*String literals should not be duplicated*) y el actual. Los dos *Code Smells* refieren al uso de constantes como forma de evitar el *Code Smell*. Resulta llamativo que los dos *Code Smells* más incurridos del proyecto PIS_N tengan esta similitud. Desde nuestro punto de vista, sería bueno hacer énfasis en la enseñanza de los estudiantes (ya sea en el PIS o previamente en la carrera) sobre los beneficios que tiene el uso de constantes en la productividad, mantenibilidad, claridad y legibilidad del código.

3- ***Exception handlers should preserve the original exceptions***

- Descripción: cuando se captura una excepción dentro de un bloque *catch* se debe realizar el registro de la excepción capturada y hacer el *throw* correspondiente para lanzar explícitamente la excepción.
- Categoría: *Exceptions*.
- Ejemplos de ocurrencias: una de las clases que presenta ocurrencias del *Code Smell* es la clase *Controller.java*:

```
    try {
 result = Persistence.getSequentialWorkflow(wfid);
} catch (DBAException e) {
 e.printStackTrace();
}
```

En este caso se puede observar que se está realizando el registro de la excepción. Creemos que quizá sería más adecuado usar algún *framework* (como el *logging* de *Java*) para realizar el registro de la excepción y así aprovechar las ventajas que este provee. Por otra parte, no se está realizando el *throw* correspondiente.

Otra clase que contiene ocurrencias de este *Code Smell* es la clase *DoWhileBlockActivity.java*:

```
try {
  dtSequentialWorkflow =
  Persistence.getSequentialWorkflow(wfid);
} catch (DBAException e) {
  throw new KezmoWorkflowException(ERROR_DBAEXEPTION
  + e.getMessage());
}
```

En este caso, se está creando una nueva excepción *custom*, pero se está utilizando un constructor que no recibe la excepción original como parámetro. Como consecuencia, la excepción original se pierde.

- Sugerencia de resolución: en el primer caso mencionado, una posible solución sería realizar el registro de la excepción con un *framework* de registro y luego hacer el *throw* correspondiente. Para el segundo caso, luego de realizar el registro de la excepción se debería utilizar el constructor de la clase *KezmoWorkflowException.java* que recibe la excepción original por parámetro.
- Reglas del documento de estándar de *Java*: no existen reglas asociadas al *Code Smell*.
- Errores reportados: el siguiente error es reportado en el informe de revisión de PIS_N.
  - *"Al arrojar una nueva excepción, agregar la excepción atrapada"*. Por la descripción de error se entiende que existe una relación con el *Code Smell* detectado. En el error reportado en la revisión por parejas se observa que se están creando nuevas excepciones sin hacer referencia a la excepción original.
- Observaciones: resulta llamativo la cantidad de ocurrencias del *Code Smell* dadas las posibles consecuencias negativas que trae el incurrir en el mismo. Existen algunos casos, muy puntuales, en que puede ser necesario dar tratamiento a la excepción dentro del *catch*. Si no se lanza la excepción capturada, la misma se perderá. Este último punto, sumado a que la excepción no fue registrada, podría resultar en que el error pase desapercibido. Esto podría repercutir en serios problemas de inconsistencias, que a su vez, serán muy difíciles de entender e identificar.

*4-**Static non-final field names should comply with a naming convention***

- Descripción: los nombres de los atributos definidos como estáticos y no finales deben corresponder con el estándar definido para este tipo de atributos.
- Categoría: *Conventions*.
- Ejemplos de ocurrencias: una de las clases que presenta ocurrencias del *Code Smell* es la clase *EventListener.java*:

```
private static String HOST =
System.getenv("REDIS_HOST");
```

La expresión regular definida para este tipo de atributos no permite nombres en mayúscula. Sin embargo, vale la pena preguntarse si estos atributos deberían ser finales. En dicho caso, se estaría respetando el estándar definido para atributos de ese tipo.

- Sugerencia de resolución: la solución a problemas de convenciones de nombres suele ser directa, renombrar de acuerdo a la convención definida. Se busca que el nombre a utilizar cumpla con la expresión regular definida para ese tipo de atributo.
- Reglas del documento de estándar de *Java*: la regla a continuación se encuentra relacionada al *Code Smell* detectado. "No se permiten valores de tipo *String hardcoded*. Los mismos deberán ser declarados al comienzo de cada clase. Deberán poseer el modificador *static* y el nombre deberá escribirse en mayúsculas. En caso de que esté compuesto de varias palabras, las mismas deberán separarse por un guión bajo" [11].
  Sin embargo, *SonarQube* define una expresión regular para detectar al *Code Smell Static non-final field names should comply with a naming convention*:

  ```
  ^[a-z][a-zA-Z0-9]*$.
  ```

  Como consecuencia, y en oposición a la regla definida por el equipo PIS_N, *SonarQube* no admite nombres completamente en mayúsculas o que incluyan guión bajo.
- Errores reportados: No existen errores reportados asociados a este *Code Smell*.
- Observaciones: la regla definida por PIS_N respecto al nombre de atributos estáticos no finales, y la regla definida por *SonarQube* para este *Code Smell*, no son coherentes entre sí. Creemos que esta incoherencia puede haber impactado en la cantidad de ocurrencias detectadas del *Code Smell*.

5- ***"public static" fields should be constant***

- Descripción: un atributo definido como público y estático debe estar definido como final.
- Categoría: *Java technology*.
- Ejemplos de ocurrencias: la clase *EntitiesQueries.java* presenta el total de las ocurrencias del *Code Smell* mencionado. El propósito de esta clase en particular es proporcionar los atributos que hacen referencia a consultas de la base de datos. A continuación, a modo de ejemplo se detalla una línea de código de la clase *EntitiesQueries.java* que contiene ocurrencias del *Code Smell* en cuestión.

```
public static String INSERT_ACTIVITY_QUERY =
"INSERT INTO kw_activity values (?, ?, ?)";
```

- Sugerencia de resolución: definir como final cada una de las ocurrencias detectadas del *Code Smell* en la clase. El total de ocurrencias presentadas en esta clase es 83. Teniendo en cuenta que cualquier entorno de desarrollo brinda la posibilidad de sustituir una cadena de texto por otra, podemos pensar que la refactorización es simple y rápida.
- Reglas del documento de estándar de *Java*: creemos que la regla definida en el estándar del equipo PIS_N que mencionamos en el *Code Smell* anterior y que se encuentra asociada a *Strings hardcoded*, también tiene relación con este *Code Smell*. Se destaca nuevamente la incoherencia entre la regla definida en el estándar de PIS_N y la regla definida por *SonarQube* donde se indica que este tipo de atributos debe estar definido como final.
- Errores reportados en revisión por parejas: No existen errores reportados asociados a este *Code Smell*.
- Observaciones: nuevamente se puede observar el posible impacto negativo en la calidad del código como consecuencia de la diferencia entre el estándar definido y usado por el equipo PIS_N y la regla definida por *SonarQube* para el *Code Smell*.

Observaciones generales de los Code Smells con mayor cantidad de ocurrencias

A partir del análisis realizado, vemos que cuatro de los cinco *Code Smells* más incurridos están vinculados con reglas del estándar de codificación *Java* definido por el equipo PIS_N y con los reportes que realizaron. En las revisiones por parejas realizadas, se reportan algunas pocas ocurrencias de estos *Code Smells*, lo cual difiere en cuanto a la cantidad de ocurrencias detectadas por las herramientas automatizadas utilizadas en nuestro proyecto. Asimismo, se observan diferencias entre las reglas usadas en el estándar de codificación del equipo PIS_N y las reglas definidas por *SonarQube*, lo cual posiblemente haya repercutido en las ocurrencias de los *Code Smells*. En base al estudio realizado y nuestra propia experiencia como ex alumnos de la asignatura PIS, consideramos que no ha sido bien definido el estándar para PIS_N, la cantidad de reglas que se incluyen son pocas en comparación a las definidas por *SonarQube* para la tecnología *Java* además de que creemos que no se ha ejecutado de la mejor forma.

Ocurrencias de *Code Smells* en el proyecto PIS_N categorizados según la taxonomía extendida

En la tabla 5.3 se detalla para cada categoría de la taxonomía extendida, figura 4.2, la cantidad de ocurrencias de *Code Smells* detectados en PIS_N y el porcentaje de ocurrencias sobre el total de las ocurrencias detectadas.

Categoría	**Ocurrencias de *Code Smells***	**Ocurrencias de *Code Smells* sobre el total de ocurrencias de *Code Smells* detectados**
Dispensables	298	21,1 %
Exceptions	245	17,4 %
Java Technology	227	16,1 %
Conventions	225	16 %
Change Preventers	224	15,9 %
Encapsulators	141	10 %
Clarity and Readability	102	7,2 %
OOAbusers	86	6,1 %
Bloaters	59	4,2 %
Good Coding Practices	57	4 %
Security	18	1,3 %
Performance	5	0,4 %
Couplers	2	0,1 %

Tabla 5.3: Ocurrencias de *Code Smells* por categoría de la taxonomía extendida.

Las categorías *Dispensables, Exceptions, Java Technology, Conventions y Change Preventers* agrupan aproximadamente el 86,5 % de las ocurrencias en PIS_N. A modo de profundizar el análisis sobre las categorías mencionadas, se detallan los *Code Smells* con más ocurrencias dentro de cada una de ellas. En particular, se observa la cantidad de ocurrencias y la cantidad de clases en las que se detecta cada *Code Smell*.

En la tabla 5.4 se detallan los *Code Smells* detectados en PIS_N para la categoría *Change Preventers*, la cantidad de ocurrencias y el número de clases en los que se detectan.

Change Preventers		
Code Smell	**Cantidad de ocurrencias**	**Cantidad de clases en las que se detecta**
String literals should not be duplicated	126	24
Magic numbers should not be used	91	26
Raw types should not be used	5	3
Local variables should not shadow class fields	2	2

Tabla 5.4: Ocurrencias de *Code Smells* en la categoría *Change Preventers*.

Como ya se mencionó anteriormente, llama la atención la similitud de los dos primeros *Code Smells*, donde ambos implican un tipo de defecto de diseño muy similar relacionado a la creación de constantes de clase. Entre estos dos *Code Smells* se distribuyen el 96,62 % de las ocurrencias dentro de la categoría.

Se han detectado en PIS_N 4 de los 6 *Code Smells* estudiados de la categoría *Change Preventers*.

La tabla 5.5 detalla los *Code Smells* detectados en PIS_N para la categoría *Conventions*, su cantidad de ocurrencias y la cantidad de clases en las que se detectan. Posteriormente se profundiza sobre los dos *Code Smells* con mayor cantidad de ocurrencias en esta categoría. Entre estos dos primeros *Code Smells* se distribuyen el 57 % del total de ocurrencias para la categoría.

Conventions		
Code Smell	**Cantidad de ocurrencias**	**Cantidad de clases en las que se detecta**
Static non-final field names should comply with a naming convention	88	4
Statements should be on separate lines	40	12
Modifiers should be declared in the correct order	17	17
Loggers should be "private static final.ªnd should share a naming convention	16	16
Packages should have a javadoc file 'package-info.java'	14	0 *
The members of an interface or class declaration should appear in a predefined order	12	5

Tabla 5.5: Ocurrencias de *Code Smells* en la categoría *Conventions*.
* El *Code Smell* se encuentra asociado a paquetes y no a clases.

El *Code Smell Static non-final field names should comply with a naming convention* tiene 83 ocurrencias en la clase *EntitiesQueries.java* de un total de 88 ocurrencias en el proyecto PIS_N. Si bien este *Code Smell* es el que tiene mayor cantidad de ocurrencias dentro de la categoría *Conventions*, las ocurrencias se presentan mayoritariamente en una clase.

Para el *Code Smell Statements should be on separate lines* se observa que 37 de las 40 ocurrencias en PIS_N son detectadas sobre una porción de código que se repite en diez clases diferentes:

```
if (statement != null) try { statement.close(); }
catch (SQLException sql) { throw new DBAException(sql); }
```

En el código anterior se observa que hay múltiples sentencias *Java* en una misma línea. Creemos que este defecto de diseño se ha propagado en PIS_N a causa de la acción "copiar y pegar". En este sentido, se destaca que el *Code Smell Statements should be on separate lines* fue reportado en la revisión por parejas

realizada por el equipo PIS_N y no se corrigió. Para solucionar este defecto de diseño, se deben separar en distintas líneas las sentencias del código.

La tabla 5.6 detalla los *Code Smells* con mayor cantidad de ocurrencias en la categoría *Java technology*. Los primeros 3 *Code Smells* de la tabla abarcan el 58 % del total de ocurrencias para la categoría y pertenecen únicamente a esta categoría.

Java technology		
Code Smell	**Cantidad de ocurrencias**	**Cantidad de clases en las que se detecta**
"public static" fields should be constant	83	1
Mutable members should not be stored or returned directly	29	11
Utility classes should not have public constructors	20	20
"catch" clauses should do more than rethrow	16	2
The diamond operator ("<>") should be used	16	8

Tabla 5.6: Ocurrencias de *Code Smells* en la categoría *Java technology*.

La tabla 5.7 contiene los *Code Smells* con mayor cantidad de ocurrencias en la categoría *Exceptions*. Los primeros 3 *Code Smells* de la tabla acumulan el 81 % del total de las ocurrencias en la categoría. Estos *Code Smells* están relacionados sólo a la categoría *Exceptions*.

Exceptions		
Code Smell	**Cantidad de ocurrencias**	**Cantidad de clases en las que se detecta**
Exception handlers should preserve the original exceptions	88	27
Exceptions should not be thrown in finally blocks	67	13
Generic exceptions should never be thrown	45	27
catch clauses should do more than rethrow	16	2
Try-catch blocks should not be nested	13	5

Tabla 5.7: Ocurrencias de *Code Smells* en la categoría *Exceptions*.

Al analizar la distribución de las ocurrencias de los *Code Smells* de la tabla 5.7 en el código fuente del proyecto, se observa que el 26 % del total de las

ocurrencias de *Code Smells* de la categoría *Exceptions* se encuentran en las clases *Persistence.java* y *WorkflowResource.java*. Las ocurrencias detectadas en estas clases son sobre líneas de código muy similares entre sí, lo cual nos lleva nuevamente a la hipótesis de la propagación de *Code Smells* a causa de la acción "copiar y pegar". Se destaca que las dos clases mencionadas son parte del *core* del proyecto PIS_N, es decir, son clases que concentran la lógica de las funcionalidades principales del producto y/o cuya función es fundamental para la arquitectura del sistema.

La tabla 5.8 detalla los *Code Smells* con mayor cantidad de ocurrencias en la categoría *Dispensables*.

Dispensables		
Code Smell	**Cantidad de ocurrencias**	**Cantidad de clases en las que se detecta**
String literals should not be duplicated	126	24
Redundant casts should not be used	38	11
Source files should not have any duplicated blocks	32	32
Unused private fields should be removed	20	10
Unused method parameters should be removed	20	3

Tabla 5.8: Ocurrencias de *Code Smells* en la categoría *Dispensables*.

Resulta interesante mencionar que el *Code Smell String literals should not be duplicated* se lleva el 42 % de las ocurrencias de *Code Smells* en la categoría *Dispensables*. Este *Code Smell* además está relacionado a la categoría *Change Preventers*.

Se destaca que 90 % de las ocurrencias del *Code Smell Unused method parameters should be removed* se encuentran en la clase *EventListener.java*. Clase que forma parte del *core* del proyecto PIS_N.

Análisis comparativo entre las ocurrencias de *Code Smells* y sus detecciones por categoría

La figura 5.1 ilustra gráficamente la cantidad de ocurrencias (en color azul) de *Code Smells* y la cantidad de *Code Smells* detectados (en color naranja) en el proyecto PIS_N, clasificados en las categorías de la taxonomía extendida.

Cabe destacar que el proyecto PIS_N contiene una distribución despareja en las ocurrencias de los *Code Smells*. Es decir, pueden existir *Code Smells* que sean "responsables" de gran parte de la cantidad de ocurrencias dentro de una categoría. A modo de ejemplo, la categoría *OOAbusers* presenta tres *Code Smells* detectados. Sin embargo, del total de 86 ocurrencias de *Code Smells*, 83 de ellas pertenecen solo a uno de los tres *Code Smells* de la categoría. Como consecuencia, esta distribución peculiar nos permite inferir que existe una repetición sistemática del mismo *Code Smell* en el proyecto analizado.

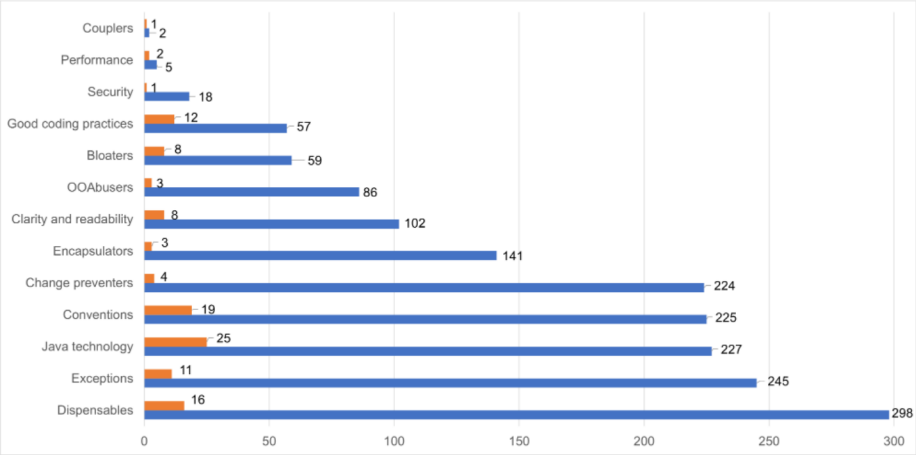

Figura 5.1: Comparación entre la cantidad de *Code Smells* detectados (naranja) y ocurrencias (azul) de los *Code Smells*

Observaciones

Las ocurrencias de *Code Smells* detectados en el proyecto PIS_N están relacionadas principalmente a las categorías *Java technology*, *Conventions*, *Dispensables*, *Good coding practices*, *Exceptions* y *Clarity and readability*.

Un estándar de implementación fue definido y utilizado por el equipo PIS_N en las revisiones de código por parejas [11]. Los reportes de estas revisiones refieren principalmente a *Code Smells* de las categorías antes mencionadas y que son detectables mediante las herramientas automatizadas seleccionadas. Creemos que se podrían enfocar las revisiones por parejas en aspectos del código fuente que hoy en día sean únicamente detectables por personas de forma manual. De esta manera se evitaría además la falta de coherencia entre el estándar utilizado y las reglas definidas por *SonarQube*.

5.1.2. RQ2: ¿Qué criticidad tienen los *Code Smells* en los que incurren los estudiantes?

En esta sección se analiza la criticidad de los *Code Smells* incurridos desde dos perspectivas. Por un lado, se estudia la gravedad de los mismos a través de los distintos niveles de gravedad establecidos por *SonarQube*. Por otro lado, se analizan los tiempos de refactorización de los *Code Smells* incurridos a través del tiempo de refactorización establecido por *SonarQube*. A partir de allí, surgen algunas observaciones sobre la precisión de estos tiempos definidos por *SonarQube*. Por último, se analizan los cinco *Code Smells* con mayor tiempo de refactorización y se establecen algunas observaciones.

En la tabla de *Code Smells* seleccionados para la investigación [3], se detalla la gravedad y el tiempo de refactorización establecido por *SonarQube* para cada *Code Smell*.

Análisis de la gravedad establecida por *SonarQube*

La documentación proporcionada por la herramienta *SonarQube* indica que cada vez que un fragmento de código infringe una regla del *Quality Profile*, se detecta un "problema". Estos problemas son categorizados por la herramienta según su gravedad. Existen cinco niveles de gravedad[1] en *SonarQube*.

- ***Blocker***, problema con una alta probabilidad de afectar el comportamiento de la aplicación en producción. Por ejemplo, pérdida de memoria, o conexión JDBC no cerrada. El código debe ser corregido de inmediato.
- ***Critical***, problema con baja probabilidad de afectar el comportamiento de la aplicación en producción o problema relacionado a defectos de seguridad. Ejemplos: un bloque *try/catch* con el *catch* vacío, inyección *SQL*, entre otros. El código debe ser revisado inmediatamente.
- ***Major***, problema que puede afectar en gran medida la productividad del desarrollador. Por ejemplo, código inalcanzable, bloques duplicados, parámetros no utilizados.
- ***Minor***, problema que puede afectar ligeramente la productividad del desarrollador. Por ejemplo, las líneas no deben ser demasiado largas, o las declaraciones de un *switch/case* deben tener al menos 3 casos.
- ***Info***, hallazgo en el código. No afecta la productividad del desarrollador ni el comportamiento de la aplicación. Suelen ser reglas cuyo objetivo es informar. Por ejemplo, *imports* no utilizados.

A partir de los resultados obtenidos de la ejecución de las herramientas seleccionadas, se observa que en el proyecto PIS_N la distribución de las ocurrencias de *Code Smells* según los niveles de gravedad de *SonarQube* es: 0 % *info*, 43,5 % *minor*, 38,7 % *major*, 17,7 % *critical*, 0,1 % *blocker*.

A simple vista se observa que el porcentaje de ocurrencias de *Code Smells* clasificados con gravedad *blocker* en PIS_N es escaso, en comparación con los porcentajes de las demás categorías de gravedad (a excepción de la categoría *Info*, en la cual no se presentaron ocurrencias). El único *Code Smell* detectado como *blocker* es *main should not throw anything*, con dos ocurrencias. Este *Code Smell* hace referencia al hecho de que no tiene sentido hacer un *throw* desde un método *main* ya que la excepción lanzada allí no sería capturada. Como consecuencia, no tendrá tratamiento y podría llegar a exhibirse al usuario final. Las ocurrencias se asocian a la clase que carga y ejecuta *tests* unitarios y a la clase *Persistence.java*. Este *Code Smell* fue reportado en las revisiones por parejas del equipo PIS_N, pero al parecer, no fue corregido.

Adicionalmente, se observa que el 82,2 % de las ocurrencias de *Code Smells* detectadas en PIS_N son de gravedad *major* y *minor*. En otras palabras (y de acuerdo a la definición de los niveles de *SonarQube*), existe un gran porcentaje de *Code Smells* incurridos que afectan la productividad de los programadores en mayor o menor medida. El 17,7 % restante, perteneciente a la categoría *critical*, podrían llegar a repercutir principalmente en el comportamiento de la aplicación.

[1] https://docs.SonarQube.org/latest/user-guide/issues/

En la tabla 5.9 se detalla la cantidad de ocurrencias de *Code Smells* detectados en PIS_N y los porcentajes asociados a los distintos niveles de gravedad para cada categoría de la taxonomía extendida. Allí se observa que las categorías *Bloaters* y *Change preventers* son las únicas categorías de la taxonomía extendida cuyos *Code Smells* superan el 50 % de sus ocurrencias en nivel *critical*, siendo el resto de las ocurrencias en dichas categorías en nivel *major*.

Categoría	**Ocurrencias de *Code Smells***	***Blocker* %**	***Critical* %**	***Major* %**	***Minor* %**
Dispensables	298	0	44,3	35,2	20,5
Exceptions	245	0,8	27,4	61,6	10,2
Java Technology	227	0,9	3,1	17,2	78,8
Conventions	225	0	0	19,1	80,9
Change Preventers	224	0	56,2	43,8	0
Encapsulators	141	0	0	1,4	98,6
Clarity and Readability	102	0	0	60,8	39,2
OOAbusers	86	0	1,2	2,3	96,5
Bloaters	59	0	52,5	47,5	0
Good Coding Practices	57	0	21,1	19,3	59,6
Security	18	0	0	100	0
Performance	5	0	0	60	40
Couplers	2	0	0	100	0

Tabla 5.9: Porcentaje de gravedad de los *Code Smells* detectados en PIS_N distribuidos por categoría de la taxonomía extendida.

En relación a la categoría *Bloaters*, los *Code Smells* asociados a nivel *critical* están relacionados principalmente a complejidad de métodos y flujos en el código. Por ejemplo, el *Code Smell Methods should not be too complex* y/o *Control flow statements ïf", "for", "while", "switch.ªnd "try"should not be nested too deeply*. La complejidad excesiva puede afectar el comportamiento del sistema, por ejemplo a nivel de *performance*. En cuanto a los *Code Smells* asociados a nivel *major*, refieren esencialmente a métodos o clases demasiado extensas, por ejemplo, *Methods should not have too many lines* y *Classes should not have too many fields*. Estos *Code Smells* impactan negativamente en la productividad de los desarrolladores, por ejemplo complejizando el entendimiento de la introducción de nuevos flujos dentro de estos métodos complejos.

La categoría *Change preventers* presenta un único *Code Smell* de nivel *critical*: *String literals should not be duplicated*. Desde nuestra perspectiva, no consideramos que este *Code Smell* esté asociado al nivel *critical* ya que no afectaría el comportamiento del sistema. Sin embargo, si podría afectar la productividad de los programadores. En cuanto a nivel *major*, más del 90 % de las ocurrencias se las lleva el *Code Smell Magic numbers should not be used*. La refactoriza-

ción de estos *Code Smells* mejora el entendimiento y la legibilidad del código, mejorando de esta manera la productividad de los desarrolladores.

Las categorías *Security* y *Couplers* tienen el total de sus ocurrencias en nivel de gravedad *major*. Cabe destacar que estas categorías poseen únicamente un *Code Smell* detectado *Standard outputs should not be used directly to log anything*. Este *Code Smell* refiere a que si la información del *log* no se encuentra formateada, puede tornarse difícil analizarla. Todo *log* debería ser fácilmente recuperable y analizable por un desarrollador.

Las categorías *Java technology* y *Conventions* poseen una distribución similar por nivel de gravedad, centrando casi el 80 % de sus ocurrencias en nivel *minor* y el resto en *major*. Se observa que la distribución de las ocurrencias según el nivel de gravedad es coherente dadas las definiciones de ambas categorías. Es decir, es lógico que los *Code Smells* asociados a *Java technology* y *Conventions* no impacten en el comportamiento del sistema y que en caso de afectar la productividad del equipo, lo hagan en un grado menor.

En términos generales, se observa que la categorización que establece *SonarQube* por nivel de gravedad es coherente considerando nuestra experiencia a nivel académico y laboral. Esta categorización puede ser un buen punto de partida para definir la estrategia de refactorización. Un primer objetivo podría ser no tener *Code Smells* de nivel *blocker* y *critical*. Como paso siguiente, se podrían considerar los *Code Smells* de nivel *major*. Teniendo en cuenta que este tipo de *Code Smells* afecta la productividad de los desarrolladores, sería razonable comenzar por corregir aquellos *Code Smells* que se presenten en las clases *core* del sistema.

Análisis del tiempo de refactorización establecido por *SonarQube*

La documentación de *SonarQube* indica que la herramienta asocia el concepto "deuda técnica" al tiempo estimado de refactorización que se necesita para corregir un problema. En el artículo *On the Accuracy of SonarQube Technical Debt Remediation Time* [29], se analiza la precisión de los tiempos de refactorización estimados por *SonarQube* para la corrección de *Bugs, Vulnerabilities y Code Smells* en un conjunto de 15 proyectos *Java* de código abierto. Los resultados de este estudio señalan que el tiempo de refactorización de *SonarQube*, en comparación con el tiempo real invertido por los desarrolladores en la corrección de los *Bugs, Vulnerabilities y Code Smells*, es generalmente sobre estimado. Sin embargo, el estudio concluye que al analizar la precisión de los tiempos únicamente para *Code Smells*, esta sobreestimación disminuye, es decir, los tiempos son más precisos. El estudio *On the Accuracy of SonarQube Technical Debt Remediation Time* [29] se centra en *Code Smells* de *SonarQube*, dejando por fuera los *Code Smells* asociados a los *plugins PMD, CheckStyle, Code Smells and AntiPatterns detection*.

A diferencia de los *plugins PMD y CheckStyle*, donde el tiempo de refactorización varía por *Code Smell*, el *plugin Code Smells and AntiPatterns detection* establece un tiempo de refactorización fijo de 1 hora y media por *Code Smell*.

En relación al *plugin Code Smells and AntiPatterns detection*, se observa que es complejo determinar manualmente los tiempos de refactorización asociados a los *Code Smells* y *AntiPatterns* del *plugin* ya que depende de un conjunto de

características asociadas a la regla en particular, sus ocurrencias en el código y el contexto donde se incurre. A modo de ejemplo, la refactorización del *AntiPattern Blob*, detallado en la sección 2.3, dependerá de qué tanto haya crecido la clase donde se incurre, qué tantas responsabilidades haya asumido, qué tan complejos sean sus métodos y qué tan acoplada esté respecto al resto del sistema. Por esta razón, se decide descartar del análisis los tiempos de refactorización del *plugin Code Smells and AntiPatterns detection*.

Por otra parte, se analizaron manualmente los *Code Smells* para los *plugins PMD y CheckStyle*, para los cuales se considera que los tiempos establecidos son coherentes, a excepción del *Code Smell NPath Complexity* de *CheckStyle*. Este *Code Smell* presenta un tiempo de refactorización fijo de 1 hora. Contrastando contra el *Code Smell Cognitive Complexity of methods should not be too high* de *SonarQube*, se observa que este último define un tiempo de refactorización dinámico de acuerdo a la complejidad del método. Desde nuestra perspectiva consideramos que el tiempo de refactorización del *Code Smell* asociado a *CheckStyle* también debería ser dinámico.

Como conclusión, considerando el estudio realizado sobre los tiempos de *SonarQube* y la revisión manual realizada sobre los *Code Smells* de los *plugins PMD y CheckStyle*, se decide tener en cuenta para el análisis los tiempos de refactorización de *SonarQube* y los *plugins PMD y CheckStyle*. Asimismo, se decide descartar del análisis los tiempos de refactorización brindados por el *plugin Code Smells and AntiPatterns detection* dado que son estáticos y resulta complejo establecer valores precisos para este tipo de *Code Smells* y *Antipatterns*.

Code Smells con mayor tiempo de refactorización

Por cuestiones de alcance y esfuerzo, a continuación se analizarán los cinco *Code Smells* con mayor tiempo de refactorización detectados en PIS_N. En esta sección el análisis estará enfocado en evaluar si los tiempos son acordes, si presentan variabilidad en casos particulares, etc. Para cada *Code Smell* se detallará:

- descripción del *Code Smell*.
- nivel de gravedad del *Code Smell*.
- tiempo de refactorización del *Code Smell*.
- observaciones.

1- ***Classes should not be coupled to too many other classes (Single Responsibility Principle)***

- Descripción: de acuerdo con el principio de responsabilidad única [23], una clase debe tener una sola responsabilidad ya que:
  - si una clase tiene más de una responsabilidad, las responsabilidades se acoplan.
  - los cambios en una responsabilidad pueden afectar o inhibir la capacidad de la clase para cumplir con sus demás responsabilidades.

  - este tipo de acoplamiento conduce a diseños frágiles que se rompen de manera inesperada al sufrir modificaciones.

- Nivel de gravedad: *major*.
- Tiempo de refactorización: 2 horas por ocurrencia.
- Observaciones: en la revisión por parejas se reportó que la clase contiene métodos de prueba que no son utilizados en el sistema con otra finalidad que no fuese la verificación del producto. En este sentido, deberían ser eliminados. Es importante mencionar que si se hubiera tomado acción luego del reporte realizado en la revisión por parejas de PIS_N, el acoplamiento de la clase hubiese disminuido. Por otra parte, este es un caso donde el tiempo de refactorización dependerá en gran parte de qué tan acoplada se encuentre la clase. Es probable que a mayor acoplamiento, mayor será el tiempo de refactorización.

2- ***NPath Complexity***

- Descripción: la métrica *NPath* calcula el número de posibles rutas de ejecución a través de una función. La regla comprueba la complejidad "*NPath*" de un método con un límite especificado (el valor predeterminado es 200). Para ello, se tiene en cuenta el anidamiento de declaraciones condicionales y expresiones *booleanas* de varias partes.
- Nivel de gravedad: *major*.
- Tiempo de refactorización: 1 hora por ocurrencia.
- Observaciones: analizando las ocurrencias del *Code Smell* se observan complejidades muy diferentes entre las ocurrencias, con valores que van desde 208 hasta 8030. Como se mencionó anteriormente, los tiempos de refactorización serán muy diferentes en estos casos.

3- ***Source files should not have any duplicated blocks***

- Descripción: se comprueba la existencia de al menos un bloque de código duplicado en una clase *Java*.
- Nivel de gravedad: *major*.
- Tiempo de refactorización: 10 minutos (fijo) más 10 minutos para cada bloque duplicado.
- Observaciones: se observan tiempos de refactorización que van desde 70 minutos, es decir, clases que presentan 6 bloques de código duplicado, hasta 20 minutos. Se destaca que el *Code Smell* detecta bloques de código duplicado únicamente en un mismo archivo *Java*. Sería interesante analizar en estas ocurrencias, si el código está duplicado en otras clases *Java*.

4- ***Classes should not have too many fields***

- Descripción: una clase que crece demasiado tiende a acumular demasiadas responsabilidades e inevitablemente se vuelve más difícil de entender y mantener. Tener muchos campos es un indicio de que una clase ha crecido demasiado. Por encima de un umbral (por defecto 20 campos), se recomienda refactorizar la clase.
- Nivel de gravedad: *major*.
- Tiempo de refactorización: 1 hora por ocurrencia.
- Observaciones: existe una única ocurrencia del *Code Smell* en la clase *EntitiesQueries.java*. Los atributos de esta clase deberían estar definidos como constantes. Impactar esta solución no llevaría más de 10 minutos. Este es un ejemplo claro donde se puede observar cómo el tiempo de refactorización puede variar en gran medida para cada caso en particular.

5- ***Exceptions should not be thrown in finally blocks***

- Descripción: lanzar una excepción desde dentro de un bloque *finally* enmascara cualquier excepción que se haya lanzado previamente en el bloque *try/catch*. Como consecuencia, se pierde el mensaje de excepción enmascarado y el seguimiento del *stack*.
- Nivel de gravedad: *critical*.
- Tiempo de refactorización: 30 minutos por ocurrencia.
- Observaciones: El *Code Smell* presenta 67 ocurrencias. En su mayoría están asociadas a la necesidad de cerrar recursos luego de ser utilizados. Una posible solución a este problema es el uso de la instrucción "*try with resources*" que se introdujo en *Java 7*. Mediante *try with resources* se permite declarar los recursos que se utilizaran en un bloque *try* con la seguridad de que los recursos se cerrarán después de la ejecución del bloque, evitando el uso de *finally*. Aplicando esta solución, es probable que los tiempos de refactorización sean menores a los establecidos por *SonarQube*.

A continuación se analizan y discuten algunas de las observaciones más relevantes sobre los tiempos de refactorización.

- El tiempo de refactorización de un *Code Smell* no siempre debería ser el mismo para todas sus ocurrencias, ya que la solución de cada ocurrencia dependerá del contexto donde se incurra. Las herramientas seleccionadas para esta investigación no tienen la capacidad de contextualizar las ocurrencias de los *Code Smells* detectados. Por lo tanto, las soluciones propuestas por la herramienta pueden no ser las adecuadas. Como por ejemplo, el caso del *Code Smell* ya presentado *Exceptions should not be thrown in finally blocks*, donde la solución propuesta por *SonarQube* es eliminar el *throw* que se realiza en el bloque *finally*. Sin embargo, se puede observar que muchas ocurrencias de este *Code Smell* están asociadas a recursos que necesitan ser liberados en el bloque *finally*. En estos casos, el uso de *try with resources* solucionaría este problema. Por lo tanto, comprender el contexto en el cual se detecta la ocurrencia del *Code Smell* es fundamental para poder aplicar una solución adecuada.

- Existen situaciones en las que se detectan *Code Smells* sobre la misma línea de código, en estos casos es probable que corrigiendo uno de los *Code Smells* se solucionen los otros *Code Smells* asociados a la misma línea. En este sentido, existe la posibilidad de que los tiempos de refactorización totales del proyecto sean menores que los estimados por *SonarQube*.
- La mayoría de los entornos de desarrollo ofrecen facilidades para realizar ciertas refactorizaciones en el código, algunas a destacar son: formateo automático de la clase, reemplazar textos de forma masiva, eliminar *imports* innecesarios, entre otras. *SonarQube* no tiene en cuenta los "atajos" que ofrecen los entornos de desarrollo al estimar el tiempo de refactorización de un *Code Smell*. Es probable que estas facilidades disminuyan los tiempos de refactorización frente a los establecidos por *SonarQube*.
- Existe una diversidad considerable en la complejidad de los *Code Smells* detectados. Es probable que la refactorización de los *Code Smells* con alta complejidad tengan un alto impacto en la estabilidad del sistema y por lo tanto se necesite una planificación de refactorización adecuada. Por ejemplo, realizar una refactorización del *Code Smell NPath Complexity* asociado a complejidad excesiva en métodos, posiblemente implique analizar cómo disminuir esa complejidad. Es decir, cómo desagregar ese método en métodos más simples, cómo reagrupar condiciones "*if-else*". Este tipo de cambios pueden introducir nuevos errores en el sistema. En este sentido, será necesario evaluar el momento adecuado para realizarlo, buscando minimizar el riesgo de introducir nuevos errores. Adicionalmente, la asignación de los *Code Smells* a refactorizar deberá ser adecuada según la experiencia del desarrollador. A modo de ejemplo, los *Code Smells* asociados a convenciones y tecnología *Java* podrían ser corregidos por desarrolladores con poca experiencia. Sin embargo, *Code Smells* relacionados a complejidad excesiva o acoplamiento deberían ser asignados a desarrolladores con cierta experiencia.

5.1.3. RQ3: ¿Cómo se distribuyen los *Code Smells* detectados en el código del proyecto?

Como se menciona en la sección 3.1, la arquitectura del proyecto PIS_N se distribuye en cinco capas. Para el análisis a continuación se considerarán únicamente las capas codificadas en el lenguaje *Java*. Como consecuencia, la capa de presentación implementada en *ReactJS*[2] queda por fuera del análisis. Por otra parte, también se excluirán del análisis las clases que refieren al *testing* unitario, con el objetivo de poner foco en las clases *Java* que tengan repercusión en cuanto al desarrollo del producto.

Distribución de *Code Smells* según las capas del proyecto PIS_N

En la tabla 5.10 se presentan las capas del proyecto donde para cada capa se incluye la cantidad de clases, la cantidad de líneas de código, la cantidad de

[2] https://es.reactjs.org/

ocurrencias de *Code Smells* por capa y la densidad de *Code Smells* por capa.

Capa	**Cantidad de clases**	**LOC**	**Ocurrencias de *Code Smells***	**Densidad de *Code Smells***
Acceso a datos	19	2419	573	0,24
Negocio	60	2852	365	0,13
Transversal	19	580	65	0,11
Servicios	7	615	38	0,06

Tabla 5.10: Distribución de *Code Smells* en las distintas capas del proyecto PIS_N.

Para cada capa mencionada en la tabla 5.10 se presentan las diez clases con mayor cantidad de ocurrencias de *Code Smells*, a excepción de la capa de servicios donde se presentan sus siete clases. La información correspondiente a cada capa a estudiar se muestra en las siguientes tablas: 5.11, 5.12, 5.13, 5.14, donde cada tabla contiene las columnas: clase, cantidad de líneas de código (LOC), cantidad de ocurrencias de *Code Smells*, densidad de *Code Smells*, cantidad de *Code Smells* detectados, porcentaje de ocurrencias *Code Smells* de la clase sobre el total de ocurrencias de *Code Smells* de la capa.

La tabla 5.11 muestra la distribución de ocurrencias de *Code Smells* por clase en la capa de acceso a datos.

Capa de acceso a datos					
Clase *	**LOC**	**Ocurrencias ****	**Densidad ****	**Detecciones ****	**% Ocurrencias de la clase sobre ocurrencias de la capa ****
EntitiesQueries	107	269	2,51	7	46,95 %
Persistence	736	87	0,12	15	15,16 %
SequentialWorkflowDBA	127	21	0,17	8	3,66 %
StateMachineWorkflowDBA	121	19	0,16	7	3,31 %
WorkflowVersionDBA	123	18	0,15	4	3,14 %
DBManager	64	17	0,27	7	2,96 %
EventTypeDBA	108	15	0,14	7	2,61 %
ActivityDBA	78	12	0,15	7	2,09 %
StateDBA	85	12	0,14	5	2,09 %
RelationWorkflowEvtDBA	52	11	0,21	7	1,92 %

Tabla 5.11: Clases con mayor cantidad de ocurrencias de *Code Smells* en la capa de acceso a datos.
* La columna sintetiza el nombre de la clase como "x", cuyo nombre en el proyecto PIS_N sigue la nomenclatura *"x.java"*.
** de *Code Smells*.

En la tabla 5.11 se observa que la distribución de las ocurrencias de *Code Smells* de la capa de acceso de datos se centraliza en las clases *EntitiesQueries.java* y *Persistence.java*. Estas clases contienen el 62,11 % del total de las ocurrencias de la capa de acceso a datos. Cabe destacar que estas clases serán

analizadas en profundidad posteriormente en esta RQ, al estudiar la distribución de *Code Smells* por clase.

La clase *EntitiesQueries.java* presenta la mayor densidad de *Code Smells* del proyecto PIS_N. La segunda clase con mayor densidad es *DBManager.java*. Esta clase es la encargada de manejar las conexiones a la base de datos. Los siete *Code Smells* detectados se asocian a convenciones, buen uso de la tecnología y manejo de excepciones. El tiempo de refactorización de estos *Code Smells* no supera los diez minutos por ocurrencia.

Por último, el resto de las clases de la tabla 5.11 tienen la responsabilidad de acceder y persistir a las tablas de la base de datos. Cada una de estas clases hace referencia a una tabla de la base de datos, siguiendo la nomenclatura *"xDBA.java"* donde "x" va tomando los nombres de las distintas tablas de la base de datos. Este conjunto de clases presentan las mismas responsabilidades y tienen una estructura similar. Como consecuencia, las ocurrencias de los *Code Smells* detectados son muy similares, en múltiples casos idénticas. Como ejemplo se vuelve a mencionar el *Code Smell Statements should be on separate lines* estudiado en RQ1 (sección 5.1.1). Este *Code Smell* presenta un gran porcentaje de sus ocurrencias idénticas a causa de que la misma línea de código se propaga a través de las diferentes clases. En este sentido, creemos que la refactorización de los *Code Smells* incurridos en estas clases podría optimizarse haciendo uso de las ventajas que brindan los entornos de desarrollo. Por ejemplo, podría realizarse un reemplazo masivo de la línea que presenta la ocurrencia del *Code Smell* por su solución. De esta forma, se reemplazarán todas las ocurrencias de esa línea exacta por la solución propuesta. Aplicando estas técnicas, los tiempos de refactorización pueden llegar a ser menores que los estimados por *SonarQube*.

La tabla 5.12 muestra la distribución de ocurrencias de *Code Smells* por clase de la capa de negocio. Esta distribución es más homogénea en comparación con la capa de acceso a datos. La clase *EventListener.java* presenta el 15,07 % de las ocurrencias totales de la capa de negocio. Analizando las categorías de los *Code Smells* detectados de la capa de negocio se observa que aproximadamente el 75 % de las ocurrencias de *Code Smells* pertenecientes a la categoría *Bloaters* se presentan en esta capa. Dado que la complejidad del desarrollo del producto se centraliza en la capa de negocio, resulta esperable este porcentaje.

Capa de negocio					
Clase *	**LOC**	**Ocurrencias ****	**Densidad ****	**Detecciones ****	**% Ocurrencias de la clase sobre ocurrencias de la capa ****
EventListener	333	55	0,17	14	15,07 %
ForBlockActivity	91	25	0,27	14	6,85 %
Controller	226	24	0,11	14	6,58 %
CreateKezmoFlowItem	113	21	0,19	12	5,75 %
ForEachBlockActivity	81	17	0,21	12	4,66 %
CreateKezmoSpaceActivity	98	16	0,16	8	4,38 %
CreateKezmoContainerItem	107	15	0,14	9	4,11 %

Sigue en la página siguiente.

Clase *	**LOC**	**Ocurrencias ****	**Densidad ****	**Detecciones ****	**% Ocurrencias de la clase sobre ocurrencias de la capa ****
Activity	103	14	0,14	9	3,84 %
ExecuteResultSetQuery	71	14	0,2	8	3,84 %
CreateKezmoContainer	104	14	0,13	8	3,84 %

Tabla 5.12: Clases con mayor cantidad de ocurrencias de *Code Smells* en la capa de negocio.
* La columna sintetiza el nombre de la clase como "x", cuyo nombre en el proyecto PIS_N sigue la nomenclatura *"x.java"*.
** de *Code Smells*.

La clase *Controller.java* es el punto de entrada a la capa de negocio, es decir, la capa de servicios se comunica con la capa de negocio a través de esta clase. Por esta razón, es una de las clases *core* del proyecto. A su vez, esta clase se comunica con *EventListener.java* y *Persistence.java*, dos clases fundamentales en el desarrollo del producto. Esta clase presenta 14 *Code Smells*, donde para cada uno no se presentan más de cuatro ocurrencias. En la clase *Controller.java* se detecta el *Code Smell Classes should not be coupled to too many other classes (Single Responsibility Principle)*. Como se mencionó en RQ2 (sección 5.1.2), este *Code Smell* es el que presenta mayor esfuerzo de refactorización de los detectados en PIS_N. Dadas las características de la clase *Controller.java* y las ocurrencias de *Code Smells* que presenta, creemos sería conveniente distribuir las responsabilidades de la clase en clases más pequeñas. De lo contrario, es probable que la clase comience a asumir demasiadas responsabilidades, complejizando el mantenimiento de la misma en el corto plazo.

El resto de las clases que se presentan en la tabla 5.12 extienden de la clase *Activity.java*, presentando una estructura similar entre ellas. Como se mencionó en RQ2 (sección 5.1.2), la clase abstracta *Activity.java* es una de las clases *core* del proyecto PIS_N de la cual heredan todas las clases que siguen la nomenclatura *"xActivity.java"* donde "x" va tomando los nombres de las actividades del *workflow*. La clase presenta nueve *Code Smells* diferentes, donde ninguno de los *Code Smells* supera las tres ocurrencias. Con un razonamiento análogo al realizado recientemente para la capa de acceso de datos, llegamos a la conclusión de que existen ocurrencias de *Code Smells* muy similares entre estas clases.

A continuación, la tabla 5.13 muestra la distribución de ocurrencias de *Code Smells* por clase de la capa transversal. Las clases que comienzan con el prefijo "Dt" hacen referencia a *datatypes*[3]. Estas clases presentan *Code Smells* asociados en su mayoría a convenciones y buen uso de la tecnología *Java*. Nuevamente, las clases que pertenecen a este conjunto presentan estructuras similares y comparten detecciones de varios *Code Smells*.

[3]Clases planas cuyo objetivo es transportar información. Estas clases pueden contener información de múltiples fuentes o tablas concentrada en una única clase simple.

Capa transversal					
Clase *	**LOC**	**Ocurrencias ****	**Densidad ****	**Detecciones ****	**% Ocurrencias de la clase sobre ocurrencias de la capa ****
ActivityType	73	16	0,22	4	24,62 %
DtState	59	6	0,1	2	9,23 %
DtTransition	49	6	0,12	4	9,23 %
KezmoObjectType	37	6	0,16	3	9,23 %
DtSequentialWorkflow	21	5	0,24	3	7,69 %
WorkflowStatus	29	4	0,14	3	6,15 %
DtStateMachineWorkflow	21	4	0,19	2	6,15 %
DtWorkflowVersion	67	4	0,06	2	6,15 %
DtEvent	33	3	0,09	2	4,62 %
DtForBlock	3	2	0,67	2	3,08 %

Tabla 5.13: Clases con mayor cantidad de ocurrencias de *Code Smells* en la capa transversal.
* La columna sintetiza el nombre de la clase como "x", cuyo nombre en el proyecto PIS_N sigue la nomenclatura *"x.java"*.
** de *Code Smells*.

El resto de las clases que se presentan en la tabla 5.13 (que no son *datatypes*), son enumerados. Como consecuencia, estas clases también presentan una estructura muy similar entre sí. La clase *ActivityType.java* presenta el 24,62 % del total de ocurrencias de *Code Smells* de la capa. Dentro de los cuatro *Code Smells* que se presentan en la clase, *Magic numbers should not be used* (analizado en profundidad en la sección 5.1.1) presenta 13 ocurrencias, mientras que los *Code Smells* restantes de la capa presentan una única ocurrencia. Las clases *KezmoObjectType.java* y *WorkflowStatus.java* presentan detecciones de los mismos *Code Smells*.

Analizando las categorías a las que pertenecen los *Code Smells* detectados en la capa transversal, se puede observar que las categorías de la taxonomía extendida con mayor cantidad de ocurrencias son: *Conventions, Java technology, Change preventers, Dispensables*. En particular se nota que la totalidad de las ocurrencias de la categoría *Change preventers* son a causa de ocurrencias del *Code Smell Magic numbers should not be used*. Las características mencionadas de las clases pertenecientes a la capa transversal, donde no hay lógica de negocio compleja y las clases presentan responsabilidades muy limitadas, explica el hecho de que las categorías de la taxonomía extendida con mayor cantidad de ocurrencias sean las mencionadas.

La tabla 5.14 muestra la distribución de ocurrencias de *Code Smells* por clase de la capa de servicios.

Capa de servicios					
Clase *	**LOC**	**Ocurrencias ****	**Densidad ****	**Detecciones ****	**% Ocurrencias de la clase sobre ocurrencias de la capa ****
WorkflowResource	370	32	0,09	11	84,21 %
DBResource	46	3	0,07	3	7,89 %
CORSFilter	17	1	0,06	1	2,63 %
MyAppConfig	12	1	0,08	1	2,63 %
ExceptionMapper	11	1	0,09	1	2,63 %
APIResponse	33	0	0	0	0 %
ResponseStatus	4	0	0	0	0 %

Tabla 5.14: Clases con mayor cantidad de ocurrencias de *Code Smells* para la capa de servicios.
* La columna sintetiza el nombre de la clase como "x", cuyo nombre en el proyecto PIS_N sigue la nomenclatura *"x.java"*.
** de *Code Smells*.

De la tabla 5.14 se desprende que las ocurrencias de la capa de servicios predominan en la clase *WorkflowResource.java*. La clase presenta el 84,21 % del total de ocurrencias de *Code Smells* de la capa de servicios. Esta clase forma parte del *core* del proyecto PIS_N y expone los servicios *REST* a ser consumidos por la capa de presentación. Por lo tanto, *WorkflowResource.java* es uno de los puntos de entrada a la capa de servicios y a su vez se comunica con la capa de negocio a través de la clase *Controller.java*. De los 11 *Code Smells* que se detectan en la clase, se destaca *Exception handlers should preserve the original exceptions* (analizado en la sección 5.1.1) con 18 ocurrencias. El resto de los *Code Smells* detectados en la clase no superan las dos ocurrencias. Cabe destacar que esta clase será analizada en profundidad posteriormente en esta RQ, al estudiar la distribución de *Code Smells* por clase.

El otro punto de entrada a la capa de servicios es la clase *DBResource.java*. Esta clase presenta tres *Code Smells*, asociados a convenciones, con una ocurrencia cada uno.

Observaciones

Las capas que contienen la mayor cantidad de ocurrencias de *Code Smells* son las capas de negocio y de acceso a datos. La capa de negocio presenta una distribución más homogénea de las ocurrencias que la capa de acceso a datos. A su vez, observando las categorías a las que pertenecen los *Code Smells* detectados en estas dos capas, se aprecia la presencia de todas las categorías de la taxonomía extendida de Mäntylä.

La capa de servicios es la que presenta la menor densidad de ocurrencias de *Code Smells* por línea de código. Esta capa, en conjunto con la capa transversal, presentan *Code Smells* asociados principalmente a las categorías: *Conventions, Java technology, Dispensables, Change preventers y Exceptions*. Al presentar responsabilidades limitadas, y por lo tanto, carecer de lógica de negocio compleja, es esperable que las capas mencionadas presenten pocas ocurrencias de *Code*

Smells asociados a categorías como *Bloaters, Couplers, Encapsulator, OOAbusers*.

A partir del análisis realizado de la distribución de *Code Smells* por capas del proyecto PIS_N, se desprende que las clases *core* del proyecto centralizan gran parte de las ocurrencias de cada capa. En este sentido, como se propuso anteriormente, sería recomendable pensar en una refactorización de las clases que presentan *Code Smells* relacionados a las responsabilidades de clase, distribuyendo las responsabilidades en clases más pequeñas.

Asimismo, en tres de las cuatro capas analizadas en esta sección se identificaron conjuntos de clases de estructuras similares, clases que extienden de *Activity.java* de la capa de negocio, clases *datatypes* y enumerados de la capa transversal, clases de la capa de acceso a datos con la responsabilidad de acceder y persistir a las tablas de la base de datos. Los *Code Smells* que se detectan en estas clases presentan ocurrencias similares entre sí. Por ejemplo, la ocurrencia del *Code Smell Loggers should be "private static final.ªnd should share a naming convention* en las clases que extienden de *Activity.java*. Se considera que esta situación se podría evitar o disminuir su frecuencia si se ejecutaran las herramientas automatizadas de detección sobre el código fuente desde el inicio del proyecto. Con el objetivo de estudiar y refactorizar tempranamente los defectos de diseño, una posible estrategia sería desarrollar pruebas de concepto, ejecutar las herramientas automatizadas sobre estas pruebas y corregir los *Code Smells* detectados. Como resultado se obtendría un código de mejor calidad, aunque no nos asegura que no se introduzcan errores nuevos. Sin embargo, minimiza el riesgo de propagación de errores en clases con estructuras similares.

En la tabla 5.10 se observa que la mayor densidad de *Code Smells* se presenta en las clases de acceso a datos. En parte, esto se debe a que la clase *EntitiesQueries.java* es la clase con más *Code Smells* del proyecto PIS_N. Desde nuestra perspectiva como estudiantes y según nuestra experiencia laboral y académica, creímos que la capa de negocio sería la capa con mayor densidad de *Code Smells*, dado que allí se suelen implementar las clases con características complejas, mayores responsabilidades y con una cantidad considerable de LOCs.

Distribución de *Code Smells* por clases en el proyecto PIS_N

En el proyecto PIS_N se detectan *Code Smells* en el 100 % de las clases *Java*, que ascienden a un total de 105 clases distribuidas en las cuatro capas del *backend*. En la tabla 5.15 se presentan las cinco clases con mayor cantidad de ocurrencias de *Code Smells*, excluyendo las clases vinculadas al *testing* unitario.

Capa	**Clase**	**LOC**	**Ocurrencias de *Code Smells***	**Ocurrencias de *Code Smells* por LOC**
Acceso a Datos	*EntitiesQueries.java*	107	269	2,51
Acceso a Datos	*Persistence.java*	736	87	0,12
Negocio	*EventListener.java*	333	55	0,17

Sigue en la página siguiente.

Capa	**Clase**	**LOC**	**Ocurrencias de *Code Smells***	**Ocurrencias de *Code Smells* por LOC**
Servicios	*WorkflowResource.java*	370	32	0,09
Negocio	*ForBlockActivity.java*	123	25	0,20

Tabla 5.15: Clases con mayor cantidad de ocurrencias de *Code Smells* en el proyecto PIS_N

Siguiendo el análisis estructurado de las secciones previas, para el análisis de las clases con mayor cantidad de ocurrencias se tiene en cuenta la siguiente información:

- descripción de la clase.
- ocurrencias de *Code Smells* en la clase.
- tiempo de refactorización de la clase.
- Errores reportados relacionados a la clase en la revisión por parejas de PIS_N.
- observaciones.

1- ***EntitiesQueries.java***

- Descripción: la clase se crea con el objetivo de contener como constantes cada consulta a la base de datos. Al realizar la misma consulta desde distintos métodos se obtiene la constante asociada a la consulta y se sustituyen los parámetros necesarios.
- Ocurrencias de *Code Smells*: En la tabla 5.16 se presentan los *Code Smells* detectados en la clase. Para cada *Code Smell* se presentan las categorías a las que pertenece y la cantidad de ocurrencias.

EntitiesQueries.java		
Code Smell	**Categoría taxonomía extendida**	**Ocurrencias de *Code Smells***
Class variable fields should not have public accessibility	*OOAbusers, Encapsulators*	83
Static non-final field names should comply with a naming convention	*Conventions*	83
"public static"fields should be constant	*Java technology*	83
Lines should not be too long	*Clarity and readability*	17

Sigue en la página siguiente.

Code Smell	**Categoría de la taxonomía extendida**	**Ocurrencias de *Code Smells***
Classes should not have too many fields	*Bloaters*	1
Many field attribute not complex	*Dispensables*	1
Utility classes should not have public constructors	*Java technology*	1

Tabla 5.16: *Code Smells* de la clase *EntitiesQueries.java*

Se observa que las 269 ocurrencias están distribuidas en siete *Code Smells* detectados, clasificados en siete categorías de la taxonomía extendida. Los primeros tres *Code Smells* detallados en la tabla 5.16 están entre los *Code Smells* con mayor cantidad de ocurrencias en PIS_N.

El *Code Smell Static non-final field names should comply with a naming convention* presenta 83 de las 88 ocurrencias que se detectan en el código del producto.

Los *Code Smells Class variable fields should not have public accessibility* y *"public static"fields should be constant* presentan todas sus ocurrencias detectadas en *EntitiesQueries.java*.

El *Code Smell Lines should not be too long* presenta 17 de sus 52 ocurrencias en esta clase. Este último *Code Smell* se encuentra contemplado en el estándar definido por PIS_N, sin embargo, las ocurrencias en esta clase no fueron reportadas en las revisiones.

- Tiempo de refactorización de la clase: se estima en 48 horas. Sin embargo, considerando como solución agregar un constructor privado en la clase, y definir todos los atributos de la clase y la clase como finales, se reduciría el tiempo de refactorización de la clase. Adicionalmente, es probable que la solución propuesta corrija todos los *Code Smells* presentados en la tabla 5.16, a excepción de *Lines should not be too long*. Como consecuencia, entendemos que no es adecuado sumar los tiempos de refactorización estimados de los *Code Smells* detectados en la clase ya que seguramente este valor sea mayor al que se deba invertir realmente en esta refactorización.

- Errores reportados: no existen registros de errores reportados en esta clase.

- Observaciones: las limitaciones de las herramientas en cuanto a contextualización de las ocurrencias no permite identificar cuál es la refactorización indicada para solucionarlo. En este sentido, se detectan varios *Code Smells* sobre las mismas líneas de código relacionados al mismo problema de origen.

2- ***Persistence.java***

- Descripción: la clase pertenece al *core* del sistema y se encarga de la persistencia a la base de datos a través de clases de acceso a datos.

- Ocurrencias de *Code Smells*: En la tabla 5.17 se presentan los *Code Smells* detectados en la clase. Para cada *Code Smell* se presentan las categorías a las que pertenece y la cantidad de ocurrencias.

Persistence.java		
Code Smell	**Categoría taxonomía extendida**	**Ocurrencias de *Code Smells***
Exceptions should not be thrown in finally blocks	*Exceptions*	23
catch clauses should do more than rethrow	*Java technology, Exceptions*	14
Magic numbers should not be used	*Change preventers*	11
Boolean expressions should not be gratuitous	*Good coding practices*	7
Method parameters should be declared with base types	*Good coding practices*	7
Sections of code should not be commented out	*Dispensables*	3
Local variable and method parameter names should comply with a naming convention	*Conventions*	3
Cognitive Complexity of methods should not be too high	*Bloater*	2
Control flow statements ïf", "for", "while", "switch.ªnd "try"should not be nested too deeply	*Bloater*	2
String literals should not be duplicated	*Change preventers, Dispensables*	2
Unused "private"methods should be removed	*Dispensables*	2
Exception handlers should preserve the original exceptions	*Exceptions*	2
Methods should not be too complex	*Bloaters*	1
NPath Complexity	*Bloaters*	1
Classes should not be coupled to too many other classes (Single Responsibility Principle)	*OOAbusers, Couplers*	1
Source files should not have any duplicated blocks	*Dispensables*	1
Underscores should be used to make large numbers readable	*Conventions*	1
"main"should not "throw.ªnything	*Java technology, Exceptions*	1

Sigue en la página siguiente.

Code Smell	**Categoría de la taxonomía extendida**	**Ocurrencias de *Code Smells***
Catches should be combined	*Java technology, Exceptions*	1
Declarations should use Java collection interfaces such as "Listrather than specific implementation classes such as "LinkedList"	*Java technology*	1
Mutable members should not be stored or returned directly	*Java technology*	1

Tabla 5.17: *Code Smells* en la clase *Persistance.java*

La clase *Persistence.java* presenta 87 ocurrencias de 21 *Code Smells*, resultando en la segunda clase con mayor cantidad de ocurrencias en el proyecto PIS_N. Las ocurrencias de esta clase se encuentran distribuidas en 736 líneas de código.

El *Code Smell Exceptions should not be thrown in finally blocks* presenta en esta clase 23 de sus 87 ocurrencias en el proyecto. En RQ2 (sección 5.1.2) se describió el *Code Smell*.

El *Code Smell Magic numbers should not be used* es el segundo *Code Smell* con mayor cantidad de ocurrencias en el proyecto, en esta clase se detectan 11 de ellas. En RQ1 (sección 5.1.1) se estudió el *Code Smell*.

Exceptions es la categoría con mayor cantidad de ocurrencias de *Code Smells* en la clase, donde los más incurridos son *Exceptions should not be thrown in finally blocks y catch clauses should do more than rethrow.*

- Tiempo de refactorización de la clase: es de 22 horas, lo cual es ampliamente superior comparación a los tiempos de las demás clases del proyecto que como máximo llegan a 7 horas.

- Errores reportados: El 8 % de los errores reportados corresponden a la clase *Persistence.java*. Entre estos errores el equipo de PIS_N menciona que "en la clase *Persistence.java* se indica en una única línea el código correspondiente a un try-catch". Este error no se corresponde con ningún *Code Smell* detectable automáticamente por las herramientas utilizadas en nuestra investigación.

- Observaciones: Esta clase tiene aproximadamente siete veces más líneas de código que la clase *EntitiesQueries.java* y el doble de *Code Smells* detectados, pero menor cantidad de ocurrencias de *Code Smells*. La densidad de *Code Smells* de esta clase (0,12 CS/LOC) no es significativa en comparación con la clase *EntitiesQueries.java* que es la más incurrida y de mayor densidad de *Code Smells* (2,51 CS/LOC).

  A través de una revisión manual de código observamos que la clase centraliza todos los accesos a la base de datos. Esta decisión de diseño puede que

no haya sido la mejor, ya que esta clase es candidata a centralizar la lógica de acceso a datos del sistema, asumiendo demasiadas responsabilidades. En este sentido, la clase presenta seis ocurrencias de *Code Smells* dentro de la categoría *Bloaters* asociados a complejidad, por lo que se identifica un exceso de responsabilidades. Asimismo, se infiere que la clase es candidata a presentar los *AntiPatterns Blob y Brain Class*.

Por las características de la clase, esta es propensa a sufrir cambios con frecuencia, aumentando la posibilidad de introducir *Code Smells*. Como posible refactorización, creemos que se podría desagregar la clase separando los accesos a bases de datos por entidades, donde cada una de estas nuevas clases agrupe los accesos a bases de datos por entidad, como se sugiere en RQ2 (sección 5.1.2). Esto reduciría en gran forma la complejidad de estas clases, obteniendo un código más claro y simple, y por lo tanto más mantenible. Esta refactorización es aplicable a otras clases del proyecto donde se centraliza gran parte de la lógica de negocio.

3- ***EventListener.java***

- Descripción: la clase contiene toda la lógica asociada a suscripciones y al manejo de los distintos eventos que se pueden presentar en el sistema.
- Ocurrencias de *Code Smells*: En la tabla 5.18 se presentan los *Code Smells* detectados en la clase. Para cada *Code Smell* se presentan las categorías a las que pertenece y la cantidad de ocurrencias.

EventListener.java		
Code Smell	**Categoría taxonomía extendida**	**Ocurrencias de *Code Smells***
Unused method parameters should be removed	*Dispensables*	18
String literals should not be duplicated	*Change preventers, Dispensables*	11
Exception handlers should preserve the original exceptions	*Exceptions*	7
Raw types should not be useds	*Change preventers*	3
if ... else ifçonstructs should end with .elseçlauses	*Good coding practices*	3
Boxed "Boolean"should be avoided in boolean expressions	*Java technology*	3
Control flow statements if", "for", "while", "switch.and "try"should not be nested too deeply	*Bloaters*	2
Static non-final field names should comply with a naming convention	*Conventions*	2

Sigue en la página siguiente.

Code Smell	**Categoría de la taxonomía extendida**	**Ocurrencias de *Code Smells***
Loggers should be "private static final.ªnd should share a naming convention	*Conventions*	1
Modifiers should be declared in the correct order	*Conventions*	1
Local variables should not be declared and then immediately returned or thrown	*Good coding practices*	1
Fields should not be initialized to default values	*Java technology*	1
Utility classes should not have public constructors	*Java technology*	1

Tabla 5.18: *Code Smells* en la clase *EventListener.java*

Las 55 ocurrencias de los *Code Smells* detectados en la clase se distribuyen en 333 líneas de código. Las ocurrencias se encuentran asociadas a 14 *Code Smells* que se clasifican en siete categorías distintas de la taxonomía extendida. La categoría que presenta mayor cantidad de ocurrencias en la clase es *Dispensables* (54,5 %) donde *Unused method parameters should be removed* es el *Code Smell* más incurrido en la clase (33 %). La segunda categoría con mayor cantidad de ocurrencias es *Change preventers* (25,5 %) donde *String literals should not be duplicated* es el segundo *Code Smell* más incurrido en la clase (20 %). En tercer lugar se presenta la categoría de *Exceptions*, donde su único *Code Smell* presente en la clase es *Exception handlers should preserve the original exceptions.*

- Tiempo de refactorización de la clase: es de 7 horas aproximadamente, siendo el tercer mayor tiempo de refactorización de una clase en PIS_N. Tras un análisis manual del código, sugerimos posibles soluciones para los *Code Smells* con mayor cantidad de ocurrencias en esta clase:
  - eliminar los parámetros no usados en los métodos de clase.
  - crear constantes y sustituir de todas las ocurrencias de *Strings* duplicados por estas constantes.
  - registrar la excepción con un *framework* para registro y luego hacer el *throw* correspondiente.
- Errores reportados: El 8 % de los errores reportados corresponden a la clase *EventListener.java*. Entre estos errores el equipo de PIS_N menciona que "se atrapa la excepción genérica". Este error reportado está relacionado con el *Code Smell* antes mencionado, *Exception handlers should preserve the original exceptions.* Se revisa manualmente el código de la clase y se verifica que este error no fue corregido por el equipo de PIS_N.
- Observaciones: la densidad de *Code Smells* en la clase (0,17 CS/LOC) no es significativa en comparación con la clase más incurrida *Entities-Queries.java* (2,51 CS/LOC). Sin embargo, la densidad de esta clase es

superior a la de la clase *Persistence.java*, siendo la segunda una clase con mayor cantidad de ocurrencias de *Code Smells*.

4- ***WorkflowResource.java***

- Descripción: la clase define todos los *endpoints* utilizados para obtener, listar, crear, editar y eliminar flujos secuenciales y máquinas de estados, así como para pausar y reactivar flujos de ambos tipos, y para verificar (simular ejecución) flujos secuenciales.
- Ocurrencias de *Code Smells*: En la tabla 5.19 se presentan los *Code Smells* detectados en la clase. Para cada *Code Smell* se presentan las categorías a las que pertenece y la cantidad de ocurrencias.

WorkflowResource.java		
Code Smell	**Categoría taxonomía extendida**	**Ocurrencias de *Code Smells***
Exception handlers should preserve the original exceptions	*Exceptions*	18
String literals should not be duplicated	*Change preventers, Dispensables*	2
Local variables should not be declared and then immediately returned or thrown	*Good coding practices*	2
"throws"declarations should not be superfluous	*Java technology, Exceptions*	2
Generic exceptions should never be thrown	*Exceptions*	2
Control flow statements if", "for", "while", "switch.ªnd "try"should not be nested too deeply	*Bloaters*	1
Raw types should not be used	*Change preventers*	1
Source files should not have any duplicated blocks	*Change preventers*	1
Unused assignments should be removed	*Change Preventers*	1
Classes from "sun."packages should not be used*	*Java technology*	1
Try-catch blocks should not be nested	*Exceptions*	1

Tabla 5.19: *Code Smells* de la clase *WorkflowResource.java*

Las 32 ocurrencias de los 11 *Code Smells* detectados se distribuyen en 370 líneas de código de la clase *WorkflowResource.java* y se categorizan en seis categorías de la taxonomía extendida. Al igual que en la clase *Persistence.java*, *Exceptions* es la categoría que presenta mayor cantidad de

ocurrencias (72 %) de *Code Smells* en la clase, siendo *Exception handlers should preserve the original exceptions* el *Code Smell* más incurrido (56 %). Al igual que en *EventListener.java*, la segunda categoría con mayor cantidad de ocurrencias es *Change preventers* (15,6 %) y dentro de esta categoría el *Code Smell String literals should not be duplicated* es el más incurrido (6,3 %).

- Tiempo de refactorización de la clase: es de 6 horas aproximadamente. Como consecuencia, es la clase en el cuarto lugar en cuanto a clases con mayor tiempo de refactorización de PIS_N.
- Errores reportados: El 3 % de los errores reportados corresponden a la clase *WorkflowResource.java*. Entre estos errores el equipo de PIS_N menciona que esta clase "tiene varias operaciones por lo tanto es considerada compleja". Revisando manualmente el código de esta clase y sus operaciones entendemos que este error reportado no ha sido corregido.
- Observaciones: la densidad de *Code Smells* en la clase (0,09 CS/LOC) no es significativa en comparación con la clase más incurrida *EntitiesQueries.java* (2,51 CS/LOC), siendo la de menor densidad entre las cinco clases con mayor cantidad de ocurrencias de *Code Smells*.

  Si bien no se detectaron *Code Smells* relacionados a las responsabilidades de la clase, coincidimos con el equipo de PIS_N que es una clase que podría estar asumiendo demasiadas responsabilidades y que eventualmente podría crecer demasiado si se siguieran agregando funcionalidades al producto. Con el objetivo de mitigar el crecimiento de la clase creemos que sería conveniente crear clases más pequeñas agrupadas por funcionalidad, redistribuyendo sus responsabilidades.

5- ***ForBlockActivity.java***

- Descripción: implementa la iteración sobre un conjunto de actividades que se encuentra definido en un *workflow* secuencial.
- Ocurrencias de *Code Smells*: En la tabla 5.20 se presentan los *Code Smells* detectados en la clase. Para cada *Code Smell* se presentan las categorías a las que pertenece y la cantidad de ocurrencias.

ForBlockActivity.java		
Code Smell	**Categoría taxonomía extendida**	**Ocurrencias de *Code Smells***
Control flow statements ïf", "for", "while", "switch.ªnd "try"should not be nested too deeply	*Bloaters*	3
Exception handlers should preserve the original exceptions	*Exceptions*	3

Sigue en la página siguiente.

Code Smell	**Categoría de la taxonomía extendida**	**Ocurrencias de *Code Smells***
Lines should not be too long	*Clarity and readability*	3
Strings literals should be placed on the left side when checking for equality	*Java technology, Exceptions*	3
Try-catch blocks should not be nested	*Exceptions*	3
String literals should not be duplicated	*Change Preventers, Dispensables*	2
Cognitive Complexity of methods should not be too high	*Bloater*	1
Loggers should be "private static final.ªnd should share a naming convention	*Conventions*	1
Modifiers should be declared in the correct order	*Conventions*	1
NPath Complexity	*Bloater*	1
Redundant casts should not be used	*Dispensables*	1
Source files should not have any duplicated blocks	*Dispensables*	1
Switches should be used for sequences of simple "String"tests	*Clarity and readability*	1
Variables should not be declared before they are relevant	*Clarity and readability, Good coding practices*	1

Tabla 5.20: *Code Smells* en la clase *ForBlockActivity.java*

Las 25 ocurrencias de los 14 *Code Smells* detectados están distribuidas en las 125 líneas de código de la clase *ForBlockActivity.java*. Los *Code Smells* detectados pertenecen a ocho categorías de la taxonomía extendida. La categoría *Exceptions* es nuevamente la categoría que presenta mayor cantidad de ocurrencias de *Code Smells* en la clase (24 %), donde el *Code Smell Exception handlers should preserve the original exceptions*, presenta el 50 % de ocurrencias de la categoría (un porcentaje apenas menor al presentado en la clase *Persistence.java*). El *Code Smell Try-catch blocks should not be nested* también se presenta en la clase *WorkflowResource.java* pero no así en *Persistence.java*. La segunda categoría con mayor cantidad de ocurrencias en esta clase es *Bloaters* (20 %). Dentro de esta categoría el *Code Smell* que presenta mayor cantidad de ocurrencias es *Control flow statements if, for, while, switch and try should not be nested too deeply* (12 %). Este *Code Smell* también se presenta en las clases *Persistance.java, EventListener.java, WorkflowResource.java*, sin embargo, dada la baja cantidad de ocurrencias en estas clases y en el proyecto

en general (10 ocurrencias) no se profundiza en el mismo.

- Tiempo de refactorización de la clase: es de 5 horas aproximadamente. En este sentido, la clase se encuentra quinta en cuanto a las clases con mayor tiempo de refactorización de PIS_N.

- Errores reportados: no existen registros de errores reportados en esta clase. Sin embargo hay reglas específicas asociadas por ejemplo al manejo de bloques *try-catch* en el estándar definido por PIS_N que se corresponden a *Code Smells* detectados automáticamente en la clase por las herramientas utilizadas en nuestra investigación. Revisando manualmente el código de esta clase verificamos que este error no ha sido corregido.

- Observaciones: la densidad de *Code Smells* de *"ForBlockActivity.java"* (0,20 CS/LOC) no es significativa en comparación con la clase más incurrida, *"EntitiesQueries.java"* (2,51 CS/LOC), sin embargo es mayor a la de las demás clases con mayor cantidad de ocurrencias de *Code Smells*.

Observaciones

Las clases que presentan mayor cantidad de ocurrencias de *Code Smells* son las que tienen mayor esfuerzo de refactorización. Destacamos que mayor cantidad de ocurrencias de *Code Smells* en una clase no implica que la densidad de los *Code Smells* sea mayor que en otra clase con menor cantidad de ocurrencias.

Exceptions es la categoría con mayor cantidad de ocurrencias de *Code Smells* en la mayoría de las clases estudiadas. El correcto manejo de errores en el sistema contribuye a la construcción de productos robustos y mantenibles.

Algunos de los errores reportados en la revisión por parejas de PIS_N relacionados a las clases estudiadas no son detectables automáticamente por las herramientas utilizadas en el estudio. Existen *Code Smells* reportados por PIS_N y detectados por las herramientas que no fueron corregidos por el equipo de PIS_N.

5.1.4. RQ4: ¿En qué *AntiPatterns* incurren los estudiantes?

En esta sección se analizan las ocurrencias de *AntiPatterns* y *Code Smells* relacionados a *AntiPatterns* que surgen de las herramientas seleccionadas para la investigación. Primero se analizará la herramienta *Code Smells and AntiPatterns detection* por ser la única herramienta incluida en la investigación que permite detectar *Antipatterns*, y luego se analizarán las herramientas *SonarQube, CheckStyle, PMD*.

Plugin Code Smells and AntiPatterns detection

Fueron seleccionados 12 *Code Smells* y *AntiPatterns* de la herramienta *Code Smells and AntiPatterns detection*. Dado que en la documentación proporcionada para la herramienta no se distingue entre cuáles de ellos son *Antipatterns* y cuáles son *Code Smells*, fue necesario realizar esta clasificación de forma manual tomando como base el catálogo de *Antipatterns* definido en la sección 2.3.

A partir de esta clasificación surgen los siguientes *Antipatterns*: *Functional decomposition, Spaghetti code* y *Swiss army knife*. Cabe destacar el caso particular de la regla *Antisingleton*; por su nombre podríamos pensar que es un *Antipattern* asociado al patrón de diseño *Singleton*. Sin embargo, luego de compararlo con el *Antipattern Lack of singleton* definido en nuestro conjunto de *AntiPatterns* (sección 2.3), se llega a la conclusión de que no representan el mismo *Antipattern*. Es decir, se observa que ambas reglas no detectan el mismo defecto de diseño. Como consecuencia, se realiza una revisión en la literatura existente con el fin de determinar si efectivamente *Antisingleton*, utilizando la definición del *plugin*, es propiamente un *Antipattern*. Como resultado de la revisión no se encuentra información referente al mismo. Por lo tanto, no podemos confirmar que estemos ante la presencia de un *Antipattern* y para nuestra investigación será considerado como un *Code Smell*.

En la tabla 5.21 se detallan los *Code Smells* y *AntiPatterns* del *plugin Code Smells and Antipatterns detection* a tener en cuenta en la investigación, nuestro mapeo manual con *AntiPatterns* (en caso de que corresponda) y la cantidad de ocurrencias del defecto de diseño detectadas en el proyecto PIS_N.

Mapeo *Code Smells-Antipatterns*		
AntiPattern/Code Smell	**Mapeo con el conjunto de *AntiPatterns***	**Ocurrencias detectadas**
Antisingleton	-	1
Lazy class	*Poltergeists*	5
Many field attribute not complex	*Blob*	1
Base class knows derived class	*Law of demeter violation*	0
Base class should be abstract	*Tangle*	0
Message chains	*Yoyo problem*	0
Refused parent bequest	*Functional decomposition*	0
Speculative generality	*Functional decomposition*	0
Tradition breaker	*Poltergeists*	0
Functional decomposition	-	0
Spaghetti code	-	0
Swiss army knife	-	0

Tabla 5.21: *Code Smells* y *AntiPatterns* del *plugin Code Smells and AntiPatterns detection* incluidos en el *Quality Profile* a utilizar en la investigación.

Como se observa en las últimas tres filas de la tabla 5.21, no se detectan ocurrencias de *AntiPatterns*. De los tres *Code Smells* detectados del *plugin* analizado en esta sección, dos de ellos tienen relación con *Antipatterns*. A continuación se analizan estos dos *Code Smells* con el fin de poder determinar la presencia del *Antipattern* mapeado. Para el análisis se tienen en cuenta los siguientes puntos:

- descripción del *Code Smell*.
- clases en las que se incurre en el *Code Smell*.
- análisis sobre relación con el *AntiPattern* mapeado.
- sugerencia de resolución.

- observaciones.

1- ***Lazy class***

- Descripción: una clase que tiene pocos atributos y métodos. Este *Code Smell* fue mapeado directamente con el *AntiPattern Poltergeists*. Como se menciona en la sección 2.3, este *AntiPattern* refiere a clases con responsabilidades y roles limitados en el sistema, por lo que su ciclo de vida efectivo es corto.

- Clases en las que se incurre: cuatro de las cinco ocurrencias del *Code Smell* se detectaron en clases de *datatypes*: *DtForBlock.java, DtKezmoEventType.java, DtSimpleActivity.java, DtSystemType.java*. Una revisión de las clases permite identificar que *DtForBlock.java* se encuentra vacía, es decir, no tiene atributos ni método definidos. Además, las tres clases restantes solo tienen el método constructor implementado. A diferencia de las ocurrencias mencionadas, la quinta ocurrencia, se presenta en la clase *DALException.java*:

```
public class DALException extends Exception {
 public DALException(String msj) {
 super(msj);
 }
}
```

- Relación con el *Antipattern* mapeado: analizando las ocurrencias detectadas, se pudo observar que cuatro de las cinco clases en las que se detectó el *Code Smell* no son utilizadas, es decir, no son instanciadas ni se hace uso de ellas en el código fuente. La clase restante *DtSystemType* se utiliza una única vez, de la siguiente forma:

```
    try {
    dt = mapper.readValue(eventString, DtSystemType.class);
    // TODO: System todavía no está implementado
throw new NotImplementedException();
```

  Como se puede observar, la instancia no es utilizada ya que forma parte de un código que no fue implementado. Esto confirma que el ciclo de vida de la instancia es muy breve, ya que finaliza al lanzar la excepción. En base a la revisión manual de código realizada, entendemos que el *Antipattern Poltergeists* se encuentra presente en las detecciones del *Code Smell*, ya que las clases detectadas efectivamente presentan responsabilidades y roles limitados o nulos en el sistema, por lo tanto, su ciclo de vida efectivo es corto.

- Sugerencia de resolución: las cuatro clases que no son utilizadas podrían ser eliminadas del código fuente, luego de hacer una revisión y validar si estas clases no forman parte de algún posible desarrollo a futuro. Para el

caso de la clase *DtSystemType.java* sería necesario analizar si el desarrollo pendiente será realizado en una versión posterior o no.

2- ***Many field attribute not complex***

- Descripción: una clase que declara muchos atributos pero que no es compleja y, por lo tanto, es más probable que sea algún tipo de clase de datos que contenga valores sin proporcionar un comportamiento. El *Code Smell* ha sido mapeado indirectamente con el *AntiPattern Blob*. Como se menciona en la sección 2.3, *Blob* refiere a clases que monopolizan los procesos y el procedimiento teniendo parte importante de las responsabilidades del sistema.
- Clases en las que se incurre: este *Code Smell* es detectado sólo una vez en el proyecto PIS_N, en la clase *EntitiesQueries.java*.
- Relación con el *Antipattern* mapeado: la clase *EntitiesQueries.java* no tiene métodos implementados. Su única responsabilidad es brindar las consultas a base de datos como *Strings*. Como consecuencia, se confirma que el *Blob* no se encuentra presente en esta clase.
- Sugerencia de resolución: se dejarían de detectar las ocurrencias asociadas a esta regla a partir de la redefinición de las variables como constantes.

Code Smells de *SonarQube, CheckStyle, PMD* mapeados a *AntiPatterns*

En la tabla 5.22 se detallan los *Code Smells* mapeados manualmente a *AntiPatterns*, detectados en el proyecto PIS_N mediante alguna de las siguientes herramientas: *SonarQube, CheckStyle, PMD*. La tabla 5.22 esta agrupada por *AntiPattern*. Para cada uno de ellos se listan los *Code Smells* mapeados junto con su cantidad de ocurrencias en el proyecto PIS_N.

Mapeo *Code Smells-Antipatterns*	
AntiPattern	***Code Smell* junto con su cantidad de ocurrencias**
lava-flow	*String literals should not be duplicated (126), Magic numbers should not be used (91), Redundant casts should not be used (38), Unused "private"fields should be removed (20), Unused method parameters should be removed (20), Unnecessary imports should be removed (12), Sections of code should not be commented out (11), Nested blocks of code should not be left empty (10), Boolean expressions should not be gratuitous (7), Methods should not be empty (6), Unused assignments should be removed (4), Redundant modifiers should not be used (3), Unused "private"methods should be removed (2), Unused local variables should be removed (2), Boolean literals should not be redundant (1), Classes should not be empty (1)*

Sigue en la página siguiente.

AntiPattern	***Code Smell* mapeado con su cantidad de ocurrencias**
law-of-demeter	*Class variable fields should not have public accessibility (83), Utility classes should not have public constructors (20), Constructors of an .ªbstractçlass should not be declared "public"(2)*
tangle	*"public static"fields should be constant (83), Classes should not be coupled to too many other classes (Single Responsibility Principle) (2)*
blob	*NPath Complexity (14), Cognitive Complexity of methods should not be too high (11), Control flow statements ïf", "for", "while", "switch.ªnd "try"should not be nested too deeply (10), Methods should not be too complex (10), Classes should not have too many fields (3), Classes should not be coupled to too many other classes (Single Responsibility Principle) (2)*
brain-class	*NPath Complexity (14), Cognitive Complexity of methods should not be too high (11), Control flow statements ïf", "for", "while", "switch.ªnd "try"should not be nested too deeply (10), Methods should not be too complex (10), Classes should not have too many fields (3)*
spaghetti-code	*NPath Complexity (14), Cognitive Complexity of methods should not be too high (11), Control flow statements ïf", "for", "while", "switch.ªnd "try"should not be nested too deeply (10), Methods should not be too complex (10), Classes should not have too many fields (3)*
cut-and-paste-programming	*Source files should not have any duplicated blocks (32)*
extending-a-mutable-class	*Mutable members should not be stored or returned directly (29)*
functional-decomposition	*Methods should not be too complex (10), Methods should not have too many lines (5), Methods should not have too many parameters (4)*
Lack of Singleton	*Only one getInstance method is allowed (1)*

Tabla 5.22: *AntiPatterns* del conjunto definido en la sección 2.3 mapeados a *Code Smells* detectados en el proyecto PIS_N.

Observando los *Code Smells* detectados para cada *Antipattern*, se observa que hay *Code Smells* que se repiten para distintos *Antipatterns*, entre ellos encontramos *NPath Complexity, Methods should not be too complex*, entre otros. En este sentido, resulta interesante realizar una revisión manual de estos *Code Smells* con el fin de detectar posibles *Antipatterns* asociados.

Tomando como insumo los *Code Smells* seleccionados mapeados con los *AntiPatterns* del conjunto definido para la investigación [5], y las ocurrencias detectadas en el proyecto PIS_N (tabla 5.22), se decide realizar una revisión manual de algunos *Code Smells* con el fin de confirmar o no la presencia de *AntiPat-*

terns. Cabe destacar que la selección de los *Code Smells* para la revisión manual no siguió ningún criterio específico, sino que fue basada en el conocimiento del código fuente adquirido durante la investigación.

A continuación se describen aquellos *Antipatterns* que se detectaron a través de revisiones de código manuales. Para el análisis de estos *AntiPatterns* se tienen en cuenta los siguientes puntos:

- descripción del *AntiPattern*.
- *Code Smells* detectados que dan indicios al *AntiPattern*.
- clases en las que se incurre en el *AntiPattern*.
- análisis de la presencia del *AntiPattern*.
- reglas definidas en el documento de estándar de *Java* por el equipo PIS_N que se podrían vincular al *AntiPattern*.
- errores reportados en la revisión por parejas de PIS_N vinculados al *AntiPattern*.
- sugerencia de resolución del *AntiPattern*.

1- ***Cut and paste programming***

- Descripción: refiere a código reutilizado mediante la copia de declaraciones y fragmentos de código.
- *Code Smells* relacionados al *Antipattern*: *Source files should not have any duplicated blocks (SonarQube)*. El *Code Smell* está asociado a bloques de código duplicado en una misma clase.
- Clases en las que se incurre: ha sido detectado en 32 clases del producto final, de las cuales 19 son clases de la capa de negocio, 12 son clases de la capa de acceso a datos y una clase es de la capa de servicios.
- Análisis de la presencia del *Antipattern*: las 32 clases presentan el *Code Smell Source files should not have any duplicated blocks*. La revisión manual realizada sobre el *Code Smell* incurrido en estas clases nos permite verificar la presencia del *AntiPattern Cut and paste programming* en todas sus ocurrencias ya que por definición del *Code Smell* se asocian a fragmentos de código exactamente iguales que creemos son reutilizados mediante la acción "copiar y pegar". Es importante mencionar que la detección del *Code Smell* se realiza a nivel de clase por limitaciones de la herramienta, pero puede suceder que estos bloques de código se repitan en varias clases, por lo que la cantidad de ocurrencias del *Antipattern* podría variar.
- Reglas del documento de estándar de *Java*: no existen reglas específicas asociadas a este *AntiPattern*.
- Errores reportados: No existen errores reportados asociados a este *AntiPattern*.

- Sugerencia de resolución: una posible solución para eliminar este *AntiPattern* es extraer el bloque de código detectado como duplicado a un método aparte e invocar a este método en todos los casos que corresponda.

2- ***Blob***

- Descripción: refiere a una clase que monopoliza los procesos y el procesamiento. De esta forma contiene la mayor parte de las responsabilidades del programa.
- *Code Smells* relacionados al *Antipattern*: Los *Code Smells NPath Complexity(Checkstyle), Cognitive Complexity of methods should not be too high(SonarQube), Control flow statements ïf", "for", "while", "switch.ªnd "try"should not be nested too deeply(SonarQube), Methods should not be too complex(SonarQube), Classes should not have too many fields(SonarQube), Classes should not be coupled to too many other classes (Single Responsibility Principle)(SonarQube)*, están asociados principalmente a métodos excesivamente complejos, clases con muchos atributos y clases que están demasiado acopladas.
- Clases en las que se incurre: ha sido detectado las clases *Persistence.java* y *Controller.java*.
- Análisis de la presencia del *AntiPattern*: la clase *Persistence.java* centraliza toda la lógica de acceso y persistencia sobre la base de datos, es decir, existe una única clase en todo el proyecto PIS_N con la responsabilidad de consultar, insertar, actualizar y eliminar información de la base de datos. La clase está acoplada a 21 clases diferentes, de aquí la detección del *Code Smell Classes should not be coupled to too many other classes (Single Responsibility Principle)*. Por otra parte, la clase presenta una ocurrencia de los *Code Smells NPath Complexity, Methods should not be too complex* y dos ocurrencias de los *Code Smells Control flow statements ïf", "for", "while", "switch.ªnd "try"should not be nested too deeply y Cognitive Complexity of methods should not be too high*. La clase posee 28 métodos, 888 LOCs y es utilizada 87 veces en el proyecto. Por todo lo mencionado, consideramos que estamos ante la presencia del *AntiPattern Blob* en la clase mencionada.
  La otra clase analizada es *Controller.java*, que es el punto de entrada a la capa de negocio, es decir, la capa de servicios se comunica con la capa de negocio a través de esta clase. La clase presenta una ocurrencia del *Code Smell Classes should not be coupled to too many other classes (Single Responsibility Principle)* ya que se encuentra acoplada a 22 clases. También presenta una ocurrencia de los *Code Smells Control flow statements ïf", "for", "while", "switch.ªnd "try"should not be nested too deeply y Cognitive Complexity of methods should not be too high*. Realizando un análisis similar, se observa que la clase presenta 18 métodos, 386 LOCs y es utilizada 26 veces. Desde nuestra perspectiva la clase resulta menos compleja en comparación con *Persistence.java*, pero no deja de tener una responsabilidad excesiva a nivel del proyecto PIS_N, centralizando mucha lógica de negocio y procesamiento.

- Reglas del documento de estándar de *Java*: no existen reglas específicas asociadas a este *AntiPattern*.
- Errores reportados: los errores reportados asociados a este *AntiPattern* han sido en las clases:
  - *WorkflowResource.java*, donde el equipo PIS_N reporta: "tiene varias operaciones por lo tanto es considerada compleja" [11]. En este sentido, realizamos una revisión en esta clase y no se detectan ocurrencias de *Code Smells* relacionados a complejidad de clases. Concluimos que el equipo PIS_N resolvió el error reportado.
  - *Persistence.java* donde el equipo PIS_N reporta: "Existen métodos de prueba en la clase *Persistence* que no son utilizados en el sistema con otro fin que no sea verificación" [11]. No se detectan métodos de prueba en la revisión que realizamos sobre esta clase, por lo que asumimos que ese error fue corregido.
- Sugerencia de resolución: la solución propuesta para ambos casos implica desagregar la clase en clases más pequeñas con el fin de distribuir las responsabilidades.

3- ***Spaghetti code***

- Descripción: refiere a estructuras de software *ad hoc* que dificultan la extensión y optimización del código.
- *Code Smells* relacionados al *Antipattern*: NPath Complexity(Checkstyle), Cognitive Complexity of methods should not be too high(SonarQube), Control flow statements ïf", "for", "while", "switch.ªnd "try"should not be nested too deeply(SonarQube), Methods should not be too complex(SonarQube), Classes should not have too many fields(SonarQube), estos *Code Smells* están asociados principalmente a complejidad excesiva en métodos y clases con demasiados atributos.
- Clases en las que se incurre: fue detectado en cuatro clases, tres de ellas pertenecen a la capa de negocio (*Controller.java, WorkflowInstanceManager.java, ForBlockActivity.java*) y una clase perteneciente a la capa de acceso a datos (*Persistence.java*).
- Análisis de la presencia del *AntiPattern*: las cuatro clases mencionadas presentan un método en el cual se detectan los *Code Smells Cognitive Complexity of methods should not be too high y Control flow statements ïf", "for", "while", "switch.ªnd "try"should not be nested too deeply.* Analizando los métodos en detalle se puede observar el alto grado de complejidad que presentan, dificultando el entendimiento y con una alta probabilidad de complejizar una posible extensión del producto. Desde nuestro punto de vista las cuatro clases presentan el *AntiPattern Spaghetti code*.
- Reglas del documento de estándar de *Java*: no existen reglas específicas asociadas a este *AntiPattern*.

- Errores reportados: no existen errores reportados asociados a este *AntiPattern*.
- Sugerencia de resolución: reducir estos métodos complejos en métodos más simples, con menos líneas de código. Extraer partes del código complejas a métodos auxiliares y llamar a estos métodos desde el método principal.

Observaciones generales

El *plugin Code Smells and AntiPatterns detection* no detectó *AntiPatterns* de forma automática, aunque sí se pudo confirmar la presencia de un *AntiPattern* a partir de la detección de un *Code Smell* del *plugin*. Las revisiones manuales del código fuente realizadas para confirmar o no la presencia de un *AntiPattern* resultaron ser complejas y costosas. Creemos que esta misma complejidad dificulta la detección de *AntiPatterns* por herramientas automatizadas.

El mapeo manual realizado para establecer la relación de *Code Smells* con *AntiPatterns* fue de gran utilidad. Tomando como entrada las ocurrencias de *Code Smells* relacionados con *AntiPatterns*, se realizaron revisiones manuales del código fuente donde se pudo confirmar la presencia de cuatro *AntiPatterns*. Dentro de los *AntiPatterns* encontrados existen distintos niveles de gravedad. Por ejemplo, el *Antipattern Poltergeists* fue detectado en clases que no se utilizan en el proyecto, por lo que el impacto que puede tener sobre el producto en sí es casi nulo. Mientras que el *Antipattern Blob* fue detectado sobre dos clases que son parte del *core* del producto, por lo que su impacto seguramente sea mucho mayor. Creemos que es importante definir una planificación de refactorización adecuada, priorizando los *AntiPatterns* de mayor impacto. Consideramos que el *Quality Profile* definido para la investigación, en conjunto con el mapeo manual realizado, puede ser de utilidad para otros proyectos de desarrollo, siendo un buen punto de entrada para la detección de *AntiPatterns*.

Al observar los *Code Smells* detectados y su relación con *Antipatterns* se notó que ciertos *Code Smells* detectados en el proyecto PIS_N se encuentran relacionados a más de un *AntiPattern*. Por ejemplo, el *Code Smell NPath Complexity* se encuentra asociado a tres *AntiPatterns*. Una estrategia para las revisiones manuales de código enfocadas en la detección de *AntiPatterns* podría ser planificar y comenzar las revisiones a partir de las ocurrencias de estos *Code Smells*.

Página ha sido intencionalmente dejada en blanco.

Capítulo 6

Conclusiones y trabajos a futuro

En este capítulo se presentan las conclusiones, las principales contribuciones del proyecto de grado y algunas posibles líneas de trabajo que permitan ampliar y profundizar la investigación realizada en otros contextos y con otras tecnologías.

6.1. Conclusiones

Para poder cumplir con el objetivo planteado en nuestro proyecto de grado, **conocer y analizar diferentes tipos de defectos de diseño (*Code Smells* y *AntiPatterns*) en los que incurren los estudiantes de la asignatura PIS**, se realizaron diversas tareas. Primeramente se llevó a cabo una revisión de antecedentes con el propósito de profundizar en los conceptos relacionados a *Code Smells* y *AntiPatterns*. De esta revisión se obtuvo el conjunto de *Code Smells y Antipatterns* a analizar. A su vez, se realizó un estudio de un conjunto de herramientas de detección automatizada de estos defectos de diseño, dando como resultado la selección de *SonarQube*, los *plugins* oficiales *PMD* y *CheckStyle*, y el *plugin Code Smells and AntiPatterns detection*. Por último, se configuró y ejecutó las herramientas seleccionadas en un proyecto de PIS (al cual llamamos PIS_N) y se analizó los resultados obtenidos. Por cuestiones de tiempo y alcance del proyecto, se decidió reducir el conjunto de proyectos PIS a analizar a un único proyecto.

Los resultados muestran que los estudiantes del proyecto PIS_N incurren principalmente en *Code Smells* relacionados a convenciones del lenguaje, al buen uso de la tecnología *Java*, excepciones y a aspectos relacionados a código innecesario, duplicado o redundante. Las primeras dos categorías (convenciones y tecnología *Java*) se asocian principalmente a aspectos relacionados a la claridad, legibilidad y mantenibilidad del código. Mientras que el resto podrían asociarse a malas decisiones de diseño.

La categoría *Exceptions* se presenta como una de las categorías más incurridas. Si bien se desconocen los motivos por los cuales esta categoría fue tan incurrida, creemos que podría ser la falta de conocimiento por parte de los estudiantes de PIS_N acerca de los riesgos que tiene para el producto no manejar las excepciones correctamente, como ser errores que pasen desapercibidos por no

capturarlos y registrarlos correctamente. Errores de este tipo fueron detectados en las revisiones por parejas, algunos de ellos fueron corregidos y otros no.

La clase con más ocurrencias del proyecto PIS_N (*EntitiesQueries.java*) presenta *Code Smells* relacionados a la incorrecta definición de reglas en el estándar de implementación definido por el equipo del proyecto PIS_N (inconsistentes con el estándar oficial de *Java* [31]), así como a la ausencia de otras reglas presentes en el estándar oficial. Creemos que el uso de este estándar incompleto y con errores podría justificar la cantidad de ocurrencias detectadas de *Code Smells* relacionados a convenciones y a la tecnología *Java*.

Adicionalmente, analizando la distribución de los *Code Smells* por clase, se observa que tres de las cinco clases con más ocurrencias forman parte del *core* del producto. Este tipo de clases suelen ser complejas, tienen una alta tasa de modificación y centralizan la lógica de negocio. Si no se presta especial atención a mejorar la calidad del código de estas clases, es probable que afecten en gran medida el mantenimiento y la calidad interna del proyecto. Otra observación respecto a la distribución es que ciertas clases con estructuras similares presentan los mismos *Code Smells*. Suponemos que la propagación de *Code Smells* en el código fuente puede haber surgido por la acción "copiar y pegar".

En relación a los *AntiPatterns*, no se realizaron detecciones de *Antipatterns* de forma automática. Sin embargo, fueron detectados 4 *AntiPatterns* a través de revisiones manuales del código fuente a partir de *Code Smells relacionados a AntiPatterns*. Los *AntiPatterns* detectados manualmente son: *Poltergeists* en clases de *Data Types* de la capa transversal, *Blob* en las clases *core "Persistence.java"* y *"Controller.java"*, *Cut and Paste Programming* en 32 clases y *Spaghetti Code* en 6 clases. Cabe destacar que si bien se detectaron algunos *AntiPatterns* de forma manual, no se realizó una revisión total de los *Code Smells* relacionados a *AntiPatterns*, por lo que creemos que podrían existir otros *AntiPatterns* que podrían ser detectados a partir de estas revisiones.

La taxonomía de Mäntylä (figura 2.1) no resultó suficiente para la categorización de los *Code Smells* y *AntiPatterns* detectados. Se detectó *Code Smells* que no eran contemplados en la misma. Por tal motivo, se extendió dicha taxonomía, incorporándose las siguientes categorías: *Conventions, Java technology, Good coding practices, Clarity and readability, Exceptions, Performance, Security*. Esta taxonomía extendida permite categorizar una mayor cantidad de *Code Smells* comunes a proyectos de software de múltiples características implementados en *Java*.

El proceso de selección de *Code Smells* y *AntiPatterns* deja como resultado un *Quality Profile*. Allí se incluye un conjunto de reglas que permiten detectar *Code Smells* y *AntiPatterns* relacionados a defectos de diseño mediante las herramientas utilizadas para el proyecto. Este *Quality Profile* puede ser usado por practicantes de software para realizar análisis de código estático en sus proyectos *Java* de forma automatizada con foco en la detección de estos defectos de diseño. A su vez, se realizó un mapeo de los *Code Smells* relacionados con *AntiPatterns*. Este mapeo puede ser usado en la detección manual de *AntiPatterns* en código *Java*.

Por otro lado, nuestro trabajo permitió identificar algunas limitaciones de las herramientas seleccionadas. Entre ellas se destacan la dificultad en la detección de *AntiPatterns* y la poca precisión en los tiempos de refactorización estimados

para la resolución de algunos de los *Code Smells*. Debido a estas limitaciones, consideramos que no se debería evaluar la calidad del código **únicamente** en base a lo detectado por herramientas automatizadas. Sugerimos complementar las mismas con otras actividades, como ser las revisiones de código. Sugerimos que estas revisiones estén guiadas por el mapeo de *Code Smells* relacionados a *AntiPatterns* generado en el proyecto de grado, permitiendo identificar posibles *AntiPatterns* que partan de las ocurrencias detectadas de estos *Code Smells*.

Debido a los problemas ya mencionados que fueron detectados en el estándar de codificación del PIS_N (incompleto y con errores), y que la mayoría de los *Code Smells* reportados por las revisiones por pareja eran detectables mediante las herramientas seleccionadas, sugerimos enfocar dichas revisiones en aspectos del código que hoy en día sean detectables únicamente de forma manual.

6.2. Contribuciones

A continuación se destacan los aportes más importantes de este trabajo.

- La obtención de una primer aproximación al tipo de *Code Smells* y *AntiPatterns* en los que incurren estudiantes de PIS, quedando la misma disponible para el uso de la Facultad de Ingeniería, UdelaR.
- La extensión de la taxonomía de Mäntylä, quedando la misma disponible para el uso de la comunidad científica.
- La creación de un *Quality Profile* que incluye la detección de *AntiPatterns* y *Code Smells* relacionados a defectos de diseño, quedando disponible para su uso en proyectos *Java*.
- El mapeo de un conjunto de *Code Smells* relacionados con *AntiPatterns* seleccionados, quedando disponible para su uso en proyectos *Java*.
- La información relacionada a los defectos de diseño detectados (tipos, distribución, criticidad, etc), la cual puede servir como base para futuros trabajos de investigación.

6.3. Trabajos a futuro

Así como se menciona en la Tesis de Maestría "Detectando y Evitando Defectos de Diseño de Software" [9], creemos que podría ser interesante continuar la exploración de las causas por las cuales los estudiantes avanzados de nuestra Facultad incurren en defectos de diseño, especialmente en *Code Smells* y *AntiPatterns*.

Replicar el análisis realizado en un contexto similar permitiría contrastar los resultados obtenidos, así como reforzar las conclusiones descritas y ampliar las mismas. Por otro lado, se podría replicar este análisis incorporando nuevas herramientas de detección, complementarias a las utilizadas, con el fin de ampliar la cantidad de *Code Smells* y *AntiPatterns* a analizar.

Adicionalmente, se podría replicar este análisis en otros contextos de interés. Por ejemplo, se podrían seleccionar proyectos PIS que sigan otros procesos de

desarrollo, permitiendo estudiar la relación de los *Code Smells* y *AntiPatterns* detectados según el proceso utilizado. Análogamente, otra opción podría ser una variación en la tecnología utilizada, siendo necesario redefinir el conjunto de *Code Smells* y *AntiPatterns*, así como el conjunto de herramientas de detección. Por otro lado, como forma de minimizar la influencia de la inexperiencia de los estudiantes PIS en los resultados, se podría replicar el análisis en código de la industria. De esta forma se estarían considerando desarrolladores con mayor diversidad o mayor nivel de experiencia.

Apéndice A

Anexos

A.1. Tablas

A.1 *Code Smells* básicos presentados por Folwler en 1999 [17].

***Code Smells* Básicos**	
Code Smell	**Breve descripción**
Long method	Refiere a un método que es muy largo. Es difícil de entender, cambiar o extender.
Large class	Refiere a una clase con múltiples responsabilidades. Tiene varias variables instanciadas o métodos.
Long parameter list	Refiere a una función con una lista de parámetros extensa y difícil de entender.
Primitive obsession	Refiere a la utilización de primitivas en lugar de clases pequeñas (*datatypes*).
Data clumps	Refiere a la eliminación de elementos, dentro de un grupo de elementos que aparecen juntos en el software, que hace que otros elementos pierdan sentido.
Switch statements	Refiere a la utilización de códigos de tipo o detección de tipo de clase en tiempo de ejecución en lugar de polimorfismo.
Temporary field	Refiere a una clase que tiene una variable que se utiliza únicamente en algunas situaciones.
Refused bequest	Situación en la que una "clase hija" no soporta todos los métodos/datos que hereda.
Alternative classes with different interfaces	Situación en la que las diferencias en las interfaces de clases similares conducen a código duplicado.

Sigue en la página siguiente.

Code Smell	**Breve descripción**
Parallel inheritance hierarchies	Situación en la que existen dos clases paralelas con herencia y ambas herencias deben ser extendidas.
Lazy class	Refiere a una clase que no está haciendo lo suficiente y debe ser removida.
Data class	Refiere a una clase que contiene datos y no contiene lógica.
Duplicated code	Refiere a la existencia de código redundante.
Speculative generality	Existencia de código innecesario generado a partir de la anticipación de futuros cambios en el software. Por lo general, agrega complejidad innecesaria al software.
Message chains	Situación en la que una clase solicita un objeto de otro objeto, que lo solicita de otro y así sucesivamente.
Middle man	Situación en la que una clase delega la mayoría de sus tareas a otras clases.
Feature Envy	Situación en la cual un método está más interesado en otras clases que en la que actualmente se encuentra implementado.
Inappropiate intimacy	Situación en la cual dos clases están muy acopladas entre ellas. Las clases pasan demasiado tiempo profundizando en los atributos privados de la otra.
Divergent change	Refiere a una clase que debe ser modificada continuamente.
Shotgun surgery	Situación en la cual se deben modificar múltiples clases por un cambio general pequeño del sistema.
Incomplete libary class	Situación en la cual el software utiliza una librería no completa, que debe extenderse en funcionalidad.
Comments	Existencia de comentarios en el código para compensar código mal estructurado.

Tabla A.1: *Code Smells* básicos de Fowler.

A.2 Conjunto de *AntiPatterns* a considerar en el proyecto de grado.

Conjunto de *AntiPatterns*	
AntiPattern	**Breve descripción**
Blob	Clase que monopoliza los procesos y el procesamiento. Contiene la mayor parte de las responsabilidades del programa.

Sigue en la página siguiente.

AntiPattern	**Breve descripción**
Continuous Obsolescence	Refiere a problemas de interoperabilidad entre las versiones del software, principalmente como consecuencia del rápido avance de la tecnología.
Lava flow	Consecuencia del cambio constante en el diseño del software y la no actualización o eliminación de código e información de diseño obsoletos.
Ambiguous viewpoint	Refiere a una separación ambigua entre las interfaces del software y la implementación del código.
Functional decomposition	En una arquitectura orientada a objetos, la implementación de las clases se realiza de forma similar a un lenguaje estructural generando gran complejidad en los procedimientos.
Poltergeists	Refiere a clases con roles limitados y ciclos de vida cortos y efectivos. Por lo general estas clases solo inician procesos para otros objetos.
Golden hammer	Tecnología o concepto familiar aplicado obsesivamente para resolver muchos problemas de software.
Dead end	Refiere a la utilización/modificación de un componente de software reutilizable no mantenido por su proveedor, generando problemas de soporte y mantenimiento.
Spaghetti code	La estructura del software *ad hoc* dificulta la extensión y optimización código.
Input kludge	Refiere al uso de algoritmos *ad hoc* para manejar los parámetros de entrada del programa.
Cut and paste programming	Código reutilizado mediante la copia de declaraciones y fragmentos de código.
Base bean	En una arquitectura orientada a objetos se utiliza la herencia solo para poder usar sus métodos. Esta mala solución de diseño viola el principio de sustitución de Liskov[1].
Bloated facade	El patrón de diseño de fachada proporciona una interfaz unificada a un subsistema. La fachada se considera "inflada" cuando se asigna dinámicamente o refiere a algunas grandes subestructuras asignadas dinámicamente, además de subestructuras más pequeñas.

Sigue en la página siguiente.

[1]El principio de sustitución de Liskov establece que "las funciones que utilizan punteros o referencias a clases base deben poder utilizar objetos de clases sin saberlo ".

AntiPattern	**Breve descripción**
Brain class	Clase compleja que acumula un exceso de inteligencia, generalmente en la forma de múltiples métodos. Su estrategia de detección complementa la estrategia de *Blob* (implica romper el principio de encapsulación) ya que detecta las clases excesivamente complejas. Cabe destacar que estas clases no son consideradas *Blob* ya que no acceden abusivamente a datos de clases "satélite" o porque estos datos son un poco más cohesivos.
Constant interface	Se declara una constante en una interfaz sin ningún método.
Construction of zombie references	Referencia a un objeto basura. La referencia nunca se usa.
Extensive processing	Ocurre cuando un proceso de larga ejecución monopoliza el procesador e impide un buen tiempo de respuesta general.
Dormant references	Refiere a una referencia creada cuando su destino estaba activo, pero que persiste después de que su destino se convierte en basura.
God component	Componente que encapsula múltiples servicios. Tiene mucha responsabilidad otorgada a través de distintos métodos con múltiples parámetros de intercambio. Existe un alto acoplamiento con los servicios de comunicación.
Lack of bridge	Uso del patrón *Bridge* (para desacoplar la implementación) en un mal contexto donde la abstracción y la implementación están estrechamente relacionadas.
Lack of singleton	Uso incorrecto del patrón *Singleton* ya que, previo a instanciar la clase, se realiza una comprobación condicional para verificar que sea un objeto *singleton*.
Lack of strategy	Uso incorrecto del patrón *Strategy* para organizar y encapsular los algoritmos en una jerarquía. El patrón es usado en un mal contexto, generando algoritmos encapsulados en una gran clase compleja o en una jerarquía de clases donde cada subclase ofrece una implementación distinta del algoritmo.
Law of demeter violation	Refiere a la violación del principio "no hables con extraños".
Missing builder	Clase que contiene múltiples constructores con distintos parámetros, lo que lleva a que un objeto pueda tener diferentes representaciones.

Sigue en la página siguiente.

AntiPattern	**Breve descripción**
Missing factory method	Clase que se instancia directamente sin usar fábricas. Se crean instancias de diferentes clases en diferentes rutas de ejecución.
Missing prototype	Refiere al uso de múltiples instancias de un mismo objeto.
Swiss army knife	Interfaz excesivamente compleja. Contiene una gran cantidad de firmas en un intento de satisfacer todas las necesidades posibles que tenga el sistema.
Tangle	Gran grupo de objetos/paquetes cuyas relaciones están tan interconectados que un cambio en cualquiera de ellos podría afectar a todos los demás.
The knot	Conjunto de servicios de baja cohesividad que están altamente acoplados.
Tiny service	Servicio que representa completamente una abstracción pero implementa únicamente un subconjunto reducido de los métodos necesarios.
Vendor lockin	Ocurre en sistemas que adoptan una tecnología/producto y a partir de allí dependen de la arquitectura implementada por el propietario.
Wide subsystem interface	Interfaz muy amplia, provoca un acoplamiento estrecho e indeseado entre el paquete y el resto del mundo.
Yoyo problem	Cambio continuo de las definiciones de las clases para seguir el flujo de control del programa. Implica una estructura difícil de entender debido a un exceso de fragmentación.

Tabla A.2: Conjunto de *AntiPatterns* a considerar en el proyecto de grado.

A.3 Tabla comparativa de las herramientas de detección automatizadas *SonarQube* y *SpotBugs*.

SonarQube* vs *SpotBugs		
Característica	***SonarQube***	***SpotBugs***
Versiones de *Java* soportadas	*Java* 15	*Java* 9
Ultimo *release*	Marzo, 2021	Marzo, 2021
Comunidad Activa	Si	Si
Usabilidad	Fácil, amigable, de aprendizaje rápido	Aplicación de escritorio, anticuada

Sigue en la página siguiente.

Característica	***SonarQube***	***SpotBugs***
Capacidades	Permite detectar *Code Smells, Bugs, Vulnerabilities, Security HotSpots*. Puede rastrear cobertura de pruebas unitarias y duplicación de código	Permite detectar *Bugs*. No distingue entre *Code Smells, Bugs, Vulnerabilities, Security HotSpots*
Técnica de detección	Basada en heurística	Basada en patrón de *Bug*
Integración	Se integra con *Jenkins*[2]. Se integra al IDE con *SonarLint*[3]	Se integra al IDE
Instalación y ejecución	Simple	Simple
Detecciones en el código PIS_N utilizando el *Quality Profile* por defecto de *Java*	945 *Code Smells* y 128 *Bugs*	315 *Bugs*

Tabla A.3: Tabla comparativa entre las herramientas *SonarQube* y *SpotBugs*

A.4 *Code Smells* a considerar en el proyecto de grado que complementan a la herramienta *SonarQube*.

Code Smells* que complementan a *SonarQube		
Code Smell	***Plugin***	**Breve descripción**
Throws Count	*CheckStyle*	Restringe el número de *throws* por método. Máximo permitido por defecto: cuatro
NPath Complexity	*CheckStyle*	La complejidad *NPath* de un método es el número de rutas de ejecución acíclicas a través de ese método. 200 es el número que indica que se deben tomar medidas para reducir la complejidad.
Avoid Accessibility Alteration	*PMD*	Métodos que permiten la alteración en tiempo de ejecución de la visibilidad de variables, clases o métodos, incluso si son privados. Viola el principio de encapsulación.
Only one getInstance method is allowed	*PMD*	No conviene que una implementación del patrón *Singleton* exponga de forma pública más de un método *getInstance()*.

Sigue en la página siguiente.

[2]https://www.jenkins.io/
[3]https://www.sonarlint.org/

Code Smell	***Plugin***	**Breve descripción**
Antisingleton	*plugin Code Smells and AntiPatterns detection*	Clase que proporciona variables de clase mutables que podrían usarse como variables globales.
Lazy class	*plugin Code Smells and AntiPatterns detection*	Una clase que tiene pocos campos y métodos.

Tabla A.4: *Code Smells* detectados por los *plugins PMD, CheckStyle o Code Smells and AntiPatterns detection* que no son detectados por *SonarQube* y seran tenidos en cuenta en la investigación.

A.5 Etiquetas de *SonarQube* que no son de interés para el proyecto de grado.

Etiquetas de *SonarQube* descartadas		
Etiqueta	**Descripción**	**Motivo de descarte**
Annotation	Reglas sobre anotaciones de *Java*.	No es de interés el análisis de herramientas o librerías específicas.
Tests	Reglas asociadas a *testing* unitario.	No es de interés el análisis de herramientas o librerías específicas.
Spring	Reglas asociadas al *framework* de *Spring*.	No es de interés el análisis de herramientas o librerías específicas.
Assertj	Reglas asociadas a *testing* unitario.	No es de interés el análisis de herramientas o librerías específicas.
Guava	Reglas específicas de la librería *Guava*.	No es de interés el análisis de herramientas o librerías específicas.
Javadoc	Reglas asociadas a documentación de *Java*. con *javadoc*	No es de interés el análisis de *javadoc*.
Comment	Reglas asociadas a documentación de *Java*.	No es de interés el análisis de *javadoc*.
Blocks	Contiene *templates* de reglas. A partir de estos *templates* se definen reglas.	Queda fuera del alcance la definición de reglas a partir de *templates*.
Misc	Contiene *templates* de reglas. A partir de estos *Templates* se definen reglas.	Queda fuera del alcance la definición de reglas a partir de *templates*.
Imports	Contiene *templates* de reglas. A partir de estos *templates* se definen reglas.	Queda fuera del alcance la definición de reglas a partir de *templates*.
Android	Reglas específicas para la tecnología *Android*.	No es de interés el análisis de herramientas o librerías específicas.

Sigue en la página siguiente.

Etiqueta	Descripción	Motivo de descarte
Whitespace	Contiene *templates* de reglas. A partir de estos *templates* se definen reglas.	Queda fuera del alcance la definición de reglas a partir de *templates.*
Debug	Reglas especifícas asociadas al modo *debug.*	No es de interés el análisis en relación al modo *debug.*

Tabla A.5: Etiquetas de *SonarQube* que no seran tenidas en cuenta en el proyecto de grado.

A.2. Pasos de instalación de herramientas

Requisitos

- *Hardware*: Para una instancia a pequeña escala (individual o en equipo) del servidor *SonarQube* se requiere al menos 2 GB de RAM para funcionar de manera eficiente y 1 GB de RAM libre para el sistema operativo.
- Versión de Java para servidor de *SonarQube*: Oracle JRE 11, OpenJDK 11.
- Java para *Sonar Scanner*:Oracle JRE 8, OpenJDK 8 o Oracle JRE 11, OpenJDK 11.
- Base de datos: PostgreSQL (9.3 - 9.6, 10, 11,12, 13), Microsoft SQL Server (12,13, 14,15), Oracle (XE Editions, 11G, 12C, 18C,19C).
- Navegador web: Habilitar JavaScript en el navegador.

Instalación A continuación se describen los pasos necesarios para instalar y configurar *SonarQube* en CentOS 7 con la edición *Community Edition.*

1. Descargar e instalar *SonarQube.*

   1.1 Descargar la última versión y descomprimir el archivo descargado en una carpeta. A modo de ejemplo, se toma como ubicación el directorio *opt/sonarqube.*

   1.2 Crear usuario y asignar permisos sobre la carpeta *opt/sonarqube.* Desde línea de comandos ejecutar:

   ```
   useradd sonar
   passwd sonar
   chown -R sonar:sonar /opt/sonarqube
   ```

   1.3 Verificar permisos de ejecución sobre los ejecutables dentro de las siguientes carpetas:
   /opt/sonarqube/bin/linux-x86-64
   /opt/sonarqube/elasticsearch/bin.

   1.4 Deshabilitar *system call filters* para evitar error al levantar *elasticsearch.* Editar archivo */opt/sonarqube/conf/sonar.properties* agregando la siguiente propiedad:

```
sonar.search.javaAdditionalOpts=-Dbootstrap.
system_call_filter=false
```

1.5 Opcional: En caso que se necesite exponer el servidor de *SonarQube* por https, se puede realizar instalando un *reverse proxy* y configurando SSL a este nivel[4].

1.6 Agregar regla en el *firewall* para habilitar entrada por el puerto 443. Editar el archivo */etc/sysconfig/iptables* agregando la siguiente línea:

```
-A INPUT -p tcp -m state --state NEW -m tcp --dport 443
-j ACCEPT
```

2. Configurar base de datos

2.1 Instalar PostgreSQL 13 para linux. Desde línea de comandos ejecutar:

```
sudo yum install -y https://download.postgresql.org/pub/
reporpms/EL-7-x86_64/pgdg-redhat-repo-latest.noarch.rpm
sudo yum install -y postgresql13-server
sudo /usr/pgsql-13/bin/postgresql-13-setup initdb
sudo systemctl enable postgresql-13
sudo systemctl start postgresql-13
```

2.2 Luego de la instalación de PostgreSQL se habrá creado un usuario "postgres", por lo que el siguiente paso será cambiar la contraseña de este usuario.

```
sudo passwd postgres
```

2.3 Cambiar la contraseña de usuario de la base de datos "postgres". Sustituir "NuevaPassword" por la nueva contraseña del usuario.

```
su - postgres
psql -d template1 -c "ALTER USER postgres WITH
PASSWORD 'NuevaPassword';"
```

2.4 Crear usuario y base de datos en PostgreSQL. Desde línea de comandos ejecutar las siguientes instrucciones:

```
psql -U postgres
CREATE ROLE sonar LOGIN PASSWORD 'sonar' NOINHERIT
CREATEDB;
CREATE DATABASE sonar WITH ENCODING 'UTF8' OWNER sonar
TEMPLATE=template0;
```

2.5 Asociar la base de datos recién creada a *SonarQube*. Editar el archivo *sonar.properties* ubicado en *opt/sonarqube/conf/* para configurar el acceso a la base de datos. Quitar los comentarios de las siguientes líneas. Asignar la base de datos, el usuario y la contraseña creados en el paso anterior.

[4]Ver http://httpd.apache.org/docs/2.4/ssl/

```
# User credentials.
# Permissions to create tables, indices and triggers must
be granted to JDBC user.
# The schema must be created rst.
sonar.jdbc.username=sonar
sonar.jdbc.password=sonar
#|{ PostgreSQL 9.3 or greater
# By default the schema named ‘‘public s used. It can be
overriden with the parameter \curretschema’’.
sonar.jdbc.url=jdbc:postgresql://localhost/sonar
```

3. Iniciar el Servidor *SonarQube*

   3.1 Desde línea de comandos ejecutar:

   ```
   sh /opt/sonarqube/bin/linux-x86-64/sonar.sh start
   ```

   Confirmar que el servidor inició correctamente al ver el siguiente mensaje:

   ```
   INFO  app[][o.s.a.SchedulerImpl] SonarQube is up
   ```

4. Acceder al servidor web. Verificar que se haya iniciado correctamente el servidor web en la dirección por defecto: *http://localhost:9000*. Seleccionar la opción *Log in* e ingresar el usuario y contraseña por defecto *(admin/admin)*. Cambiar la contraseña de usuario *admin*.

5. Análisis de código Java *Sonar Scanner* es el cliente por defecto de *SonarQube*. Al iniciar solicita al servidor las reglas del lenguaje a analizar y ejecuta el análisis de código línea por línea. Al finalizar envía el resultado al servidor *SonarQube*

   5.1 Descargar e instalar *Sonar Scanner*. Descargar la última versión y descomprimir el archivo descargado en una carpeta. A modo de ejemplo, se toma como ubicación el directorio *opt/sonar-scanner*.

   5.2 Configurar *Sonar Scanner*. Editar el archivo *sonar-scanner.properties* ubicado en *opt/sonar-scanner/conf* para configurar los datos del servidor.

   ```
   #|{ Default SonarQube server
   sonar.host.url=http://localhost:9000
   ```

   5.3 Verificar permisos de ejecución sobre los ejecutables dentro de la carpeta: */opt/sonar-scanner/bin/*.

   5.4 Crear un proyecto a analizar en *Sonarqube Server*. Acceder a *Sonarqube Server* (http://localhost:9000) y seleccionar *Add Project*. Asignar una clave de proyecto y un nombre. Generar un token en el paso siguiente. Estos tres datos serán utilizados en el siguiente punto.

   5.5 Configurar el proyecto a analizar. Crear el archivo *sonar-project. properties* en la carpeta raíz del proyecto con el código fuente a analizar. El contenido del archivo es el siguiente:

```
# must be unique in a given SonarQube instance
sonar.projectKey=keyDeNuestroProyecto
# this is the name and version displayed in the
SonarQube UI. Was mandatory prior to SonarQube 6.1.
sonar.projectName=NombreDeNuestroProyecto
sonar.projectVersion=1.0
# Path is relative to the sonar-project.properties le.
Replace \n" by \/" on Windows.
# This property is optional if sonar.modules is set.
sonar.sources=.
# Encoding of the source code. Default is default system
encoding
#sonar.sourceEncoding=UTF-8
sonar.java.binaries=.
sonar.login=tokenDeNuestroProyecto
```

5.6 Configurar variable de entorno para *Sonar Scanner*. Editar archivo */etc/profile* agregando al final del archivo la siguiente línea:

```
export PATH=/opt/sonar-scanner/bin:$PATH
```

5.7 Ejecutar análisis. Desde línea de comandos ubicarse en el directorio raíz de proyecto con el código fuente y ejecutar:

```
sonar-scanner
```

Al finalizar se puede consultar toda la información referente al análisis en el servidor *SonarQube*.

Página ha sido intencionalmente dejada en blanco.

Referencias

[1] M. Ratcliffe A. Eckerdal, R. McCartney and C. Zander. Can graduating students design software systems? *ACM SIGCSE Bulletin*, 2006.

[2] M. Azanza. Adimen investigación. curso ingeniería de software 2013/2014. ultima lección. *HiTZ Basque Center for Language Technologies. Universidad del País Vasco / Euskal Herriko Unibertsitatea*, page 17, 2013.

[3] N. Sanchez B. Silvotti and R. Lago. Code smells seleccionados para la investigación. *Repositorio Open Science, DOI:10.5281/zenodo.6053983*, 2022.

[4] N. Sanchez B. Silvotti and R. Lago. Code smells seleccionados para la investigación categorizados. *Repositorio Open Science, DOI:10.5281/zenodo.6054079*, 2022.

[5] N. Sanchez B. Silvotti and R. Lago. Mapeo entre code smells-antipatterns seleccionados para nuestra investigación. *Repositorio Open Science, DOI:10.5281/zenodo.6054131*, 2022.

[6] M. Bates. The online deskbook. *New York: Pemberton Press*, 1996.

[7] G. Booch. Object-oriented analysis and design with applications. *Addison-Wesley Professional*, 3, 2007.

[8] L. Thomas C. Loftus and C. Zander. Can graduating students de- sign: revisited. *Proceedings of the 42nd ACM technical symposium on Computer science education. ACM.*, 2011.

[9] V. Casella. Detectando y evitando defectos de diseño de software. *Programa de Maestría en Informática. PEDECIBA Informática, Facultad de Ingeniería, Universidad de la República*, 2021.

[10] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 1994.

[11] Grupo de ingeniería de Software Fing. Modelo de proceso mum - proyecto de ingeniería de software. *Instituto de Computación, Facultad de Ingeniería, UDELAR*, 1.1, 2021.

[12] M. Dodani. Patterns of anti-patterns. *Journal of Object Technology*, page 29–33, 2006.

[13] R. Johnson E. Gamma, R. Helm and J. Vlissides. Design patterns. *Addison-Wesley*, page 395, 1994.

[14] F. Engelbertink and H. Vogt. How to save on software maintenance costs. *Omnext White Paper.*, 2010.

[15] Y. Gueheneuc F. Khomh, M. Di Penta and G. Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering.*, 2012.

[16] M. Fowler. Refactoring: Improving the design of existng code. *Addison-Wesley Professional*, 1999.

[17] M. Fowler and K. Beck. Refactoring. *Pearson*, 2(1), 2018.

[18] R. Mirgalimova I. Khomyakov, Z. Makhmutov and A. Sillitti. Automated measurement of technical debt: A systematic literature review. *Innopolis University, Russian Federation*, page 12, 2019.

[19] S. Joc and E. Curran. Software quality. a framework for success in software development and support. *Addison Wesley*, 1994.

[20] M. Lanza and R. Marinescu. Object-oriented. metrics in practice: Using software metrics to characterize, evaluate, and improve the design of object-oriented systems. *Springer*, 2006.

[21] Y. Gueheneuc M. Abbes, F. Khomh and G. Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. *European Conference on Software Maintenance and Reengineering*, page 181–190, 2011.

[22] R. Marinescu. Measurement and quality in object-oriented design. *PhD thesis, Department of Computer Science, Politehnica University of Timisoara*, 2002.

[23] R. Martin. Design principles and design patterns. *objectmentor.com*, page 34, 2000.

[24] J. Garcia Molina. Xxi jornadas de ingenieria de software y bases de datos. *Universidad de Salamanca*, 2016.

[25] S. Moreno and D. Vallespir. ¿los estudiantes de pregrado son capaces de diseñar software? estudio de la relación entre el tiempo de codificación y el tiempo de diseño en el desarrollo de software. *Conferencia Iberoamericana de Ingenieria de Software 2018*, 2018.

[26] T. Mowbray. The seven deadly sins of object-oriented architecture. *OBJECT Magazine*, pages 22–24, 1997.

[27] B. Mynatt. Software engineering with students project guidance. *Upper Saddle River, NJ, USA: Prentice-Hall International*, 1990.

[28] M. Mäntylä. Bad smells in software - a taxonomy and an empirical study. *Helsinki University of Technology*, page 75, 2003.

[29] V. Lenarduzzi N. Saarimaki, M. Baldassarre and S. Romano. On the accuracy of sonarqube technical debt remediation time. *45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2019.

[30] The Institute of Electrical ang Electronics Engineers. Ieee standard glossary of software engineering terminology. *STANDAR COORDINATING COMMITTEE OF THE COMPUTER SOCIETY OF THE IEEE*, 1990.

[31] Oracle. The java tutorials. *Java Documentation*, 2018.

[32] Dr. Francisco José García Peñalvo. Ciencia de la computación e inteligencia artificial. *Capitulo 7 - Ingenieria de Software. Universidad de Salamanca*, pages 277–388, 2018.

[33] R. Pressman. Ingenieria de software. un enfoque practico. *Mc Graw Gill*, 7, 2010.

[34] J. Pérez. Refactoring planning for design smell correction in object-oriented software. *PhD thesis, Escuela Técnica Superior de Ingeniería informática*, 2011.

[35] M. Redondo. Manual de calidad y procedimientos para la gestión del sistema de calidad de una empresa de desarrollo de software. *Universidad de Sevilla*, pages 15–25, 2016.

[36] L. Rising. The patterns handbook. *Cambridge University Press*, 8(1), 1995.

[37] ISO/IEC JTC 1/SC 7 Software and systems engineering. Iso/iec 25010:2011. systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models. *International Organization for Standardization*, page 34, 2011.

[38] Open source. Open source html5 charts for your website. *https://www.chartis.org/*, 2021.

[39] R. McCartney T. Chen, S. Cooper and L. Schwartzman. The (relative) importance of software design criteria. *ACM SIGCSE Bulletin*, 37, 2005.

[40] E. Moonen V. Emden and L. Moonen. Java quality assurance by detecting code smells. *Reverse Engineering. Proceedings. Ninth Working Conference on. IEEE*, 2002.

[41] F. Palomba V. Lenarduzzi, N. Saarimäki and S. Lujan. A critical comparison on six static analysis tools: Detection, agreement, and precision. *LUT University, Finland*, page 35, 2021.

[42] H.McCormick W. Brown, R. Malveau and T. Mowbray. Antipatterns:refactoring software, architectures, and projects in crisis. *John Wiley & Sons*, 1998.

[43] B. Webster. Pitfalls of object-oriented development. *M & T Books*, 1995.

[44] L. Xu. A study on the relationship between findbugs warnings, metrics and expert judgments. *University of Stuttgart, Germany*, page 95, 2016.

Página ha sido intencionalmente dejada en blanco.

Índice de tablas

Índice de figuras