UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

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

Vanessa Casella

Programa de Maestría en Informática
PEDECIBA Informática, Facultad de Ingeniería
Universidad de la República

Montevideo – Uruguay
Setiembre de 2020

UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

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

Vanessa Casella

Tesis de Maestría presentada al Programa de Desarrollo de las Ciencias Básicas (PEDECIBA), como parte de los requisitos necesarios para la obtención del título de Magíster en Informática.

Director:
Ph.D. Diego Vallespir

Director académico:
Ph.D. Diego Vallespir

Montevideo – Uruguay
Setiembre de 2020

Casella, Vanessa

Detectando y Evitando Defectos de Diseño de Software / Vanessa Casella. - Montevideo: Universidad de la República, PEDECIBA Informática, Facultad de Ingeniería, 2020.

XVII, 164 p. 29, 7cm.

Director:

Diego Vallespir

Director académico:

Diego Vallespir

Tesis de Maestría – Universidad de la República, Programa en Informática, 2020.

Referencias bibliográficas: p. 79 – 90.

1. antipatterns, 2. code smells, 3. mapeo sistemático, 4. diseño detallado, 5. calidad de diseño. I. Vallespir, Diego, . II. Universidad de la República, Programa de Posgrado en Informática. III. Título.

INTEGRANTES DEL TRIBUNAL DE DEFENSA DE TESIS

Ph.D. Marcela Genero (Revisora)

Ph.D. Daniel Calegari

Ph.D. Juliana Herbert

Montevideo – Uruguay
Setiembre de 2020

Agradecimientos

Embarcarme en este viaje, que aunque personal, no hubiera sido posible sin llevar entre el equipaje el cariño y apoyo de muchas personas, y el amor incondicional de mi "perrhija" Arya. Les agradezco de corazón.

A mi tutor, Diego, por guiarme y comprender mis tiempos. Gracias por darme las herramientas para aprender y reflexionar, por animarme a seguir en momentos complicados y confiar en mí para realizar este trabajo.

A Gus, por su apoyo infinito y paciencia, por cuidarme como nadie, entender y respetar mis decisiones. Gracias por tu amor.

A mis padres, porque sin ellos nada de esto hubiera sido posible. Gracias por el esfuerzo que hicieron para que pudiera estudiar lejos de casa, sin dudas fue el mejor regalo que recibí. Gracias por el apoyo incondicional y por confiar en mí.

A mi hermano, mi persona favorita en el mundo, por ser mi ejemplo de perseverancia y dedicación.

Al resto de mi familia, en especial a mis abuelos "Yola", "Ñata" y "Pepe", por comprender mis ausencias y mimarme en todo momento.

A mis amigas y amigos de siempre y a las amistades que la vida me regaló en este último tiempo, por el aguante y aliento. Gracias por esas juntadas revitalizadoras. En especial agradecer a mi hermana del corazón, "Mansu".

Al Grupo de Ingeniería de Software de Facultad de Ingeniería, por su aporte y soporte. Me llena de orgullo poder formar parte de ese equipo.

A la Comisión Sectorial de Investigación Científica, por confiar en mí y apoyar esta investigación.

A Sonda, por ser flexibles en todo aspecto. Gracias por entender que este trabajo era importante para mí.

En especial a José Manuel, el ángel de la familia, por ser la persona más luchadora que he conocido. Sé que nos vamos a volver a encontrar.

RESUMEN

El diseño de software es un proceso creativo y fundamental en la construcción de software de calidad. La creación de un diseño simple y eficiente puede ser una tarea muy compleja y su construcción requiere de ciertas habilidades que al parecer no están desarrolladas en los estudiantes de pregrado.

Las malas prácticas de diseño de software originan defectos de diseño, que no necesariamente producen errores de compilación o de ejecución pero que afectan negativamente a los factores de calidad del software. Los defectos pueden surgir en diferentes niveles de granularidad, como son los *antipatterns* y *code smells*. Desafortunadamente, identificar y corregir estos defectos de diseño puede ser una tarea muy compleja.

Este trabajo pretende contribuir al conocimiento existente sobre técnicas y herramientas de detección de *antipatterns* y *code smells*, a partir de una revisión secundaria de la literatura. Además, busca conocer los defectos de diseño en los que incurren los estudiantes de pregrado de nuestra Facultad e investigar, mediante estudios experimentales, si estos pueden ser evitados utilizando un conjunto de plantillas de diseño.

El resultado de la revisión secundaria es un catálogo de defectos de diseño, donde para cada uno se incluyen técnicas y herramientas utilizadas para detectarlo. Este catálogo puede ser utilizado en otras investigaciones o por profesionales de la industria como guía en el diseño de software.

Del análisis de defectos de diseño, se observa que los estudiantes de pregrado incurren en una gran variedad de estos defectos y no mejoran la calidad interna del software cuando utilizan plantillas para representar el diseño. El uso de estas plantillas no evitan ni disminuyen la aparición de ciertos defectos de diseño.

Comprender los tipos de defectos de diseño que están presentes en proyectos que desarrollan los estudiantes, sirve como puntapié inicial para generar nuevas hipótesis y diseñar nuevos estudios experimentales. Además, estos resultados pueden servir como insumo en la investigación sobre las prácticas de diseño y

cómo se enseña a diseñar software.

Palabras clave:
antipatterns, code smells, mapeo sistemático, diseño detallado, calidad de diseño.

ABSTRACT

Software design is a creative and fundamental process in building quality software. Creating a simple and efficient design can be a very complex task and its construction requires certain skills that apparently are not developed in undergraduate students.

Bad software design practices lead to design flaws, which may not affect compilation or runtime errors but negatively affect software quality factors. Defects can arise at different levels of granularity, such as antipatterns and code smells. Unfortunately, identifying and correcting these design flaws can be a very complex task.

This work aims to contribute to the existing knowledge about detection techniques and tools for antipatterns and code smells, based on a mapping study. In addition, this thesis seeks to know the design defects that the undergraduate students of our Engineering School incur and investigate, through experimental studies, whether these can be avoided by the use of a set of design templates.

The result of the mapping study is a catalog of design defects, where each includes techniques and tools used to detect it. This catalog can be used in other research or by industry professionals as a guide in software design.

From design defects analysis, we note that undergraduates incur in a wide variety of these defects and do not improve the internal quality of the software when templates are used to represent the design. The use of these templates does not prevent or diminish the appearance of certain design defects.

Understand the types of design flaws that are present in projects that students develop, serves as an initial kick to generate new hypotheses and design new experimental studies. In addition, these results can serve as input into research on design practices and how to design software is taught.

Keywords:
antipatterns, code smells, systematic mapping, detailed design, design

quality.

Tabla de contenidos

1 Introducción **1**

1.1 Antipatterns y Code smells . . . 3
1.2 Motivación . . . 5
1.3 Objetivos . . . 7
1.4 Trabajo realizado . . . 8
1.5 Publicaciones, proyectos y formación de recursos humanos . . . 9
1.6 Estructura del documento . . . 10

2 Detección de *Antipatterns* y *Code Smells*: Protocolo de Mapeo Sistemático **13**

2.1 Objetivo y preguntas de investigación . . . 13
2.2 Revisión de antecedentes . . . 14
2.3 Herramientas y estrategia de búsqueda . . . 17
2.4 Selección de estudios . . . 18
2.5 Clasificación de estudios y extracción de datos . . . 19

3 Detección de *Antipatterns* y *Code smells*: Ejecución del Mapeo Sistemático **21**

3.1 Ejecución . . . 21
3.1.1 Selección de los estudios . . . 21
3.1.2 Clasificación de estudios . . . 22
3.1.3 Extracción de datos . . . 26
3.2 Resultados . . . 27
3.2.1 *Antipatterns* . . . 27
3.2.2 *Code smells* . . . 36
3.3 Amenazas a la validez . . . 39
3.4 Discusión y conclusiones . . . 41

4 Efectos del diseño sobre la aparición de *code smells* 43
4.1 Contexto . . . 44
4.2 Diseño experimental . . . 46
4.3 Objetivos . . . 47
4.4 Proceso de análisis . . . 48
4.5 Resultados y Discusión . . . 55
4.6 Relación entre los *code smells* y la enseñanza de grado . . . 69
4.7 Amenazas a la validez . . . 71
4.8 Conclusiones . . . 72

5 Conclusiones y Trabajo a Futuro 73
5.1 Conclusiones . . . 73
5.2 Trabajo a futuro . . . 76

Referencias bibliográficas 79

Anexos 91
Anexo 1 Resultados del mapeo sistemático . . . 93
Anexo 2 Catálogo de *antipatterns* . . . 101
Anexo 3 SonarQube. . . . 141
3.1 Herramienta . . . 141
3.1.1 Conceptos principales . . . 142
3.1.2 Principales funcionalidades . . . 145
3.1.3 Arquitectura . . . 146
3.2 Instalación y configuración . . . 148
3.2.1 Análisis de código Java . . . 149
3.2.2 Análisis de código Ruby . . . 151
3.2.3 Análisis de código C/C++ . . . 151
3.2.4 Análisis de código C# . . . 154
Anexo 4 Plantillas de diseño del PSP . . . 157

Capítulo 1

Introducción

La calidad del software tiene distintos significados dependiendo de quién la mire. Los usuarios finales esperan un software de calidad para realizar sus tareas de manera rápida, simple y confiable. Para un administrador del sistema, un software de calidad es fácil de instalar, se adapta a las nuevas configuraciones de hardware y no tiene efectos secundarios con otro software instalado. Para los desarrolladores de software, una cualidad clave, entre otras, es la facilidad de agregar nuevas funcionalidades con menos esfuerzo, tiempo y costo.

El diseño de software es un proceso creativo (Sommerville, 2016), que se compone de dos actividades: el diseño arquitectónico y el diseño detallado. Durante el diseño arquitectónico (a veces llamado diseño de alto nivel) se desarrolla la estructura y la organización de alto nivel del software y se identifican los diversos componentes. Durante el diseño detallado se especifica cada componente con detalles suficientes para facilitar su construcción (Bourque y Fairley, 2014).

Sin embargo, el diseño de software es un proceso complejo y difícil de comprender, donde se deben tomar una serie de decisiones fundamentales, en lugar de aplicar una serie de actividades específicas. Estas decisiones afectan profundamente el software y el proceso de desarrollo (Bourque y Fairley, 2014). Construir un buen diseño requiere un cierto nivel de desarrollo cognitivo que pocos estudiantes de pregrado alcanzan (Carrington y Kim, 2003; Hu, 2013; Linder et al., 2006). Estos desarrollos cognitivos tienen que ver con la capacidad para lograr dimensionar el software, la capacidad de reconocer patrones de diseño, estilos y datos relacionados, así como la capacidad lógica y deductiva para descomponer el sistema en subsistemas y componentes (Hu, 2013).

Las malas prácticas de diseño de software, causadas a menudo por la inexperiencia, el conocimiento insuficiente o la urgencia, originan defectos de diseño, que se definen como problemas en la estructura de un sistema, que no producen errores de compilación o de ejecución, pero que afectan negativamente a los factores de calidad del software (Pérez, 2011). Pueden surgir en diferentes niveles de granularidad, desde problemas de diseño de alto nivel, conocidos como *antipatterns*, hasta problemas de bajo nivel, conocidos como *code smells*. A menudo, estos problemas no están aislados y generalmente son síntomas de defectos más globales (Pérez, 2011).

Los *antipatterns* y los *code smells* documentan defectos comunes cometidos por los desarrolladores al construir sistemas de software, de forma similar a los patrones de diseño de software que documentan las mejores prácticas en el desarrollo de software. Los *antipatterns* describen malas soluciones a problemas de diseño (Brown et al., 1998), mientras que los *code smells* son síntomas de problemas de diseño que se pueden encontrar en el código fuente de un sistema (Fowler, 1999). A menudo su presencia en el código indica que existen problemas de calidad de código, de diseño o de ambos. Los *antipatterns* y los *code smells*, además de documentar el problema, sugieren posibles soluciones. Estas soluciones son normalmente refactorizaciones. El conocimiento de estos defectos de diseño (documentados en *antipatterns* o *code smells*) puede ser utilizado, al igual que los patrones de diseño, para guiar el diseño de software.

Esta tesis busca ordenar el conocimiento existente sobre técnicas y herramientas de detección de *antipatterns* y *code smells*. También busca conocer la calidad interna de los productos de software desarrollados por estudiantes de pregrado de nuestra Facultad, cuando utilizan y cuando no utilizan un conjunto específico de plantillas de soporte a la representación del diseño de software.

El resto del capítulo se estructura de la siguiente forma. En la sección 1.1 se describen los defectos de diseño: *antipatterns* y *code smells*. Luego se presentan, en la sección 1.2 la motivación y en la sección 1.3 los objetivos del trabajo. Los resultados obtenidos a partir del trabajo realizado se describen en la sección 1.4. En la sección 1.5 se presentan las publicaciones y proyectos realizados así como la formación de recursos humanos. Finalmente se presenta el contenido del resto de la tesis en la sección 1.6.

1.1. Antipatterns y Code smells

Debido a que los *antipatterns* han tenido muchos contribuyentes, sería injusto adjudicar la idea original de *antipattern* a una sola fuente (Brown et al., 1998). Más bien, los *antipatterns* son un paso natural para complementar los patrones de diseño y extender su modelo.

La idea de un patrón de diseño se originó con Christopher Alexander, un arquitecto que documentó un lenguaje de patrones para la planificación de ciudades y la construcción de edificios dentro de las ciudades (Alexander et al., 1977). La noción de Alexander de un patrón fue adoptada por varios investigadores de Ingeniería de Software (Beck y Cunningham, 1987; Gamma et al., 2001) y se hizo popular en este campo principalmente después del trabajo publicado por Gamma et al. (1994).

Desde 1994 ha habido un crecimiento importante en la publicación de patrones de diseño, pero también se han documentado construcciones de software que describen malas soluciones a problemas de diseño, muchas veces causadas por la ausencia o mal uso de los patrones de diseño. Una de las primeras presentaciones sobre malas soluciones a nivel de diseño fue en la conferencia *"Object World West"* en 1996 titulada *"AntiPatterns: Vaccinations Against Object Maluse"* (Akroyd, 1996). Michael Akroyd presentó un análisis sobre problemas de diseño que se repetían en varios proyectos de software. La discusión sobre la utilidad de malas soluciones de diseño conocidas o *antipatterns* comenzó casi en paralelo con la introducción de patrones de diseño, donde algunos autores como Webster (1995) presentaron guías basadas en la identificación del comportamiento disfuncional y la refactorización de la solución.

Los *antipatterns* son complementarios a los patrones de diseño y a menudo muestran situaciones en las que los patrones se usan incorrectamente. De hecho, a medida que las tecnologías evolucionan, algunos patrones pueden volverse obsoletos (Dodani, 2006), es decir, lo que alguna vez fue una mejor práctica puede convertirse en una mala práctica en algunos casos.

Los *antipatterns* destacan los problemas más comunes que enfrenta la industria del software y proporcionan información para permitir reconocer estos problemas y determinar sus causas subyacentes. Además, brindan una solución refactorizada que describe cómo cambiar el problema de diseño a una situación más saludable. Por lo tanto, permiten a los desarrolladores identificar problemas de diseño en su sistema y rectificar estos problemas con la solución

correspondiente, proporcionada en la descripción del *antipattern*.

Los *antipatterns*, así como los patrones de diseño, se han documentado en diferentes tipos. Por ejemplo Brown et al. (1998) introdujo un gran número de *antipatterns* de desarrollo orientado a objetos y *antipatterns* arquitectónicos. Asimismo se han documentado *antipatterns* dependiendo de la arquitectura implementada, por ejemplo, orientada a servicios (Jones, 2006) o en capas (Rosenberg y Scott, 1999), mientras que otros se centran en determinados atributos de calidad, como rendimiento (Smith y Williams, 2002) y seguridad (Kis, 2002). Además, hay documentos que describen *antipatterns* específicos de una tecnología, por ejemplo para Java (Tate, 2002), JEE (Java Enterprise Edition) (Tate et al., 2003; Dudney et al., 2002), y otros que especifican *antipatterns* cuando se requiere procesar grandes volúmenes de datos (Bersani et al., 2016).

Similar a los *antipatterns*, Fowler (1999) introdujo la noción de *code smell*. Los *code smells* son síntomas de problemas a nivel de código. Puede que no sean necesariamente un problema (en términos de errores y fallas), pero tienen un impacto negativo en la estructura general del sistema y, en consecuencia, en sus factores de calidad. Al igual que los *antipatterns*, sugieren una posible solución refactorizada.

Los *code smells* pueden desordenar el diseño de un sistema, lo que dificulta su comprensión y mantenimiento. Además, la presencia de *code smells* puede advertir sobre problemas de desarrollo más amplios, como elecciones de diseño incorrectas (Pérez, 2011). Por ejemplo, los desarrolladores pueden agregar varias responsabilidades a una clase, entendiendo que no es necesario incluirlas en clases separadas. El resultado de esto es que la clase crece rápidamente, aumentando las líneas de código. Este síntoma en el código es conocido como el *code smell: Large Class*. Si bien en principio no parece tener consecuencias importantes, cuando las responsabilidades adicionales de la clase crecen y se reproducen, la clase se vuelve demasiado compleja y su calidad se deteriora, generando un problema más grave de diseño. Cuando esta clase se convierte en una clase con muchas responsabilidades, que monopoliza el procesamiento e invoca a otras clases que encapsulan datos, estamos frente a la presencia del *antipattern: "The Blob"*.

1.2. Motivación

En general el software es desarrollado por equipos, necesita mantenerse y cambia a lo largo de su vida (Sommerville, 2016). Desafortunadamente, estos cambios a menudo son realizados con urgencia, debido a las limitaciones del mercado o restricciones de clientes, y por diferentes desarrolladores. Como consecuencia sucede frecuentemente que la calidad del software se descuida con el riesgo de introducir defectos de diseño.

En las últimas dos décadas, los *antipatterns* y *code smells* han sido objeto de estudios empíricos con el objetivo de analizar su impacto en la mantenibilidad del software. Existe evidencia empírica de que el código que contiene estos defectos de diseño es significativamente más propenso a los cambios que el código "limpio" (Khomh et al., 2012). El estudio presentado por Khomh et al. (2009) muestra que las clases que contienen *code smells* presentan una mayor cantidad de cambios que otras clases. Adicionalmente, el código que contiene estos defectos de diseño tiene una mayor tendencia a fallas que el resto del código del sistema (Khomh et al., 2012; Li y Shatnawi, 2007).

Otros estudios exponen que la presencia de un *antipattern* en el código fuente no disminuye el rendimiento de los desarrolladores, mientras que una combinación de estos defectos de diseño da como resultado una disminución significativa de su desempeño (Abbes et al., 2011; Yamashita y Moonen, 2013). Estos estudios indican que la combinación de *antipatterns* impacta negativamente la comprensibilidad del sistema. Además, existe evidencia de que el número de *antipatterns* en los sistemas de software aumenta con el tiempo y solo en algunos casos se eliminan mediante operaciones de refactorización (Arcoverde et al., 2011; Chatzigeorgiou y Manakos, 2010).

Estos resultados sugieren que los *antipatterns* y *code smells* deben ser cuidadosamente detectados y monitoreados y, cuando sea necesario, las operaciones de refactorización deben planificarse y realizarse para abordarlos. Desafortunadamente, la identificación y la corrección de fallas de diseño en sistemas de software de gran porte pueden ser muy complejas. A partir de estas observaciones, sería de utilidad que el ingeniero de software tenga una guía para (i) identificar *antipatterns* y *code smells* y (ii) diseñar y aplicar una solución de refactorización para eliminarlos. Por lo tanto, es muy importante comenzar por conocer cuáles son los *antipatterns* y *code smells* existentes en la literatura y contar con técnicas y herramientas para detectarlos.

Por otro lado, para los estudiantes aprender a diseñar es más difícil que aprender a programar. Uno de los motivos puede deberse a que para la mayoría de los lenguajes de programación, los estudiantes obtienen retroalimentación del compilador y errores en tiempo de ejecución. Sin embargo, esto no ocurre con el diseño (Karasneh et al., 2015).

En un trabajo realizado en la Facultad de Ingeniería de la Universidad de la República, Moreno y Vallespir (2018) estudiaron cuánto esfuerzo dedican los estudiantes al diseño de software y cuál es su percepción respecto al diseño y a la enseñanza de esta disciplina en la carrera de grado en computación. El análisis realizado muestra que los estudiantes dedican al menos cuatro veces más de tiempo a la codificación que al diseño de software. Además, pocos estudiantes se dan cuenta que no dedican el tiempo suficiente al diseño en etapas tempranas del desarrollo.

Otros estudios han examinado las habilidades de diseño que tienen estudiantes de pregrado al evaluar los artefactos producidos por estos durante la etapa de diseño (Chen et al., 2005; Eckerdal et al., 2006a,b; Loftus et al., 2011; Tenenberg, 2005). En general, los autores observan inconsistencia entre los artefactos de diseño y el código, diseños incompletos y falta de entendimiento de qué tipo de información incluir al diseñar software. Estos resultados indican que los estudiantes de pregrado presentan dificultades para diseñar un sistema de software.

Con el objetivo de mejorar la calidad del software, también se han propuesto numerosos procesos de desarrollo del software, como es el *Personal Software Process* (PSP). El PSP es un proceso de desarrollo de software para el individuo, el cual ayuda al ingeniero a controlar, administrar y mejorar su trabajo (Humphrey, 1995). Además de haber propuesto este proceso, Humphrey presentó un curso para que las personas pudieran aprender el PSP e incorporar sus prácticas. Este curso es en formato taller donde los estudiantes (muchas veces ingenieros de software) desarrollan proyectos de software mientras aprenden progresivamente las fases y técnicas del PSP. Luego el curso tuvo sus diferentes variantes, llevadas adelante por el *Software Engineering Institute* (SEI) [1] o por distintas universidades que tomaron el PSP dentro de su oferta de cursos.

Existen varios estudios publicados que muestran una mejora en el rendimiento del desarrollador con la inserción de este proceso. Los datos de los cursos que fueron analizados en estos estudios indican que el PSP mejora la

[1] https://www.sei.cmu.edu

calidad de los productos desarrollados (Hayes y Over, 1997; Rombach et al., 2008; Paulk, 2006; Khan, 2012; Grazioli et al., 2014).

La fase de diseño propuesta en el PSP tiene foco en el diseño orientado a objetos y detallado; por ejemplo: identificar clases, atributos, operaciones, escenarios del programa (o componente), cambios de estado y desarrollar pseudocódigo. Las técnicas de diseño detallado se introducen utilizando cuatro plantillas de diseño. Estas plantillas proporcionan cuatro perspectivas sobre el diseño: una especificación operativa, una especificación funcional, una especificación de estado y una especificación lógica. Las plantillas de diseño detallado del PSP permiten registrar el diseño de forma completa y precisa (Humphrey, 1995).

Por todo esto, es de interés analizar si las plantillas de diseño del PSP evitan o disminuyen la aparición de ciertos defectos de diseño (como *antipatterns* y *code smells*), a medida que estudiantes de pregrado desarrollan un conjunto de proyectos de software, utilizando el marco de medición y las fases definidas del PSP. Conocer qué tipos de defectos de diseño están presentes en el código fuente de los estudiantes, es un insumo en la investigación sobre las prácticas de diseño y cómo se enseña a diseñar software.

1.3. Objetivos

Este trabajo tiene dos objetivos, el primero es contribuir en la organización del conocimiento existente sobre *antipatterns* y *code smells* y las técnicas y herramientas de detección de estos defectos de diseño.

El segundo objetivo es conocer los defectos de diseño en los que incurren los estudiantes de pregrado de nuestra Facultad e investigar, mediante estudios experimentales, si estos pueden ser solucionados utilizando las plantillas de diseño del PSP.

A continuación se presentan las actividades para buscar alcanzar los objetivos:

1. Realizar un mapeo sistemático para conocer el cuerpo de conocimiento sobre las técnicas y herramientas de detección de *antipatterns* y *code smells*.
2. Seleccionar un conjunto de *antipatterns* y *code smells* y herramientas de detección para analizar proyectos desarrollados por estudiantes.

3. Analizar si los estudiantes incurren en el conjunto seleccionado de *antipatterns* y *code smells* durante el desarrollo de los proyectos.
4. Estudiar si existe una diferencia en la calidad interna del software entre un grupo de estudiantes que utiliza un conjunto de plantillas para representar el diseño y un grupo de estudiantes que no utiliza plantillas.
5. Estudiar si existe una diferencia en la calidad interna del software en el grupo de estudiantes que utiliza plantillas para representar el diseño, antes de usar las plantillas y mientras se utilizaron las plantillas.
6. Analizar si los conocimientos impartidos, en cursos obligatorios de la carrera Ingeniería en Computación de la Universidad de la República, ayudan a evitar la aparición de un conjunto específico de defectos de diseño.

1.4. Trabajo realizado

Para conocer el cuerpo de conocimiento sobre las técnicas y herramientas de detección de *antipatterns* y *code smells*, decidimos realizar un mapeo sistemático. Al comienzo del mapeo sistemático realizamos una revisión de antecedentes, para conocer si ya existen otros estudios que respondan algunas de nuestras preguntas de investigación. En esta búsqueda de antecedentes encontramos que el artículo de Fernandes et al. (2016) responde las preguntas de investigación sobre *code smells*, por lo que descartamos de nuestro mapeo sistemático las preguntas de investigación referentes a estos defectos de diseño. Los detalles del mapeo sistemático sobre *antipatterns*, incluyendo la búsqueda de antecedentes, se presenta en el Capítulo 2. La definición y ejecución del mapeo sistemático se basó en las propuestas de Kitchenham et al. (2015) y Petersen et al. (2008). Se conformó un catálogo de *antipatterns* con una ficha para cada uno, incluyendo datos como: nombre, descripción, clasificación, referencias a la bibliografía del *antipattern*, técnicas de detección y características de las herramientas de detección.

Para conocer los defectos de diseño en los que incurren estudiantes de pregrado y para estudiar los efectos de aplicar plantillas de diseño detallado, se analizaron proyectos realizados por estudiantes del curso *Principios y Fundamentos del Personal Software Process*, de la carrera Ingeniería en Computación de la Facultad de Ingeniería, Universidad de la República. Cada estudiante

del curso desarrolla ocho proyectos pequeños siguiendo el proceso definido en el PSP como PSP0 y PSP0.1, que proporciona una introducción al PSP y establece una base inicial de datos históricos de tamaño, tiempo y defectos (Humphrey, 1995).

Todos los estudiantes aplican el proceso definido en los primeros cuatro proyectos. Durante el desarrollo de estos proyectos la fase de diseño no exige la entrega de la representación del diseño. Una vez finalizado el proyecto 4 se divide a los estudiantes en dos grupos. Los estudiantes de uno de los grupos deben representar su diseño utilizando el conjunto de plantillas de diseño definidas en el PSP, a partir del proyecto 5 y hasta el 8.

Los resultados de nuestro trabajo muestran que los estudiantes de pregrado incurren en una gran variedad de defectos de diseño y no mejoran la calidad interna del software cuando utilizan plantillas para representar el diseño. El uso de estas plantillas no evitan ni disminuyen la aparición de ciertos defectos de diseño.

Luego de ejecutar el análisis, realizamos una encuesta a docentes de los cursos obligatorios que son previas del curso *Principios y Fundamentos del Personal Software Process*. El objetivo de esta encuesta es conocer si los conocimientos impartidos en estos cursos obligatorios, ayudarían a evitar la aparición de los defectos de diseño en los que incurrieron los estudiantes durante el desarrollo de los proyectos.

El resultado de la encuesta reveló que la mayoría de los defectos de diseño podrían ser evitados debido al conocimiento que se imparte en algunos de los cursos obligatorios. Sin embargo, los estudiantes incurren en estos defectos de diseño al desarrollar software. Por algún motivo, los estudiantes no logran incorporar estos conocimientos al momento de diseñar y construir software.

1.5. Publicaciones, proyectos y formación de recursos humanos

En el contexto de esta tesis participé como coautora del siguiente artículo aceptado en una conferencia regional:

Silvana Moreno, Vanessa Casella, Martín Solari, Diego Vallespir: La representación del diseño detallado utilizando plantillas y sus efectos en la calidad del software. Conferencia Iberoamericana de Ingeniería de Software, Curitiba, Brasil, 2020.

Mi aporte en el artículo resume parte del trabajo presentado en el Capítulo 4, que analiza los efectos de la representación del diseño sobre la aparición de *code smells* en el código fuente.

Durante la tesis fue aprobado con financiación un proyecto presentado al programa "Iniciación a la Investigación" de la Comisión Sectorial de Investigación Científica (CSIC). Durante la ejecución del proyecto se realizó parte del trabajo central de la tesis.

Título: Efectos del diseño detallado en la calidad del software

Período: 07/2016 - 12/2017

Responsable: Vanessa Casella

Orientador/Tutor: Diego Vallespir

Financiación: CSIC-UdelaR programa Iniciación a la investigación

En el transcurso de la tesis dirigí una Actividad Integradora de la carrera Licenciatura en Computación de la Facultad de Ingeniería, Universidad de la República, sobre análisis estático de código utilizando *SonarQube*.

Título: Análisis estático de código utilizando *SonarQube*

Período: 11/2018 - 09/2019

Estudiante: Gabriel Sánchez

Orientador/Tutor: Diego Vallespir (Supervisor), Vanessa Casella (Supervisor alterno)

1.6. Estructura del documento

El informe de esta tesis cuenta con 4 capítulos además del actual. En el Capítulo 2 *"Detección de Antipatterns y Code Smells: Protocolo de Mapeo Sistemático"* se presenta el protocolo del mapeo sistemático. En el Capítulo 3 *"Detección de Antipatterns y Code smells: Ejecución del Mapeo Sistemático"* se presenta la ejecución y resultados del mapeo sistemático respecto a *antipatterns* y un resumen de la revisión secundaria para *code smells*. En ambos casos, se responden las preguntas de investigación planteadas en el protocolo.

En el capítulo 4 *"Efectos del diseño sobre la aparición de code smells"*, se describe el proceso de análisis para detectar un conjunto de *code smells* en los que incurren los estudiantes en el desarrollo de los proyectos y los efectos del diseño detallado sobre la aparición de estos *code smells*. Para estos pro-

yectos descartamos el análisis de *antipatterns*, dado que estos son de pequeño porte y de baja complejidad. Los datos crudos utilizados en el análisis realizado están publicados en https://www.fing.edu.uy/biblio/datos-crudos-para-el-estudio-de-los-efectos-del-diseño-sobre-la-aparición-de-code-smells. Además, en este capítulo, se presenta la encuesta realizada a docentes, para conocer si los conocimientos impartidos en cursos obligatorios ayudarían a evitar la aparición de los *code smells* en los que incurren los estudiantes.

Por último, el capítulo 5 *"Conclusiones y Trabajo a Futuro"* presenta reflexiones sobre el trabajo realizado y posibles tareas a realizar en futuros trabajos.

El informe contiene también cuatro Anexos. En el Anexo 1 *"Resultados del mapeo sistemático"* se presentan los estudios seleccionados del mapeo sistemático, indicando para cada estudio los campos seleccionados para realizar la extracción de datos. En el Anexo 2 *"Catálogo de antipatterns"*, se presenta el catálogo de *antipatterns* donde cada *antipattern* se expone en una ficha. En el Anexo 3 *"SonarQube"*, se describe esta herramienta y una guía de instalación y configuración para detectar *code smells*. En el Anexo 4 *"Plantillas de diseño del PSP"*, se presentan las plantillas de diseño utilizadas en el curso.

Capítulo 2

Detección de *Antipatterns* y *Code Smells*: Protocolo de Mapeo Sistemático

En este capítulo se describe la versión final del protocolo del mapeo sistemático realizado. Su desarrollo y estructura se basa en las propuestas de Kitchenham et al. (2015) y Petersen et al. (2008).

2.1. Objetivo y preguntas de investigación

Queremos conocer el cuerpo de conocimiento de las técnicas y herramientas de detección de *antipatterns* y *code smells*. En particular los objetivos son:

- Conocer cuáles son las técnicas de detección de *antipatterns* y *code smells*.
- Clasificar las diversas técnicas presentadas y conocer las características de las herramientas que le dan soporte a las técnicas.
- Clasificar e identificar los *antipatterns* detectados por las técnicas o herramientas encontradas en la literatura.
- Identificar los *code smells* detectados por las técnicas o herramientas encontradas en la literatura. No se considera clasificar a los *code smells* dado que no es un aspecto que se presenta en estudios y libros que fueron revisados durante la etapa de conocimiento y entendimiento del tema.

De estos objetivos se desprenden las siguientes preguntas de investigación (RQ):

RQ1: ¿Cuáles son y cómo se clasifican las técnicas de detección de *antipatterns*?

RQ2: ¿Cuáles son y qué características presentan las herramientas utilizadas para detectar *antipatterns*?

RQ3: ¿Cuáles son y cómo se clasifican los *antipatterns* detectados por las técnicas o herramientas?

RQ4: ¿Cuáles son y cómo se clasifican las técnicas de detección de *code smells*?

RQ5: ¿Cuáles son y qué características presentan las herramientas utilizadas para detectar *code smells*?

RQ6: ¿Cuáles son los *code smells* detectados por las técnicas o herramientas?

El objetivo es responder estas preguntas de investigación mediante un mapeo sistemático. Se comienza este mapeo sistemático con una revisión de antecedentes, para conocer si ya existen otras revisiones secundarias de la literatura que respondan algunas de nuestras preguntas de investigación.

2.2. Revisión de antecedentes

Para realizar esta revisión de antecedentes se comenzó definiendo la cadena de búsqueda. Para definirla se combinaron términos que surgen de los objetivos y las preguntas de investigación. El cuadro 2.1 muestra los términos y sinónimos que se obtuvieron del conocimiento del tema y que se fueron ajustando durante búsquedas preliminares en algunos buscadores digitales.

La cadena de búsqueda se conformó con el total de dichos términos y con operadores booleanos (AND-OR):

software AND
(detect OR detection OR detecting) AND
(antipatterns OR “anti-patterns” OR “anti patterns” OR “design defect” OR “anomalie design” OR “design flaw” OR “code smell” OR “code-smell” OR “bad smell”) AND
(tool OR technique OR method)

Para realizar la búsqueda de antecedentes se agregaron a la cadena de

Término	Sinónimo
detect	detection
	detecting
antipatterns	anti-patterns
	anti patterns
	design flaw
	design defect
	anomalie design
code smell	code-smell
	bad smell
technique	method
tool	-

Cuadro 2.1: Términos y sinónimos que forman la cadena de búsqueda

búsqueda propuesta los siguientes términos:

"systematic literature review" OR "systematic review" OR "mapping study" OR "literature review"

Los buscadores digitales seleccionados para realizar esta revisión fueron: Scopus, IEEE Digital Library (IEEEXplore) y ACM Digital Library. Si bien Scopus indexa artículos publicados en IEEE y ACM, no necesariamente son los estudios más recientes de actas de conferencias; además, estos tres buscadores cubren la mayor cantidad de estudios presentados en revistas y actas de conferencias en el área de Ingeniería de Software (Kitchenham et al., 2015).

El acceso a Scopus e IEEEXplore se realizó desde el portal Timbó[1] de la Agencia Nacional de Investigación e Innovación (ANII) de Uruguay. El acceso a ACM Digital Library se realizó directamente desde su portal[2].

Para cada uno de los buscadores seleccionados se hicieron los ajustes necesarios en la cadena de búsqueda. Para los buscadores Scopus y ACM Digital Library las búsquedas se limitaron al título y resumen, mientras que para IEEEXplore se realizó por metadata (incluye título, resumen y palabras clave). Se tomó esta decisión porque IEEEXplore tiene por defecto la opción de filtrar por metadata, siendo más complicado filtrar solo por título y resumen. Esto último implicaba que se generaran más términos de los permitidos por el buscador, teniendo que realizar más de una búsqueda y combinar los resulta-

[1] http://www.timbo.org.uy/
[2] https://dl.acm.org/

dos. En el cuadro 2.2 se detallan las cadenas de búsqueda ajustadas para cada buscador digital.

Fuente	**Cadena de búsqueda**
Scopus	TITLE-ABS ((software AND (detect OR detection OR detecting) AND (antipatterns OR "anti-patterns" OR "design defect" OR "design flaw" OR "anomalie design") AND (tool OR technique OR method)) AND ("systematic literature review" OR "systematic review" OR "mapping study" OR "literature review"))
IEEEXplore	((software AND (detect OR detection OR detecting) AND (antipatterns OR "anti-patterns" OR "design defect" OR "design flaw" OR "anomalie design") AND (tool OR technique OR method)) AND ("systematic literature review" OR "systematic review" OR "mapping study" OR "literature review"))
ACM Digital Library	(recordAbstract:(software) OR acmdlTitle:(software)) AND (recordAbstract:(detect) OR acmdlTitle:(detect) OR recordAbstract:(detection) OR acmdlTitle:(detection) OR recordAbstract:(detecting) OR acmdlTitle:(detecting)) AND (recordAbstract:(antipatterns) OR acmdlTitle:(antipatterns) OR recordAbstract:("anti-patterns") OR acmdlTitle:("anti-patterns") OR recordAbstract:("design defect") OR acmdlTitle:("design defect") OR recordAbstract:("design flaw") OR acmdlTitle:("design flaw") OR recordAbstract:("anomalie design") OR acmdlTitle:("anomalie design")) AND (recordAbstract:(tool) OR acmdlTitle:(tool) OR recordAbstract:(technique) OR acmdlTitle:(technique) OR recordAbstract:(method) OR acmdlTitle:(method)) AND (recordAbstract:("systematic literature review") OR acmdlTitle:("systematic literature review") OR recordAbstract:("systematic review") OR acmdlTitle:("systematic review") OR recordAbstract:("mapping study") OR acmdlTitle:("mapping study") OR recordAbstract:("literature review") OR acmdlTitle:("literature review")

Cuadro 2.2: Revisión de antecedentes: cadena de búsqueda por buscador digital

En la ejecución de la búsqueda de antecedentes se obtuvieron 12 estudios: ocho en Scopus, tres en IEEEXplore y uno en ACM Digital Library, de los cuales tres eran repetidos. Luego de analizar cada uno de los estudios, se encontró que el artículo de Fernandes et al. (2016) respondía las preguntas de investigación RQ4, RQ5 y RQ6. Si bien el estudio define *bad smell* como un conjunto de *code smells* y *design smells*, donde este último término podría corresponderse

con *antipatterns*, no se consideró que responde las tres primeras preguntas de investigación (RQ1, RQ2 y RQ3). Esta conclusión se debe a que los defectos de diseño y herramientas descritas en el artículo hacen referencia a *code smells*. Otro punto a considerar es que la pregunta RQ4 se responde de forma parcial, esto es porque el artículo solo presenta técnicas de detección para las herramientas descritas; a diferencia del objetivo de nuestro mapeo sistemático que pretende conocer técnicas de detección en forma general, independientemente de si se presenta una herramienta que da soporte a la técnica. De todas formas, dado que el artículo expone una gran cantidad de herramientas con su correspondiente técnica de detección, se decidió tomar como válida la respuesta a la pregunta RQ4. Por lo tanto, nuestro mapeo sistemático se restringió a las primeras tres preguntas de investigación relacionadas a *antipatterns*.

2.3. Herramientas y estrategia de búsqueda

Luego de la revisión de antecedentes, la cadena de búsqueda a utilizar en nuestro mapeo sistemático sobre *antipatterns* quedó de la siguiente forma:

software AND
(detect OR detection OR detecting) AND
(antipatterns OR “anti-patterns” OR “anti patterns” OR “design defect” OR “anomalie design” OR “design flaw”) AND
(tool OR technique OR method)

Para realizar este mapeo sistemático se utiliza *Mendeley*[1] como gestor de referencias bibliográficas y la herramienta *StArt* [2] versión 2.3.4.2 como soporte para el desarrollo del protocolo.

La búsqueda se ejecuta sobre los mismos buscadores digitales utilizados para la revisión de antecedentes y se toman en cuenta las mismas consideraciones, es decir, las búsquedas en Scopus y ACM Digital Library se limitaron al título y resumen, mientras que para IEEEXplore la búsqueda se ejecutó por metadata. En el cuadro 2.3 se detallan las cadenas de búsqueda para cada buscador digital.

[1] https://www.mendeley.com
[2] http://lapes.dc.ufscar.br/tools/start_tool

Fuente	Cadena de búsqueda
Scopus	TITLE-ABS (software AND (detect OR detection OR detecting) AND (antipatterns OR “anti-patterns” OR “design defect” OR “design flaw” OR “anomalie design”) AND (tool OR technique OR method))
IEEEXplore	(software AND (detect OR detection OR detecting) AND (antipatterns OR “anti-patterns” OR “design defect” OR “design flaw” OR “anomalie design”) AND (tool OR technique OR method))
ACM Digital Library	(recordAbstract:(software) OR acmdlTitle:(software)) AND (recordAbstract:(detect) OR acmdlTitle:(detect) OR recordAbstract:(detection) OR acmdlTitle:(detection) OR recordAbstract:(detecting) OR acmdlTitle:(detecting)) AND (recordAbstract:(antipatterns) OR acmdlTitle:(antipatterns) OR recordAbstract:(“anti-patterns”) OR acmdlTitle:(“anti-patterns”) OR recordAbstract:(“design defect”) OR acmdlTitle:(“design defect”) OR recordAbstract:(”design flaw”) OR acmdlTitle:(“design flaw”) OR recordAbstract:(“anomalie design”) OR acmdlTitle:(“anomalie design”)) AND (recordAbstract:(tool) OR acmdlTitle:(tool) OR recordAbstract:(technique) OR acmdlTitle:(technique) OR recordAbstract:(method) OR acmdlTitle:(method))

Cuadro 2.3: Revisión de *antipatterns*: cadena de búsqueda por buscador digital

2.4. Selección de estudios

Para seleccionar los estudios primarios se definieron los criterios de inclusión, exclusión y de calidad.

Criterios de inclusión

- Presenta una técnica o herramienta de detección de *antipatterns* en el contexto del software.
- Estudios escritos en inglés.

Criterios de exclusión

- No cumplen con algún criterio de inclusión.
- Estudios no disponibles para ser descargados.

Criterios de calidad

- Los objetivos de la investigación se describen con claridad.

- Presenta al menos un *antipattern* y describe una técnica o herramienta que lo detecta.
- Si un estudio es una actualización o presenta la misma técnica o herramienta que otro estudio, entonces se elige el más reciente.

El proceso de selección de estudios primarios se dividió en tres actividades realizadas exclusivamente por la autora de la tesis. La primera actividad consistió en excluir los estudios duplicados de los estudios devueltos por los buscadores digitales seleccionados. Los estudios resultantes de esta actividad, se tomaron como entrada de la segunda actividad, en la que se aplicaron los criterios de inclusión y exclusión sobre el título y resumen de los estudios, en algunos casos se leyó la introducción y conclusiones para asegurar la inclusión o no del estudio. Los estudios seleccionados de la segunda actividad, se tomaron como entrada de la tercera actividad, en los cuales se aplicaron los criterios de calidad basados en el texto completo.

2.5. Clasificación de estudios y extracción de datos

Los estudios resultantes del proceso de selección se clasificaron en categorías que definen un esquema de clasificación, el cual se detalla en el Capítulo 3. Estas categorías se construyeron a partir de los objetivos de este trabajo. Una primera categoría consiste en el tipo de contribución del estudio, que puede ser una técnica, una herramienta de detección o ambas. Una segunda categoría refiere al tipo de técnica de detección utilizada para detectar *antipatterns*, por ejemplo, técnicas que se basan en heurísticas, en modelos o en aprendizaje automático. Una tercera categoría hace referencia a cómo se pueden clasificar los *antipatterns*, por ejemplo, *antipatterns* que se detectan en desarrollos orientados a objetos, arquitecturas orientadas a servicios, enfocados a problemas de rendimiento o *big data*. La última categoría consiste en las características de la herramienta de detección (cuando el tipo de contribución del estudio es una herramienta), como la disponibilidad, la licencia y el lenguaje de programación capaz de analizar.

Para cada categoría mencionada anteriormente se definieron subcategorías. La definición de las subcategorías de cada categoría se basó en el método de clasificación denominado *Keywording*, propuesto por Petersen et al. (2008), el

cual permite reducir los tiempos para la definición del esquema de clasificación y asegurar que la clasificación contempla los estudios procesados en el mapeo. Este método propone la lectura del resumen, en algunos casos también de la introducción y conclusiones, para buscar palabras clave y conceptos que reflejen la contribución del estudio. Luego de seleccionar un conjunto final de palabras clave y conceptos, se agrupan y se utilizan como forma de categorizar el mapeo.

El proceso de extracción de datos se dividió en dos actividades. La primera actividad consistió en extraer de cada estudio los datos relevantes. Para definir los nombres de las categorías, subcategorías y datos extraídos de los estudios se utilizó también el idioma inglés, por ser el que se va a usar para difundir nuestro trabajo. A continuación se presentan los datos extraídos:

- Año (*Year*), representa el año de publicación del estudio
- Tipo de contribución (*Type of contribution*)
- Técnica de detección (*Detection technique*)
- Clasificación del *antipattern* (*Antipattern classification*)
- Nombre del *antipattern* (*Antipattern's name*)
- Referencia al *antipattern* (*Antipattern's reference*), es decir la referencia que define el *antipattern* por primera vez
- Descripción del *antipattern* (*Antipattern's description*)
- Características principales de la herramienta de detección (*Main features of detection tool*)

La segunda actividad consistió en validar el nombre y la referencia bibliográfica del *antipattern*. La razón por la que se revisó el nombre es porque un *antipattern* puede ser mencionado de distinta manera por los estudios, por lo que se identificaron los *antipatterns* que refieren al mismo problema pero son llamados diferente. Para estos casos se decidió denominar al *antipattern* con el nombre más utilizado por los estudios y además se mencionan los nombres alternativos a este (*"Also know as"*). Para el caso de la referencia bibliográfica del *antipattern*, se corroboró que la referencia citada por el estudio defina al *antipattern*. Para los casos donde un *antipattern* es mencionado con más de un nombre, se registraron todas las referencias bibliográficas encontradas donde se describe ese *antipattern*.

Capítulo 3

Detección de *Antipatterns* y *Code smells*: Ejecución del Mapeo Sistemático

En este capítulo se describe la ejecución y el análisis de los resultados del mapeo sistemático.

3.1. Ejecución

Se ejecutaron las búsquedas en los buscadores digitales definidos y se evaluaron los estudios obtenidos de acuerdo a los criterios establecidos. Finalmente, se extrajo la información relevante de los estudios seleccionados para responder las preguntas de investigación. La ejecución de la búsqueda se realizó en dos oportunidades, la primera el 2 de octubre del 2016 y la segunda el 27 de marzo del 2018, el motivo de la segunda ejecución fue la actualización del mapeo sistemático.

3.1.1. Selección de los estudios

En total se obtuvieron 383 estudios devueltos por los tres buscadores seleccionados. La selección de los estudios siguió el proceso de selección presentado en la sección 2.4. Luego de ejecutar la primera actividad, que excluía los estudios duplicados, quedaron 163 estudios. Los criterios de inclusión y exclusión sobre el título y resumen dejaron como resultado 102 estudios. En la tercera y

última actividad, en la cual se aplicaron los criterios de calidad basados en el texto completo, se seleccionaron 44 estudios primarios.

3.1.2. Clasificación de estudios

A continuación se detalla el esquema de clasificación construido para responder las preguntas de investigación. Este esquema se armó de forma parcial (como ya se presentó en el capítulo anterior) durante la construcción del protocolo del mapeo sistemático. El esquema se completó durante la ejecución del mapeo debido al uso del método de *keywording* para establecer las subcategorías, ya que este método se basa en los artículos seleccionados.

Tipo de contribución (*Type of contribution*)

Esta categoría, describe el tipo de contribución del estudio para detectar *antipatterns*, que se basa en el método de *Keywording* para definir las subcategorías:

- Técnica de detección (*Detection technique*): el estudio describe una técnica de detección.
- Técnica y herramienta de detección (*Detection technique and tool*): el estudio describe una técnica de detección y presenta una herramienta que da soporte a la técnica.
- Herramienta de detección (*Detection tool*): el estudio presenta una herramienta de detección la cual implementa una técnica, pero esta no se describe en el estudio. La información de la técnica se extrae del análisis de la documentación en línea de la herramienta.

Tipo de técnica de detección (*Type of detection technique*)

La definición de las subcategorías para describir el tipo de técnicas de detección tuvo como punto de partida el método de *Keywording*. Luego de este proceso se tuvieron en cuenta otras clasificaciones ya existentes (Ouni et al., 2014; Palomba et al., 2014) y la comprensión, por parte de la autora de la tesis, de cada una de las técnicas para agrupar o crear nuevas subcategorías. De este trabajo realizado surgen las siguientes subcategorías:

Basado en heurística (Heuristic-based)

Uno de los enfoques descritos en Palomba et al. (2014) se basa en la detección de *antipatterns* utilizando algoritmos basados en heurísticas, los cuales analizan diferentes propiedades del sistema, como su estructura, a partir de métricas y reglas. Por ejemplo, para detectar un *antipattern* se puede usar el tipo de heurística estructural, contando las dependencias (tales como las invocaciones) existentes entre un método y una clase. El problema con este tipo de algoritmos está relacionado con la definición de un conjunto de umbrales para identificar un *antipattern*. Para mitigar este problema, se puede realizar una calibración empírica del umbral. Sin embargo, el valor identificado empíricamente podría no ser suficiente para detectar correctamente los *antipatterns* en todos los sistemas de software debido a la heterogeneidad de estos. Por lo tanto, se debe requerir un ajuste específico en cada sistema para mejorar la precisión de detección. Una alternativa a este problema, utilizada por algunos enfoques, es la generación automática de reglas de detección a partir del conocimiento de proyectos previamente inspeccionados, denominados "ejemplos de defectos" (Kessentini et al., 2011) para generar nuevas reglas de detección, basadas en combinaciones de métricas de calidad y valores de umbral. Estas reglas de detección se derivan automáticamente mediante un proceso de optimización (por ejemplo basado en algoritmos GA, del inglés *"Genetic Algorithm"*), que explotan los "ejemplos de defectos" disponibles.

Basado en aprendizaje automático (Machine learning-based)

Otro enfoque descrito en Palomba et al. (2014) se basa en la técnica de aprendizaje automático para predecir módulos defectuosos. El principal beneficio de los enfoques basados en el aprendizaje automático es que no requiere el conocimiento y la interpretación de grandes expertos. Además, logran hasta cierto punto detectar y descubrir defectos potenciales al informar de clases que son similares (incluso no idénticas) a los defectos detectados. Sin embargo, estos enfoques dependen de la calidad y la cantidad de defectos proporcionados para entrenar. De hecho, el alto nivel de falsos positivos representa el principal obstáculo para estos enfoques (Ouni et al., 2014).

Basado en modelo (Model-based)

Este enfoque se basa en la detección de *antipatterns* analizando diferentes propiedades del sistema, pero se diferencia de la técnica "basado en heurística" porque se utiliza un lenguaje de modelado para especificar estas reglas. Las

reglas son generadas a partir de la descripción del *antipattern* en lenguaje natural. Por ejemplo, el estudio Trubiani y Koziolek (2011) describe el lenguaje de modelado denominado *"Palladio Component Model"* (PCM). Otro ejemplo es el presentado por el estudio Bagheri et al. (2015), el cual utiliza el lenguaje *Alloy* para especificar un conjunto de reglas con el objetivo de analizar vulnerabilidades de seguridad en aplicaciones móviles desarrolladas en *Android*.

Basado en lista de verificación (Checklist-based)

Es un enfoque manual que se basa en un conjunto de "técnicas de lectura" que ayudan a un revisor a entender un artefacto de diseño, con el fin de encontrar información relevante (Palomba et al., 2014). Estas técnicas de lectura brindan orientación específica y práctica para identificar defectos de diseño (Torkamani y Bagheri, 2014).

Basado en lógica relacional (Relational logic-based)

Este enfoque se basa en la descripción de *antipatterns* utilizando lógica relacional, para describir tanto aspectos estructurales como de comportamiento. En particular, este enfoque se caracteriza por poder describir aspectos de comportamiento complejos, que son difíciles o imposibles de describir y evaluar mediante el uso de métricas en el código. Por ejemplo, en el estudio de Stoianov y Şora (2010) se utilizan predicados *Prolog* para detectar *antipatterns*.

Características principales de la herramienta de detección (*Main features of detection tool*)

Las características principales de cada herramienta consideradas en este estudio son: disponibilidad, licencia y lenguaje de programación capaz de analizar la herramienta. La construcción de las subcategorías, para cada una de las características presentadas, requirió, además del uso del método de *Keywording*, la búsqueda en línea de cada herramienta propuesta en los estudios catalogados como "Técnica y herramienta de detección" o "Herramienta de detección " de la categoría "Tipo de contribución".

Disponibilidad (*Availability*)

- Herramienta disponible en línea para descargar e instalar (*Tool available online for download and installation*): La herramienta se encuentra disponible en línea para descargar e instalar.

- Herramienta en línea pero no disponible para descargar (*Tool find online but unavailable for download*): La herramienta se encuentra en línea pero no está disponible para descargar.
- Herramienta propuesta en la literatura pero no disponible en línea (*Tool proposed in literature but unavailable online*): La herramienta propuesta en el estudio no se encuentra en línea.

Licencia (*User License*)

Luego de identificar la licencia para cada herramienta encontrada en línea, se clasificaron de la siguiente manera:

- Software gratuito y de código abierto (*Free and open-source software*): se otorga el permiso de forma gratuita a cualquier persona que obtenga una copia de este software y los archivos de documentación asociados. Brinda al consumidor el permiso de forma gratuita a obtener el código fuente original y le otorga la facultad de usarlo, modificarlo y distribuirlo (con o sin modificaciones), cuyas condiciones principales requieren la preservación de los avisos de derechos de autor y licencia. En esta categoría se incluyen licencias específicas como FreeBSD[1].
- Software de código abierto y no gratuito (*Open-source and not free software*): La obtención, uso, copia y/o distribución, ya sea con o sin modificaciones, se realiza mediante pago.
- Software propietario (*Proprietary software*): No se puede acceder al código fuente o tiene un acceso restringido y, por tanto, se ve limitado en sus posibilidades de uso, modificación y redistribución. Cualquier persona que quiera acceder a este tipo de software debe pagar por una licencia y solo puede hacer uso del mismo en un contexto restringido.

Lenguaje de programación (*Programming language*)

Refiere al lenguaje de programación capaz de analizar la herramienta, es decir, el lenguaje en el que tiene que estar implementado el programa para poder ser analizado. Ejemplos de estos lenguajes son: Java, C, C++ y C#.

Clasificación del *antipattern* (*Antipattern* classification)

[1] https://www.freebsd.org/copyright/freebsd-doc-license.html

Los *antipatterns*, así como los patrones de diseño, se han documentado en diferentes tipos, siendo los más populares los *antipatterns* de desarrollo orientado a objetos (*"Object Oriented Design"*). También se han documentado *antipatterns* dependiendo de la arquitectura implementada, por ejemplo orientada a servicios (*"Service Oriented Architecture"*) o en capas (*"Layer Architecture"*), mientras que otros se centran en determinados atributos de calidad, como rendimiento (*"Performance"*) y seguridad (*"Security"*). También hay documentos que describen *antipatterns* específicos de una tecnología y otros que describen *antipatterns* cuando se requiere procesar grandes volúmenes de datos (*"Big Data"*). Las diferentes subcategorías encontradas en los estudios, basado en el método de *Keywording*, son las siguientes:

- Grandes volúmenes de datos (*Big Data*)
- Arquitectura en capas (*Layer Architecture*)
- Diseño orientado a objetos (*Object Oriented Design* (OOD))
- Rendimiento (*Performance*)
- Seguridad en Android (*Security in Android*)
- Arquitectura orientada a servicios (*Service Oriented Architecture* (SOA))

Estas subcategorías no son excluyentes, es decir, un *antipattern* puede clasificarse como "Diseño orientado a objetos" y también como "Rendimiento"; en ambos casos el problema es el mismo pero las consecuencias pueden ser diferentes. Por ejemplo, una clase que implementa demasiadas responsabilidades, en un diseño orientado a objetos puede traer como consecuencia que la clase tenga un alto acoplamiento y una baja cohesión, mientras que si se evalúa desde el punto de vista del rendimiento, puede causar tráfico de mensajes excesivos.

3.1.3. Extracción de datos

La extracción de datos consistió en extraer los datos más relevantes de cada estudio y la validación del nombre y la referencia bibliográfica del *antipattern*, como se menciona en la sección 2.5.

Se observó que muchos *antipatterns* son mencionados por los estudios de diferente manera pero refieren al mismo problema de diseño, en especial los *antipattern* clasificados como "Diseño orientado a objetos".

Por otro lado, las definiciones de *code smells* y *antipatterns* son utilizadas de forma indistinta por algunos autores (Ujhelyi et al., 2015). Luego de eva-

luar cada uno de estos defectos de diseño, se excluyeron del análisis los que describían un *code smell*.

Con respecto a la referencia bibliográfica del *antipattern*, se observó que un gran número de estudios citaban una referencia incorrecta, es decir, cuando se revisaba la referencia que debía describir al *antipattern*, se observaba que este no era mencionado o no se definía en detalle en esa bibliografía. Para estos casos se buscó la bibliografía que definía al *antipattern*, si no se encontraba ninguna referencia se decidió dejar como referencia el estudio que lo detectaba.

Algunos estudios no mencionaban un *antipattern* en particular, sino que describían un conjunto de métricas que eran detectadas por una técnica o herramienta (Biray y Buzluca, 2015; Czibula et al., 2015; Sindhgatta et al., 2009; Tahvildari y Kontogiannis, 2004). Estos estudios fueron descartados porque requería conocer en profundidad la relación entre un *antipattern* y las métricas utilizadas.

3.2. Resultados

Los resultados se dividen en dos secciones, la sección *Antipatterns* muestra los resultados de nuestro mapeo sistemático, mientras que la sección *Code smells* muestra un resumen de los resultados obtenidos del estudio secundario de Fernandes et al. (2016).

3.2.1. *Antipatterns*

Los resultados del estudio de mapeo se presentan a continuación respondiendo las preguntas de investigación RQ1, RQ2 y RQ3.

Por otro lado, en el Anexo 1 se presentan todos los estudios que fueron seleccionados para este mapeo. Para cada estudio se presenta el año de publicación, las bibliotecas digitales en las cuales fue encontrado el estudio, la referencia bibliográfica, el tipo de contribución del estudio, el tipo de técnica, la clasificación del *antipattern* que se detecta y los *antipatterns* detectados. Si el tipo de contribución presenta una herramienta, se presentan las características principales de esta: nombre, licencia, disponibilidad y lenguaje de programación.

RQ1: ¿Cómo se clasifican las técnicas de detección de *antipatterns*?

Para responder a esta pregunta, se clasifica cada estudio según el tipo de técnica de detección. El resultado fue que más de la mitad de los estudios se basan en la técnica de detección "Basado en heurística" (55 %) para detectar *antipatterns*, le sigue la técnica "Basado en aprendizaje automático" (20 %) y "Basado en modelo" (18 %), siendo las menos populares dentro de los estudios las técnicas "Basado en lógica relacional" (5 %) y "Basado en lista de verificación" (2 %).

El cuadro 3.1 clasifica los estudios seleccionados por tipo de técnica y tipo de contribución del estudio. Se puede observar que para el tipo de técnica clasificada como "Basado en heurística", la cantidad de estudios que describen una técnica y presentan una herramienta de soporte a la técnica, es superior a los que solo presentan una técnica. Sin embargo, para el tipo de técnica catalogada como "Basado en aprendizaje automático", la cantidad de estudios que presentan una técnica y la herramienta, es significativamente menor que los estudios que presentan solo una técnica. Esto puede deberse a que desarrollar una herramienta para este tipo de técnica presenta una mayor complejidad. Para las técnicas clasificadas como "Basado en modelo" y "Basado en lógica relacional", el porcentaje de los estudios que presentan una técnica con los que presentan una técnica y una herramienta son iguales.

El cuadro 3.2 clasifica los estudios por técnica de detección y año de publicación del estudio. Se puede observar que el interés en esta área es reciente y que la investigación en el tipo de técnica "Basado en heurística" se ha mantenido a lo largo del tiempo. Las técnicas "Basado en aprendizaje automático" y "Basado en modelo" también han sido estudiadas a lo largo de los años pero en menor medida. Además, se percibe que entre los años 2010 y 2015 se publicaron la mayor cantidad de estudios. Para el caso particular del año 2018, la baja en la cantidad de estudios publicados puede deberse a la fecha de realización del mapeo y a que en esa fecha hay algunas conferencias que aún no estaban indexadas por las bibliotecas digitales.

RQ2: ¿Qué características presentan las herramientas utilizadas para detectar *antipatterns?*

Para responder a la pregunta, se presentan en el cuadro 3.3 las características de cada herramienta, clasificadas según su disponibilidad, licencia y

Tipo de técnica de detección	Tipo de contribución	Referencias
Basado en heurística	Técnica de detección	Ouni et al. (2017b), Wang et al. (2017), Aras y Selçuk (2016), Ouni et al. (2014), Palma et al. (2013), Kessentini et al. (2011), Moha et al. (2010), Rayside y Mendel (2007), Salehie et al. (2006)
	Técnica y herramienta de detección	Trubiani et al. (2018), Nahar y Sakib (2016), Peiris y Hill (2016), Peldszus et al. (2016), Nahar y Sakib (2015), Kaur y Kaur (2014), Sharma y Anwer (2014), Tekin y Buzluca (2014), Nayrolles et al. (2013), Erdemir et al. (2011), Crasso et al. (2009), Brühlmann et al. (2008), Marinescu (2005), Trifu et al. (2004)
	Herramienta de detección	Fontana et al. (2016)
Basado en aprendizaje automático	Técnica de detección	Ouni et al. (2017a), Hecht et al. (2015), Peiris y Hill (2014), Maiga et al. (2012a), Pulawski (2012), Budi et al. (2011), Hassaine et al. (2010), Vaucher et al. (2009)
	Técnica y herramienta de detección	Kreimer (2005)
Basado en modelo	Técnica de detección	Arcelli et al. (2018), Arcelli et al. (2015), Ganea y Marinescu (2015), Cortellessa et al. (2009)
	Técnica y herramienta de detección	De Sanctis et al. (2017), Bersani et al. (2016), Sfayhi y Sahraoui (2011), Trubiani y Koziolek (2011)
Basado en lista de verificación	Técnica de detección	Torkamani y Bagheri (2014)
Basado en lógica relacional	Técnica de detección	Stoianov y Şora (2010)
	Técnica y herramienta de detección	Bagheri et al. (2015)

Cuadro 3.1: Clasificación de estudios por técnica de detección y contribución del estudio

lenguaje de programación. El resultado fue que de un total de 23 herramientas presentadas por los estudios, el 43,5 % están disponibles en línea para descargar e instalar. El 39,1 % son herramientas propuestas para probar la técnica

Año	Tipo de contribución					Total
	Basado en heurística	**Basado en aprendizaje automático**	**Basado en modelo**	**Basado en lista de verificación**	**Basado en lógica relacional**	
2004	1	0	0	0	0	1
2005	1	1	0	0	0	2
2006	1	0	0	0	0	1
2007	1	0	0	0	0	1
2008	1	0	0	0	0	1
2009	1	1	1	0	0	3
2010	1	1	0	0	1	3
2011	2	1	2	0	0	5
2012	0	2	0	0	0	2
2013	2	0	0	0	0	2
2014	5	1	0	1	0	6
2015	1	1	2	0	1	5
2016	5	0	1	0	0	6
2017	1	1	1	0	0	3
2018	1	0	1	0	0	2
Total	24	9	8	1	2	44

Cuadro 3.2: Estudios agrupados por técnica de detección y año de publicación

presentada, las cuales no se encontraron en línea, es decir, el estudio no presentaba referencia a la herramienta o esta ya no se encontraba disponible. Para el 17,4 % de las herramientas presentadas, se encontró material en línea pero la herramienta no estaba disponible para descargar.

La mayoría (72,7 %) de las herramientas que están en línea para descargar e instalar presentan una licencia de software libre y gratuita. La herramienta *SonarQube*, muy utilizada en la industria, presenta dos tipos de licencias: software libre gratuito y software libre no gratuito; la diferencia entre las versiones está en los lenguajes de detección y algunos servicios extras como: análisis de ramas y *pull requests*, informes ejecutivos, soporte técnico experto, entre otros.

A pesar de que el tipo de técnica de detección más estudiada (55 %) por los estudios es "Basado en heurística", solo se presentan dos herramientas para descargar e instalar, frente a tres herramientas que utilizan técnicas clasificadas como "Basado en modelo" (18 %).

Independientemente de la disponibilidad y licencia utilizada, la mayoría de las herramientas detectan *antipatterns* en programas implementados en el lenguaje de programación Java.

Disponibilidad	Licencia	Nombre	Lenguaje de programación	Tipo de técnica de detección	Referencia
Herramienta disponible en línea para descargar e instalar	Software gratuito y de código abierto	Alloy Analyzer [1]	Alloy	Basado en lógica relacional	Bagheri et al. (2015)
		Extension PCM Bench [2]	NA	Basado en modelo	Trubiani y Koziolek (2011)
		HULK [3]	Java	Basado en heurística	Peldszus et al. (2016)
		OSTIA (On-the-fly Static Topology Inference Analysis) [4]	Java	Basado en modelo	Bersani et al. (2016)
		panda-aemilia [5]	NA	Basado en modelo	De Sanctis et al. (2017)
		PADprof [6]	Java	Basado en heurística	Trubiani et al. (2018)
		SOMAD (Service Oriented Mining for Antipattern Detection) [7]	Java	Basado en heurística	Nayrolles et al. (2013)
		SonarQube [8]	Más de 20 lenguajes de programación, incluidos C, C++, Java,	Basado en heurística	Fontana et al. (2016)
	Software de código abierto y no gratuito	SonarQube	Más de 20 lenguajes de programación, incluidos C, C++, Java, C#, Ruby	Basado en heurística	Fontana et al. (2016)
	Software propietario	inFusion [9]	Java, C, C++	Basado en heurística	Fontana et al. (2016)

Continua en la siguiente página

[1] http://alloytools.orgA
[2] https://sdqweb.ipd.kit.edu/wiki/PerOpteryx
[3] https://github.com/Echtzeitsysteme/hulk-ase-2016
[4] https://github.com/maelstromdat/OSTIA
[5] https://github.com/CatiaTrubiani/panda-aemilia
[6] https://github.com/diagnoseIT/padprof
[7] https://github.com/MathieuNls/somad
[8] http://www.sonarqube.org
[9] http://www.intooitus.com/products/infusion

Cuadro 3.3 – *Continuación de la página anterior*

Disponibilidad	**Licencia**	**Nombre**	**Lenguaje de programación**	**Tipo de técnica de detección**	**Referencia**
		Structure101 [1]	Java, .NET	Basado en heurística	Fontana et al. (2016)
Herramienta en línea pero no disponible para descargar	Software gratuito y de código abierto	ADPR (Antipattern based Design Pattern Recommender)	Java	Basado en heurística	Nahar y Sakib (2015)
		E-Quality	Java	Basado en heurística	Tekin y Buzluca (2014), Erdemir et al. (2011)
		SA4J (Structural Analysis for Java)	Java	Basado en heurística	Fontana et al. (2016)
	Software propietario	ProDeOOS	Java, C++	Basado en heurística	Marinescu (2005)
Herramienta propuesta en la literatura pero no disponible en línea	-	ACDPR (Antipattern based Creational Design Pattern Recommender)	Java	Basado en heurística	Nahar y Sakib (2016)
		Antipattern Testing Tool	Java	Basado en heurística	Kaur y Kaur (2014)
		EMAD (Excessive Memory Detector de asignación)	C, C++	Basado en heurística	Peiris y Hill (2016)
		IYC ("It's Your Code")	Java	Basado en aprendizaje automático	Kreimer (2005)
		JEETuning Expert	Java	Basado en heurística	Crasso et al. (2009)
		jGoose	Java	Basado en heurística	Trifu et al. (2004)
		Metanool	Java	Basado en heurística	Brühlmann et al. (2008)

Continua en la siguiente página

[1] http://structure101.com/products/

Cuadro 3.3 – *Continuación de la página anterior*

Disponibilidad	Licencia	Nombre	Lenguaje de programación	Tipo de técnica de detección	Referencia
		PAD (Performance Antipattern Detector)	Java	Basado en heurística	Sharma y Anwer (2014)
		Verso	NA	Basado en modelo	Sfayhi y Sahraoui (2011)

Cuadro 3.3: Clasificación de herramientas por disponibilidad y licencia

RQ3: ¿Cómo se clasifican y cuáles son los *antipatterns* presentados en los estudios?

Para responder a esta pregunta se construyó una ficha para cada *antipattern*, como se muestra en la figura 3.1, de los detectados por las técnicas o herramientas presentadas en el mapeo sistemático. El objetivo es conformar un catálogo de *antipatterns* incluyendo las técnicas y herramientas que logran detectarlos. La ficha se presenta en inglés, por ser el idioma a utilizar para difundir este catálogo:

En la figura 3.2 se presenta un ejemplo del uso de la ficha para el *antipattern "The Blob"*:

El catálogo completo de *antipatterns* se presenta en el Anexo 2. Este catálogo está conformado por 75 *antipatterns*, de los cuales 49 son detectados por al menos una herramienta y solo 12 por dos o más herramientas. De los 49 *antipatterns* detectados por alguna herramienta, 23 se clasifican como "Diseño orientado a objetos" y 15 como "Rendimiento", de los cuales dos *antipatterns* están en ambas categorías (*"The Blob"* y *"Tangle"*). Para el resto de las categorías, la cantidad de *antipatterns* es menor, presentándose seis clasificados como "Arquitectura orientada a servicios", cuatro como "Grandes volúmenes de datos" y tres como "Seguridad en Android". Para los *antipatterns* detectados como "Arquitectura en capas" no se presentó ninguna herramienta. Del total de *antipatterns* presentados, solo 25 son detectados por herramientas disponibles en línea para descargar e instalar y que presentan licencia de software libre y gratuita.

El cuadro 3.4 muestra la cantidad de *antipatterns* y estudios por clasificación del *antipattern*, donde dos *antipatterns* pertenecen a dos categorías

Name nombre más utilizado por los estudios para llamar al *antipattern*

Description breve descripción del *antipattern*

References referencias a la bibliografía del *antipattern*

Classification clasificación definida en 3.1.2 del *antipattern*. Las categorías son: *Big Data*, *Layer Architecture*, *Object Oriented Design*, *Performance*, *Security in Android*, *Service Oriented Architecture*

Also know as nombres alternativos utilizados para el *antipattern*

References to techniques referencias a la bibliografía de los estudios que presentan una técnica que detecta al *antipattern*

Type of detection techniques tipo de técnica definida en 3.1.2 para detectar al *antipattern*. Las categorías son: *Heuristic-based*, *Machine learning-based*, *Model-based*, *Checklist-based*, *Relational logic-based*

Name of the tools nombres de las herramientas que detectan al *antipattern*

User Licenses licencias de las herramientas categorizadas en 3.1.2. Las categorías son: *Tool available online for download and installation*, *Tool find online but unavailable for download*, *Tool proposed in literature but unavailable online*

Programming languages lenguajes de programación capaz de analizar las herramientas para detectar al *antipattern*

Figura 3.1: Ficha del *antipattern*

diferentes, "Diseño orientado a objetos" y "Rendimiento" respectivamente. De un total de 44 estudios, se observa que más de la mitad presentan *antipatterns* clasificados como "Diseño orientado a objetos", mientras que solo tres estudios, uno en cada categoría, presentan *antipatterns* como "Seguridad en Android", "Arquitectura en capas" y "Grandes volúmenes de datos". Por otra parte, si bien solo seis estudios se enfocan en *antipatterns* clasificados como "Arquitectura orientada a servicios", estos hacen referencia a más del 25 % del total de *antipatterns* detectados. Este resultado se debe a que algunos estudios presentan técnicas o herramientas de detección para diversos *antipatterns*. Caso contrario sucede con los *antipatterns* categorizados como "Diseño orientado a objetos", donde distintos estudios presentan la detección de un mismo *antipattern*, como por ejemplo *"The Blob"*.

De los 11 *antipatterns* clasificados como "Rendimiento", tres de ellos (*"Multiple EJB accesses per request"*, *"Redundant JNDI lookups"* y *"Round tripping"*) son específicos de la tecnología JEE (Java Enterprise Edition) (Tate et al., 2003; Dudney et al., 2002), mientras que los cuatro *antipatterns* clasi-

Name The Blob

Description The Blob is found in designs where one class monopolizes the processing, and other classes primarily encapsulate data. This antiPattern is characterized by a class diagram composed of a single complex controller class surrounded by simple data classes. The key problem here is that the majority of the responsibilities are allocated to a single class. In general, The Blob is a procedural design even though it may be represented using object notations and implemented in object oriented languages. A procedural design separates process from data, whereas an object oriented design merges process and data models, along with partitions. The Blob contains the majority of the process, and the other objects contain the data. Architectures with The Blob have separated process from data; in other words, they are procedural style rather than object oriented architectures.
From a performance perspective, this antipattern creates problems bycausing excessive message traffic. In the behavioral form of the problem, the excessive traffic occurs as the "god" class requests and updates the data it needs to control the system from subordinate classes. In the data form, the problem is reversed as subordinates request and update data in the "god" class. In both cases, the number of messages required to perform a function is larger than it would be in a design that assigned related data and behavior to thesame class.

References Palomba et al. (2014), Lanza y Marinescu (2006), Smith y Williams (2000), Brown et al. (1998), Akroyd (1996), Riel (1996)

Classification Object Oriented Design, Performance

Also know as God Class, Big Class, Winnebago, God Object, Fat Class, Blob Controller, Blob Data-Container

References to techniques Arcelli et al. (2018), De Sanctis et al. (2017), Aras y Selçuk (2016), Fontana et al. (2016), Peldszus et al. (2016), Arcelli et al. (2015), Ganea y Marinescu (2015), Hecht et al. (2015), Kaur y Kaur (2014), Ouni et al. (2014), Sharma y Anwer (2014), Maiga et al. (2012a), Pulawski (2012), Erdemir et al. (2011), Kessentini et al. (2011), Sfayhi y Sahraoui (2011), Trubiani y Koziolek (2011), Hassaine et al. (2010), Moha et al. (2010), Stoianov y Şora (2010), Cortellessa et al. (2009), Vaucher et al. (2009), Brühlmann et al. (2008), Salehie et al. (2006), Kreimer (2005), Marinescu (2005), Trifu et al. (2004)

Type of detection techniques Heuristic-based, Machine learning-based, Model-based, Relational logic-based

Name of the tools ProDeOOS, IYC, Metanool, Antipattern Testing Tool, Verso, HULK, jGoose, E-quality, PAD, Extension PCM Bench, panda-aemilia, Structure101

User Licenses Free and open-source software, Proprietary software

Programming languages Java, C++, .NET

Figura 3.2: Ficha del *antipattern "The Blob"*

ficados como "Grandes volúmenes de datos" son específicos de la tecnología *"Storm"* (Toshniwal et al., 2014), desarrollada por *"Twitter"* para enfrentar el problema del procesamiento de la transmisión de datos.

Clasificación del *antipattern*	Estudios	Cantidad de *antipatterns*
Grandes volúmenes de datos	1	4
Arquitectura en capas	1	2
Diseño orientado a objetos	24	28
Rendimiento	11	20
Seguridad en Android	1	3
Arquitectura orientada a servicios	6	20

Cuadro 3.4: Cantidad de estudios y *antipatterns* por clasificación del *antipattern*

3.2.2. *Code smells*

El cuadro 3.5 presenta un resumen del estudio de Fernandes et al. (2016) que responde las tres preguntas de investigación RQ4, RQ5 y RQ6. Además, a este resultado, se le suma la herramienta *SonarQube*, presentada en el artículo Fontana et al. (2016) y evaluada en la sección de *antipattern*, que también detecta *code smells*.

Tool Name	Plug-in	Detected Bad Smells	Language		Detection Technique	Free for Use	Guide	GUI	Release Year
			Developed	Detect					
Borland Together	Yes	Duplicated Code	Java	C#, C++, Java	Metrics	No	Yes	Yes	2011
CCFinder (CCFinderX)	No	Duplicated Code	C++	C, C#, C++, etc.	Token	Yes	Yes	Yes	2002
Checkstyle	Yes	Duplicated Code, Large Class, Long Method, Long Parameter List	Java	Java	NA	Yes	Yes	Yes	2001
Clone Digger	NA	Duplicated Code	Python	Java, Lua, Python	Tree	Yes	Yes	Yes	2008
Code Bad Smell Detector	No	Data Clumps, Switch Statements, and 3 other	Java	Java	AST	Yes	No	No	2014
Colligens	Yes	NA	C	C	NA	Yes	Yes	Yes	2014
ConcernReCS	Yes	Concern Smells: Primitive Concern Constant, and 5 other	Java	Java	Concern map	Yes	Yes	Yes	2012
ConQAT	Both	Clone Code	Java	ABAP, ADA, C++, C#, Java	Metrics	No	Yes	Yes	2005
DECKARD	No	Clone Code	C	Java	AST	Yes	Yes	No	2007
DuDe	No	Clone Code	Java	Language independent	Textual analysis	Yes	No	Yes	2005
Gendarme	No	Duplicated Code, Large Class, Long Method, and 4 other	C#	.NET, Mono	Rules	Yes	Yes	Yes	2006
inCode	Yes	Data Class, Data Clumps, Duplicated Code, and 2 other	Java	C, C++, Java	NA	No	Yes	Yes	2013
inFusion	No	Data Class, Data Clumps, Duplicated Code, and 2 other	NA	C, C++, Java	NA	No	Yes	Yes	2011
IntelliJ IDEA	No	Data Clumps, Feature Envy, Large Class, and 4 other	NA	Java, JavaScript, and 4 others	NA	No	Yes	Yes	2001
iPlasma	No	Duplicated Code, Feature Envy, Intensive Coupling, and 4 other	Java	C++, Java	Textual analysis	Yes	Yes	Yes	2005
Java Clone Detector	No	Duplicated Code	C++	Java	Tree	Yes	Yes	No	2009
jCosmo	No	InstanceOf, Switch Statement, Typecast	NA	Java	Tree	NA	Yes	Yes	2002
JDeodorant	Yes	Feature Envy, Large Class, Long Method	Java	Java	Metrics	Yes	Yes	Yes	2007
JSpIRIT	Both	Data Class, Dispersed Coupling, Feature Envy, and 5 other	Java	C++, Java, Smalltalk	Metrics	Yes	Yes	Yes	2014
NiCad	Yes	Duplicated Code	C	C, C#, Java, etc.	NA	Yes	Yes	Yes	2011
NosePrints	Both	Feature Envy, Inappropriate Intimacy, Large Class, and 5 other	NA	NA	NA	No	No	Yes	2008
PMD	Both	Duplicated Code, Large Class, Long Method, Long Parameter List	Java	C, C#, C++, Java, PHP, and 11 other	NA	Yes	Yes	Yes	2008
PoSDef	Yes	NA	C#	UML diagrams	Metrics	Yes	No	Yes	2014
SDMetrics	No	Large Class	Java	UML diagrams	NA	No	Yes	Yes	2012
Stench Blossom	Yes	Comments, Data Clumps, and 4 other	Java	Java	Metrics	Yes	Yes	Yes	2010
SYMake	No	Cyclic Dependency, Duplicated Prerequisites	NA	C and Java	NA	Yes	Yes	Yes	2012
TrueRefactor	No	Lazy Class, Long Method, and 3 other	Java	Java	Graph	Yes	Yes	Yes	2011
Understand	No	NA	NA	C, C#, C++, etc.	NA	No	Yes	Yes	2008
Wrangler	Yes	Duplicated Code	Erlang	Erlang	Textual analysis	Yes	Yes	Yes	2010

Cuadro 3.5: Lista de herramientas de detección de *code smells* disponibles para descargar e instalar

A continuación, se contestan las preguntas de investigación. Como el estudio presenta herramientas con su correspondiente técnica de detección, se van a responder las preguntas RQ4 y RQ5 en forma simultánea.

RQ4 y RQ5: ¿Cómo se clasifican las técnicas de detección de *code smells*? ¿Qué características presentan las herramientas utilizadas para detectar *code smells*?

La revisión de la literatura presentó 84 herramientas de detección de *code smells* desde el año 1993 al 2015. El mayor número de herramientas de detección fueron liberadas entre el 2005 y 2014, con una tendencia creciente en el número de herramientas liberadas, con un pico en 2012 con 16 herramientas. Además, el mayor número de herramientas propuestas se encuentra entre el 2012 y 2014, aunque en 2013 solo se lanzó una herramienta. Sin embargo, de las 84 herramientas, solo el 34,5 % están disponibles para descargar e instalar.

En los resultados presentados, se puede observar que más de la mitad de las herramientas no describen la técnica de detección utilizada (valor *"NA"* en en columna *"Detection Technique"*). Las técnicas *Metrics*, *Tree*, *Abstract Syntax Tree (AST)*, *Concern map* y *Graph* se basan en analizar diferentes propiedades del sistema, por ejemplo su estructura, a partir de métricas y reglas. Para analizar estas métricas, se utilizan distintas representaciones de datos, como listas, árboles, grafos, y luego se aplican distintos algoritmos. Estas técnicas se pueden clasificar como el tipo de técnica "Basado en heurística", según la clasificación presentada en 2.5. Las herramientas, en su gran mayoría, detectan *code smells* en programas implementados en Java, C, C++ y C#, siendo Java el lenguaje de detección más utilizado.

RQ6: ¿Cuáles son los *code smells* presentados en los estudios?

La revisión sistemática encontró un total de 61 *code smells* (Fowler, 1999; Bavota et al., 2015; Khomh et al., 2011; Maiga et al., 2012b; Tourwe y Mens, 2003; Vidal et al., 2016) detectados por las herramientas. Los 10 *code smells* más frecuentes que las herramientas detectan son: *"Duplicated Code"*, *"Large Class"*, *"Long Method"*, *"Feature Envy"*, *"Data Class"*, *"Long Parameter List"*, *"Lazy Class"*, *"Message Chain"*, *"Shotgun Surgery"* y *"Switch Statement"*. Se observó que todos estos *code smells* se describen en Fowler (1999).

La herramienta *SonarQube*, además de detectar *code smells* como: *"Du-*

plicated Code", *"Large Class"* y *"Long Method"*, revisa el incumplimiento de estándares, falta de pruebas unitarias, mala distribución de la complejidad, entre otros. La mayoría de los *code smells* presentados, dependen del lenguaje de programación en el que está implementado el programa.

3.3. Amenazas a la validez

Los conceptos de limitaciones sobre el rigor de un estudio empírico, expresados como amenazas de validez, están bien establecidos para los estudios primarios. Sin embargo, el concepto también se aplica a los estudios secundarios, y de hecho, a menudo se discuten al informar una revisión sistemática (Kitchenham et al., 2015). A continuación se presentan las amenazas a la validez en el contexto de nuestro mapeo sistemático.

Validez del constructo

La validez del constructo asegura que el diseño del estudio puede responder correctamente a las preguntas de investigación planteadas (Kitchenham et al., 2015). Para obtener un conjunto de estudios primarios tan completo como sea posible sobre el tema de investigación, se seleccionaron tres buscadores digitales que cubren la mayor cantidad de estudios presentados en revistas y actas de conferencias en el área de Ingeniería de Software (Kitchenham et al., 2015). Con respecto a la cadena de búsqueda, esta se formó a partir de términos y sinónimos que se obtuvieron del conocimiento del tema y que se fueron ajustando durante búsquedas preliminares. Sin embargo, la lectura completa de los estudios mostró algunos términos que se podrían haber utilizado como: *"approach"* y *"design anomaly"*. Otra amenaza puede deberse al sesgo en el proceso de selección de estudios primarios. Esta amenaza se disminuye con la construcción del protocolo de investigación. Una tercera amenaza a la validez del constructo es que en el mapeo sistemático no se consideró incluir literatura gris (Garousi et al., 2019). En este caso, se pueden estar "escapando" herramientas de detección de *antipatterns* que se publicaron en algún *blog* o publicación menos formal dirigida a desarrolladores de software. Esta última amenaza a la validez se puede evitar realizando, como trabajo a futuro, una revisión multivocal de la literatura.

Validez interna

La validez interna refiere a la realización del estudio, en particular relacionado a la extracción de datos y síntesis, y si hay factores que podrían haber causado algún grado de sesgo en el proceso general (Kitchenham et al., 2015). Para disminuir esta amenaza, se definió un proceso de selección de estudios, que incluyó criterios de inclusión, exclusión y de calidad sobre cada estudio. En particular, los criterios de inclusión y exclusión que se aplicaban sobre el título y resumen de los estudios, se realizaron en algunos casos sobre la introducción y conclusiones para asegurar la inclusión o no del estudio. Se realizó un formulario de extracción de datos y esquema de clasificación, para asegurar que los datos sean consistentes. Sin embargo, el proceso de selección, la extracción de datos y el esquema de clasificación fue realizado solo por la autora de la tesis, lo cual es una amenaza a la validez interna.

Validez de conclusión

La validez de conclusión refiere a la relación entre la síntesis de la revisión secundaria y qué tan bien esto respalda la conclusión de la revisión (Kitchenham et al., 2015). Para presentar resultados que sean razonables y puedan ser reproducidos por otros grupos de investigación, se presenta información detallada sobre los término de la cadena de búsqueda y los buscadores digitales utilizados. Además, para disminuir esta amenaza se desarrolló un formulario de extracción de datos que asegura que los datos relevantes extraídos sean consistentes. También, las clasificaciones del mapeo se realizaron a través del método de *Keywording* (Petersen et al., 2008). Para definir las categorías de las técnicas de detección, también se utilizaron algunas clasificaciones existentes. Sin embargo, la síntesis del mapeo sistemático fue realizada solo por la autora de la tesis.

Validez externa

La validez externa refiere a la posibilidad de generalización de los resultados (Kitchenham et al., 2015). Para poder generalizar los resultados y así disminuir esta amenaza, no se limitaron las fechas de búsqueda. Por otra parte, los resultados de este estudio se consideraron en el contexto del software. Por lo tanto, la clasificación presentada y las conclusiones extraídas solo son válidas en este contexto. Otra amenaza identificada fue que algunos estudios (Kessentini et al., 2011; Erdemir et al., 2011; Brühlmann et al., 2008; Salehie et al., 2006; Moha et al., 2010; Kaur y Kaur, 2014; Cortellessa et al., 2009;

Trifu et al., 2004) mencionan que las técnicas o herramientas utilizadas detectaban más *antipatterns* que los presentados; como consecuencia, existen más *antipatterns* que pueden ser detectados y que no se presentan en nuestro estudio de mapeo. Cabe acotar también que el no haber utilizado literatura gris, como se presentó en el análisis de la validez del constructo, también afecta a la generalización y por ende a la validez externa.

3.4. Discusión y conclusiones

El mapeo sistemático se realizó para enriquecer el cuerpo de conocimiento sobre las técnicas y herramientas de detección de *antipatterns* y *code smells* y también para obtener un listado de estos defectos de diseño que se mencionan en la literatura. En la búsqueda de antecedentes, se encontró una revisión secundaria para *code smells*, pero no se encontró ninguna para *antipatterns*. Por esta razón, el mapeo sistemático se ejecutó solo para encontrar técnicas y herramientas de detección de *antipatterns* y para conocer los *antipatterns* más utilizados en la literatura.

Los resultados obtenidos relacionados a *antipatterns* y la construcción de una ficha para cada *antipattern* detectado, que conforman un catálogo de estos defectos de diseño (ver catálogo en el Anexo 2), son importantes no solo para el desarrollo del proyecto sino que es un aporte a la comunidad. El catálogo realizado es un insumo para otras investigaciones y proporciona a los profesionales una visión general de los *antipatterns* y su detección. Sin embargo, muchos estudios no mencionaban todos los *antipatterns* detectados por la técnica o herramienta presentada, por lo que pueden existir más *antipatterns* que no se presentan en este catálogo.

Respecto al mapeo sistemático realizado, se observó que existen muchos *antipatterns* mencionados con distintos nombres pero refieren al mismo problema. Esto puede deberse a que el estudio sobre este tipo de defecto de diseño es bastante reciente y no existe una nomenclatura unificada para hacer referencia a un *antipattern*.

La existencia de *antipatterns* definidos en la literatura de los que no se encontraron técnicas o herramientas que lo detecten, hace pensar que además de que la investigación sobre este tema es inmadura, es difícil de abordar con enfoques automáticos, debido a la complejidad de estos defectos de diseño. Para los *antipatterns* que sí se detectan, se observó que hay un interés particular

por algunos de ellos, como el *antipattern "The Blob"*, el cual es considerado por más de la mitad de los estudios. En general, se observa un mayor interés en *antipatterns* clasificados como "Diseño orientado a objetos".

Por otro lado, se encontró que muchos autores que están interesados en la detección de *antipatterns*, evalúan técnicas o herramientas de detección sobre diferentes tipos de *antipatterns*. Es decir, la misma técnica es utilizada para detectar, por ejemplo, *antipatterns* clasificados como "Rendimiento" y como "Arquitectura orientada a servicios". Ejemplo de estos enfoques son presentados en los estudios de Ouni et al. (2017a), Ouni et al. (2017b) y Ouni et al. (2014). Este resultado sugiere que la construcción de técnicas o herramientas está todavía en una etapa experimental.

Con respecto a las herramientas encontradas en nuestro trabajo, se observa que muchas fueron propuestas en diferentes estudios, como soporte a una técnica de detección, pero no se encontró ninguna referencia en línea. Por lo que hace pensar que no hubo continuidad de la herramienta sino que solo se utilizó como prototipo de una técnica.

Respecto a los resultados obtenidos del estudio secundario de *code smells*, se observa que presenta una mayor cantidad de herramientas disponibles para descargar e instalar, que las encontradas en nuestro estudio de *antipatterns*. Este resultado puede deberse a que los *code smells* son detectados de forma más directa, es decir evaluando solo el código fuente, que los *antipatterns*.

Se considera que la herramienta *SonarQube* fue un importante anexo al resultado final de la revisión de *code smells*. Esta herramienta define un conjunto de reglas muy amplio de detección de posibles problemas en el código, para un conjunto muy grande de lenguajes de programación.

Respecto a las técnicas de detección de *antipatterns* se encontraron enfoques que utilizaban aprendizaje automático, mientras que para *code smells* no se presentaron este tipo de técnicas. Esto puede deberse a la naturaleza de los *code smells*, donde para detectarlos se analiza el código fuente y no problemas de diseño a más alto nivel.

Capítulo 4

Efectos del diseño sobre la aparición de *code smells*

El diseño es un proceso difícil de comprender para los estudiantes de pregrado, y el éxito (por ejemplo, construir un buen diseño) parece requerir un cierto nivel de desarrollo cognitivo que pocos estudiantes alcanzan (Carrington y Kim, 2003; Hu, 2013; Linder et al., 2006). La capacidad de los estudiantes para construir un buen diseño está relacionada con la capacidad de abstracción, comprensión, razonamiento y procesamiento de la información (Kramer, 2007; Leung y Bolloju, 2005; Siau y Tan, 2005).

Algunos estudios como el de Hayes y Over (1997) y el de Rombach et al. (2008) muestran una mejora en la calidad de los productos desarrollados con la adopción del *Personal Software Process* (PSP). En particular, el uso de las plantillas de diseño del PSP mejoran la calidad del software cuando son utilizadas por practicantes de la industria. En estos estudios la calidad del software fue medida como la densidad de defectos que se encontraron en el producto durante las pruebas unitarias.

Nuestro interés es conocer los defectos de diseño en los que incurren los estudiantes de pregrado y analizar si las plantillas de diseño del PSP evitan o disminuyen la aparición de alguno de ellos, a medida que los estudiantes desarrollan un conjunto de proyectos de software. La selección de los defectos de diseño y herramienta de detección tuvo como punto de partida los resultados obtenidos del mapeo sistemático, presentados en 3.2, y la naturaleza de los proyectos.

Dado que los proyectos son de pequeño porte y de baja complejidad, selec-

cionamos un conjunto de *code smells* para evaluar los proyectos y descartamos analizar *antipatterns* en este contexto.

En este capítulo se describe el análisis y resultado sobre los *code smells* en los que incurren los estudiantes al desarrollar un conjunto de proyectos y si las plantillas de diseño tienen un efecto en los diferentes tipos de *code smells* seleccionados.

4.1. Contexto

Para conocer el efecto del diseño detallado en la calidad interna del software (medida como cantidad y tipo de *code smells* en el código) se tomaron datos de tres estudios experimentales que fueron realizados en el contexto del curso *Principios y Fundamentos del Personal Software Process* de la carrera Ingeniería en Computación de la Facultad de Ingeniería, Universidad de la República, durante los años 2015, 2016 y 2017. De estos estudios experimentales participaron estudiantes avanzados de la carrera, que ya tenían aprobados cursos donde se enseña diseño detallado de software (principios de diseño, artefactos y diagramas de diseño, UML, patrones de diseño, etc.).

El curso tiene todos los años el mismo formato. Comienza con dos clases teóricas (de 2 horas cada una) donde se enseña el proceso de desarrollo que van a utilizar en el curso (denominado proceso base) y se explica la dinámica del trabajo práctico. El proceso base utiliza el marco de medición y las fases definidas (planificación, diseño, codificación, compilación, pruebas unitarias y postmortem) del PSP. Para seguir el proceso base se brinda un conjunto de *scripts*. Los *scripts* son una guía que establece las entradas, salidas y actividades a realizar en cada fase, los cuales ayudan al estudiante a encaminar las actividades del desarrollo pero sin exigir cómo deben realizarse.

Los datos que se registran son el tiempo dedicado a cada fase del proceso, los defectos detectados y removidos en cada fase y el tamaño en líneas de código (*locs*, del inglés *lines of code*) del proyecto construido. Las *locs* no representan el tamaño total de líneas de código del proyecto, sino que representan el tamaño en líneas de código agregadas o modificadas en el proyecto; la razón es porque al construir un proyecto es común reutilizar código fuente de proyectos anteriores. Las fases del proceso, los *scripts* y el registro de datos (*logs*) se presentan en la figura 4.1.

El trabajo práctico consiste en que cada estudiante desarrolle ocho proyec-

Figura 4.1: Proceso base, *scripts* y *logs*

tos pequeños siguiendo el proceso base y registrando los datos de ejecución del proceso en una herramienta. Los estudiantes eligen el lenguaje de programación a utilizar y realizan los proyectos de forma individual y consecutiva. No se comienza el proyecto 2 hasta no haber completado el proyecto 1 y así sucesivamente con los restantes proyectos.

Los proyectos son asignados a los estudiantes los días miércoles de cada semana y deben realizar la entrega el día domingo. La asignación consiste en el envío (por parte del docente) de los requerimientos de cada proyecto. La entrega de cada estudiante consiste en el código que resuelve el problema, los casos de prueba ejecutados y los datos registrados en la herramienta. Durante los días lunes y martes los docentes revisan la entrega y envían la corrección al estudiante. Los estudiantes realizan los proyectos en sus casas y tienen un docente asignado que será el responsable de asignar los proyectos, corregirlos y evacuar dudas.

Los proyectos, que son los mismos que se utilizan en el curso del PSP, son de baja y similar complejidad, y de naturaleza matemática (cálculo de la desviación estándar, regresión lineal, integración utilizando la regla Simpson, etc.), salvo el proyecto 2 que consiste en desarrollar un contador de lineas de código. El cuadro 4.1 presenta una breve descripción de los proyectos. La fase de diseño tiene foco en el diseño orientado a objetos y detallado; por ejemplo: identificar clases, atributos, operaciones, escenarios del programa (o componente), cambios de estado y/o pseudocódigo.

Cada estudio experimental se corresponde con una instancia del curso. Los estudiantes que participan de una instancia del curso no vuelven a participar de una instancia posterior.

Proyecto	Descripción
1	Calcular el promedio y la desviación estándar de un conjunto de números almacenados en una lista enlazada.
2	Contador de líneas de código de una clase y cantidad de métodos.
3	Calcular los parámetros de regresión lineal *b0* y *b1*, así como los coeficientes de correlación *r* y *r2* dado un conjunto de pares de valores.
4	Calcular los tamaños relativos de una clase para los rangos: muy pequeño, pequeño, medio, grande y muy grande.
5	Integrar numéricamente una función utilizando la *regla de Simpson*.
6	Calcular la función *t-student*: encontrar el valor de x para el cual integrando la función t, desde 0 hasta x, se obtiene el resultado p.
7	Calcular la correlación ente dos conjuntos de números X e Y, y la significancia de la correlación. Calcular los parámetros de regresión lineal *b0* y *b1* para un conjunto de n pares de datos, dar una estimación *XK*. Calcular la proyección *XK*, donde *XK* = *b0* + *b1*. Por ultimo, calcular el intervalo de predicción del 70 % para esa estimación.
8	Calcular los parámetros de estimación (*b0, b1, b2, b3*) para la regresión múltiple con tres variables.

Cuadro 4.1: Descripción de los proyectos

4.2. Diseño experimental

Todos los estudiantes aplican el proceso base en los primeros cuatro proyectos y registran sus datos en la herramienta. Durante los primeros cuatro proyectos la fase de diseño no exige la entrega de la representación del diseño. Una vez finalizado el proyecto 4 se divide a los estudiantes en dos grupos mediante un sorteo; se denomina a los grupos como "con representación de diseño utilizando plantillas" (*conPlantRD*) y "sin plantillas para la representación de diseño" (*sinPlantRD*). El cuadro 4.2 presenta el diseño del estudio experimental.

Grupo	Proyecto 1 a 4	Proyecto 5 a 8
conPlantRD	Proceso base	Proceso base + representación del diseño con plantillas
sinPlantRD	Proceso base	Proceso base

Cuadro 4.2: Diseño del estudio experimental

El grupo *sinPlantRD* es el grupo de control, que continúa aplicando el proceso base a lo largo de los proyectos 5 a 8. El grupo *conPlantRD*, además de continuar aplicando el proceso base, se agrega la práctica de diseño utilizando

las plantillas del PSP. Este grupo asiste a una clase teórica de 2 horas de duración donde se presentan las cuatro plantillas: operacional, funcional, lógica y estado. Las plantillas son documentos con una estructura predefinida en las cuales los estudiantes deben representar el diseño. Las plantillas permiten describir desde la operación del sistema (operacional) hasta el pseudocódigo de cada método (lógica). A continuación se presenta la descripción de cada una. Estas plantillas se detallan en el Anexo 4.

- Plantilla operacional: especifica la interacción del sistema y los usuarios. El contenido puede verse como similar a los casos de uso.
- Plantilla funcional: se especifica el comportamiento de las invocaciones y retornos del programa. Se describen variables, funciones, clases y métodos.
- Plantilla lógica: en esta plantilla se registra el pseudocódigo de cada método que aparece en la plantilla funcional.
- Plantilla de estado: define las transiciones y las condiciones de los estados internos del programa. El contenido es similar a los diagramas de máquinas de estado.

Los estudiantes del grupo *conPlantRD* deben representar su diseño utilizando las plantillas a partir del proyecto 5 y hasta el 8. La entrega de la representación del diseño utilizando plantillas es obligatoria. Cuando el estudiante entrega el proyecto, el docente asignado chequea (entre otras cosas) que las plantillas sean consistentes con el código. De esta forma se mitiga que el estudiante diseñe una solución y luego codifique otra. Sin embargo, no se controla que el diseño sea completo y verificable.

4.3. Objetivos

El objetivo de este análisis es conocer los *code smells* en los que incurren los estudiantes y evaluar si los estudiantes mejoran la calidad interna del software cuando utilizan plantillas para representar el diseño. A partir de este objetivo se plantearon las siguientes preguntas de investigación:

RQ1: ¿En cuáles *code smells* incurren los estudiantes durante el desarrollo de software?

RQ2: ¿La representación del diseño detallado utilizando plantillas mejora la calidad interna del software?

Para responder la pregunta RQ2, se analiza la calidad interna de dos formas:

- Entre grupos: refiere a conocer si existe diferencia en la calidad interna del software entre el grupo *sinPlantRD* y el grupo *conPlantRD*, mientras se utilizan las plantillas de diseño (proyecto 5 a 8).
- Intra grupo: refiere a conocer si existe diferencia en la calidad interna del software en el grupo *conPlantRD*, antes de usar las plantillas de diseño (proyecto 1 a 4) y mientras se utilizan las plantillas de diseño (proyecto 5 a 8).

4.4. Proceso de análisis

El proceso de análisis de los proyectos se dividió en cuatro fases. Una primera fase de planificación, que se realizó antes de comenzar a analizar los proyectos, y tres fases de análisis del código fuente de los proyectos, que fueron aplicadas de la misma forma para las tres instancias del curso. A continuación se presenta cada fase del proceso de análisis.

Fase I: Planificación

El objetivo de esta fase fue seleccionar la herramienta de detección de *code smells* y el conjunto de *code smells* a utilizar durante el análisis. Para seleccionar la herramienta se revisaron los resultados obtenidos de la revisión de la literatura de *code smells* presentados en la sección 3.2.2. La herramienta elegida fue *SonarQube* por las siguientes razones:

- Herramienta de software libre gratuito para una gran variedad de lenguajes de programación.
- Permite realizar análisis estático de código fuente de manera automática, buscando patrones con errores, malas prácticas o incidentes.
- Las reglas de revisión son actualizadas constantemente por la comunidad.
- Además de ofrecer su propio motor de análisis estático, permite integrarlo con otras herramientas conocidas como Checkstyle[1], PMD [2] o FindBugs[3].

[1] https://checkstyle.sourceforge.io/
[2] https://pmd.github.io/
[3] http://findbugs.sourceforge.net/

- Fácil instalación y uso de la herramienta acompañado de una amplia documentación actualizada.

En el Anexo 3 se describe la herramienta *SonarQube* y una guía de instalación y configuración de la herramienta.

De forma de realizar un análisis inicial, y que aportara valor a nuestra investigación, se seleccionaron los estudiantes que desarrollaron los proyectos en los lenguajes Java, C#, C, C++, Ruby y Python; descartándose los estudiantes que eligieron el lenguaje de programación PHP. Este lenguaje se descartó porque no es un lenguaje orientado a objetos y porque fueron solo tres los estudiantes que lo seleccionaron. El lenguaje C, que tampoco es un lenguaje orientado a objetos, no se descartó porque es utilizado en algunos cursos obligatorios de la currícula (es decir, los estudiantes están familiarizados con ese lenguaje) y además habían varios estudiantes que lo eligieron. Para los casos particulares de los proyectos implementados en C, se revisó que se aplicaran técnicas de programación orientada a objetos.

Luego de seleccionar la herramienta y los lenguajes de programación a utilizar, se buscó en *SonarQube* los *code smells* definidos para cada uno de los lenguajes[1]. Se encontraron 341 *code smells* para Java, 264 para C#, 195 para C, 323 para C++, 50 para Phyton y 33 para Ruby. Para que el análisis no dependiera del lenguaje de programación elegido por los estudiantes, se seleccionaron los *code smells* comunes a todos los lenguajes, donde Python y Ruby fueron los que limitaron la lista final de *code smells*. Finalmente se terminó descartando el lenguaje Phyton, las razones fueron porque solo un estudiante lo había elegido y además reducía la lista final de *code smells* comunes, aunque presentara mayor cantidad de *code smells* que Ruby.

Para los lenguajes Java, C#, C, C++ y Ruby se obtuvo una lista de 21 *code smells* comunes. Sin considerar Ruby, la cantidad de code smells comunes a los lenguajes Java, C#, C y C++ aumentaba a 34. Se analizaron los 13 *code smells* que no presentaba Ruby y solo tres podían ser agregados a la lista final de *code smells* (*Se debe usar el flujo de control "while" en lugar del flujo de control "for"*; *Los parámetros deben pasarse en el orden correcto*; *Las condiciones de parada del flujo de control "for" deben ser invariables*), el resto hacían referencia a estándares del lenguaje (por ejemplo: *Los archivos deben contener una línea vacía al final*) o no eran relevantes para el análisis

[1] https://rules.sonarsource.com

(por ejemplo: *Las secciones de código no deben comentarse*). Por esta razón y porque Ruby está siendo muy utilizado en los últimos años, no se descartó este lenguaje del análisis.

De la lista de 21 *code smells* comunes a todos los lenguajes se descartaron cinco, los motivos fueron porque hacían referencia a estándares de programación que no aportaban al análisis (*Los bloques de código anidados no deben dejarse vacíos*; *Se deben eliminar los pares de paréntesis redundantes*; *Los caracteres de tabulación no deben usarse*) o no eran relevantes para el análisis (*Seguimiento de etiquetas "FIXME"*; *Seguimiento de etiquetas "TODO"*). En el cuadro 4.3 se presentan los 16 *code smells* comunes a todos los lenguajes seleccionados para el análisis.

La cantidad de estudiantes inicial para los años 2015, 2016 y 2017 fueron 25, 17 y 19 respectivamente, sumando un total de 61 estudiantes. Luego de descartar los lenguajes PHP y Python se eliminaron cuatro estudiantes, dejando un total de 57 estudiantes para el análisis, 24 del año 2015, 17 del 2016 y 16 del 2017.

Id	Descripción
1	*Las declaraciones "if ... else if" deben terminar con la cláusula "else"* Siempre que a una instrucción "if" le siga una o más instrucciones "else if", el último "else if" debe ir seguido de una sentencia "else". En la declaración "else" debe tomarse la acción apropiada o escribir un comentario adecuado de por qué no se toma ninguna acción. Con el "else" final se busca aplicar programación defensiva.
2	*Las declaraciones "switch"/"case" no deben estar anidadas* Los "switch" anidados son difíciles de entender porque se pueden confundir fácilmente los "case" de un "switch" interno con los pertenecientes a una declaración de "switch" externo. Por lo tanto, se debe estructurar el código para evitar la necesidad de sentencias "switch" anidadas, pero si no se puede, entonces se debe considerar mover el "switch" interno a una función.
3	*Las declaraciones "switch"/"case" no deben tener demasiadas cláusulas "case"/"when"*

Continua en la siguiente página

Cuadro 4.3 – *Continuación de la página anterior*

Id	Descripción
4	*La complejidad cognitiva de las funciones o métodos no debe ser demasiado alta* La Complejidad Cognitiva es una medida numérica de qué tan difícil es comprender el flujo de control de un método. Esta medida permite establecer comparativas y por ende mejoras en los algoritmos que nos lleven a crear programas más "entendibles" y "manejables" por el ser humano. Nos interesa que el valor de la Complejidad Cognitiva sea el menor posible [1].
5	*Las declaraciones colapsables "if" deben fusionarse* Fusionar las declaraciones colapsables de "if" aumentan la legibilidad del código.
6	*Las declaraciones de flujo de control "if", "for", "while", "switch" y "try" no deben anidarse demasiado* Las sentencias "if", "for", "while", "switch", y "try" anidadas son ingredientes clave para hacer lo que se conoce como "código de espagueti". Tal código es difícil de leer, refactorizar y, por lo tanto, mantener.
7	*Las expresiones no deben ser demasiado complejas* La complejidad de una expresión es definida por la cantidad de "&&", "||" y condiciones "? ifTrue : ifFalse" que contiene. Para mantener el código legible la complejidad no debe ser demasiado alta.
8	*Los archivos no deben tener demasiadas líneas de código* Conocido en la literatura como *Large Class*, un archivo fuente que crece demasiado tiende a agregar demasiadas responsabilidades e, inevitablemente se vuelve más difícil de entender y por lo tanto de mantener. Por encima de un umbral específico, se recomienda refactorizarlo en partes más pequeñas de código que se centren en tareas bien definidas. Esos archivos más pequeños no solo serán más fáciles de entender sino también probablemente más fáciles de probar.
9	*Las funciones o métodos no deben tener demasiadas líneas de código* Conocido en la literatura como *Long Method*, un método que crece demasiado tiende a agregar demasiadas responsabilidades. Tal método inevitablemente se vuelve más difícil de entender y, por lo tanto, más difícil de mantener. Por encima de un umbral específico, se recomienda refactorizar en métodos más pequeños que se centren en tareas bien definidas. Esos métodos más pequeños no solo serán más fáciles de entender, sino también probablemente más fáciles de probar.

Continua en la siguiente página

[1] https://www.sonarsource.com/docs/CognitiveComplexity.pdf

Cuadro 4.3 – *Continuación de la página anterior*

Id	Descripción
10	*Las funciones o métodos no deben tener demasiados parámetros* Conocido en la literatura como *Long Parameter List*, tener un método con muchos parámetros provoca que el método sea poco mantenible, debido a que cuando hay un alto acoplamiento entre métodos, se vuelve inevitable pasar como parámetros muchas variables. Puede indicar además, que el método tiene demasiadas responsabilidades.
11	*Las líneas de código no deben ser demasiado largas* Tener que desplazarse horizontalmente hace que sea más difícil obtener una rápida comprensión y visión general del código.
12	Las funciones o métodos no deben ser vacíos
13	*Las declaraciones deben estar en líneas separadas* Para una mejor legibilidad, no se debe colocar más de una sentencia en una sola línea.
14	*Dos ramas en una estructura condicional no deben tener exactamente la misma implementación* Tener dos "case" en un "switch" o dos ramas en una cadena de "if" con la misma implementación es, en el mejor de los casos código duplicado y en el peor un error de codificación.
15	*Los parámetros de una función o método no utilizados deben eliminarse* Los parámetros no utilizados generan confusión, ya que cualesquiera que sean los valores pasados en dichos parámetros, harán que el comportamiento del método sea el mismo.
16	*Las variables locales no utilizadas deben eliminarse* Si se declara una variable local pero no se usa, es un código muerto y debe eliminarse. Eliminar estas variables mejorará la capacidad de mantenimiento y los desarrolladores no se preguntarán para qué se utiliza la variable.

Cuadro 4.3: Tipos de *code smell*

Fase II: Análisis de los proyectos en *SonarQube*

El objetivo de esta fase fue obtener para cada estudiante la cantidad de veces que incurrió en cada uno de los 16 *code smells* para cada proyecto. Para esto se configuró cada proyecto a analizar y se ejecutó el análisis. Los detalles de configuración y ejecución del análisis para cada lenguaje de programación se presentan en el Anexo 3.

Las actividades principales realizadas durante esta fase fueron:

- Verificación de la completitud de cada proyecto:

  Para cada proyecto, se verificó que esté completo (que estén todos los archivos) y que sea consistente el código fuente con el proyecto. En esta

actividad se encontraron algunos proyectos de estudiantes que tuvieron que ser descartados. Para cada proyecto descartado, se descartó el análisis del estudiante (es decir, se descartaron los ocho proyectos del estudiante).

- Compilación del código fuente para los proyectos en C#:

  Para los proyectos implementados en C#, antes de que se ejecute el análisis es necesario que el proyecto compile. En esta actividad se encontró un proyecto que no pudo ser compilado, como consecuencia se descartó el análisis del resto de los proyectos del estudiante.

- Configuración del proyecto y ejecución del análisis

  Se configuraron todos los proyectos de acuerdo al lenguaje de programación. Durante la ejecución del análisis, algunos proyectos presentaron errores y no se encontró una solución. La mayoría de estos proyectos estaban implementados en C y C++. Para estos casos se descartó el análisis del estudiante.

- Extracción de los *code smells*

  Al finalizar el análisis se puede consultar toda la información en el servidor *SonarQube*. Para los lenguajes C y C++, el análisis se puede consultar en el Reporte de *SonarLint* con *SonarCloud*. Esto facilitó la extracción de datos por estudiante en cada proyecto.

- Revisión manual de cada *code smell*

  Si bien la herramienta devolvía los *code smells* incurridos por los estudiantes en cada proyecto, se revisó manualmente cada uno de ellos con el objetivo de detectar falsos positivos o casos particulares, debido a la naturaleza de los proyectos. Por ejemplo, se observó que en el método *main* de los proyectos se leían parámetros de entrada y se verificaba su correctitud. En este tipo de método, la complejidad cognitiva era alta debido al uso de estructuras de bucle y condicionales, excepciones, entre otros. Esto implicaba que el *code smell* 4 estuviera presente en este método. Además, la interacción con el usuario para leer y desplegar errores o resultados, tuvo como consecuencia que se detectaran líneas de código demasiado largas, por ejemplo por el uso de oraciones extensas en un *print*, implicando la aparición del *code smell* 11. Otro ejemplo fue la

aparición del *code smell* 12, el cual se detectaba en constructores vacíos.

En el cuadro 4.4 se presenta un ejemplo de los datos extraídos para un estudiante. En esta actividad no se descartó ningún *code smell*, sino que se marcó en el cuadro la celda correspondiente al *code smell* y proyecto, para cada caso encontrado y se agregaron observaciones (como se visualiza en el cuadro).

Code smell	**Proyecto**								**Observaciones**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	
1	0	0	0	0	0	1	1	1	
2	0	0	0	0	0	0	0	0	
3	0	0	0	0	0	0	0	0	
4	0	1	0	0	0	1	1	1	Leer parámetros y verificar su correctitud
5	0	0	0	0	0	0	0	0	
6	0	1	0	0	0	3	0	0	
7	0	2	0	0	0	1	1	0	
8	0	0	0	0	0	0	0	0	
9	0	0	0	0	0	0	0	1	
10	0	0	0	0	0	0	0	1	
11	0	2	0	0	1	0	2	2	*Print* errores o resultados
12	1	0	1	1	1	1	1	0	Constructor vacío
13	0	0	0	0	0	0	0	0	
14	0	0	0	0	0	0	0	0	
15	0	0	0	0	2	0	4	0	
16	0	0	0	0	0	0	0	0	

Cuadro 4.4: Fase II: Ejemplo para un estudiante

De las actividades realizadas, se descartaron nueve proyectos de distintos estudiantes, como consecuencia se tuvo que descartar el análisis de esos estudiantes. Por lo tanto se analizaron en su totalidad 45 estudiantes, 19 del año 2015, 14 del 2016 y 12 del 2017. En total se analizaron 360 proyectos.

Fase III: Eliminación de *code smells* repetidos

En la construcción de un proyecto, los estudiantes podían utilizar código implementado en proyectos anteriores, es decir para construir por ejemplo el proyecto 3, se podía utilizar código de los proyectos 1 y 2. Por esta razón, el objetivo de esta fase fue eliminar los *code smells* derivados por la reutilización de código. Los reportes presentados por la herramienta fueron de gran utilidad en esta etapa. Además, se revisó manualmente los *code smells* de cada proyecto para asegurar que realmente fueran nuevos *code smells*. En el cuadro 4.5

se presenta un ejemplo de las cantidades depuradas para un estudiante luego de finalizada esta fase. Las celdas en amarillo indican que la cantidad de *code smells* disminuyó en ese proyecto.

Code smell	**Proyecto**								**Observaciones**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	
1	0	0	0	0	0	1	0	0	
2	0	0	0	0	0	0	0	0	
3	0	0	0	0	0	0	0	0	
4	0	1	0	0	0	1	1	0	Leer parámetros y verificar su correctitud
5	0	0	0	0	0	0	0	0	
6	0	1	0	0	0	3	0	0	
7	0	2	0	0	0	1	1	0	
8	0	0	0	0	0	0	0	0	
9	0	0	0	0	0	0	0	1	
10	0	0	0	0	0	0	0	1	
11	0	2	0	0	1	0	2	1	*Print* errores o resultados
12	1	0	0	0	0	0	0	0	Constructor vacío
13	0	0	0	0	0	0	0	0	
14	0	0	0	0	0	0	0	0	
15	0	0	0	0	2	0	4	0	
16	0	0	0	0	0	0	0	0	

Cuadro 4.5: Fase III: Ejemplo para un estudiante

Fase IV: Revisión de *code smells*

El objetivo de esta fase fue revisar los casos particulares registrados en la *Fase II* y unificar criterios para todos los estudiantes. En el cuadro 4.6 se presentan estos casos particulares y si se consideraron o no en el análisis. El cuadro 4.7 muestra un ejemplo de resultado final para el estudiante.

4.5. Resultados y Discusión

En esta sección se presentan de forma concisa los resultados obtenidos, respondiendo las preguntas de investigación planteadas.
Los datos crudos utilizados en el análisis se encuentran publicados en https://www.fing.edu.uy/biblio/datos-crudos-para-el-estudio-de-los-efectos-del-diseño-sobre-la-aparición-de-code-smells.

Del proceso de análisis se obtiene para cada estudiante la cantidad de *code smells* incurridos ($CantCSIncurridos_{cpe}$) para cada *code smell* y proyecto

Code smell	Descripción	¿Se considera?
4	Alta complejidad cognitiva en el método *main* debido a la lectura de parámetros de entrada y verificación de su correctitud.	No
4	Alta complejidad cognitiva en el método *main* debido a la lectura de parámetros de entrada, verificación de su correctitud y a la realización de operaciones sobre estos parámetros, para solucionar parte del proyecto.	Sí
9	Demasiadas líneas de código en el método *main* debido a la lectura de parámetros de entrada y verificación de su correctitud.	No
9	Demasiadas líneas de código en el método *main* debido a la lectura de parámetros de entrada, verificación de su correctitud y a la realización de operaciones sobre estos parámetros, para solucionar parte del proyecto.	Sí
11	Líneas demasiado largas debido a comentarios en el código.	No
11	Líneas demasiado largas debido a "prints" de errores o resultados.	No
11	Líneas demasiado largas debido a la definición de firmas de funciones o métodos.	No
11	Líneas demasiado largas debido a la invocación de métodos o funciones.	No
12	Constructores vacíos.	No
15	Variables por defecto sin utilizar en el método *main (*argv[])*.	No

Cuadro 4.6: Code smells: Análisis de casos particulares

(representados como en el cuadro 4.7). A partir de estos datos se calcula el promedio de cantidad de *code smells* incurridos en cada proyecto y para cada *code smell* de la siguiente forma:

$$PromedioCantCSIncurridos_{cp} = \frac{\sum_{e=1}^{\#estudiantes} CantCSIncurridos_{cpe}}{\#estudiantes} \tag{4.1}$$

siendo c el número de code smell (1..16), p el número de proyecto (1..8) y e el número de estudiante (1..45). $CantCSIncurridos_{cpe}$ es la cantidad de veces que aparece el *code smell* c en el código desarrollado por el estudiante e en el proyecto número p.

En el cuadro 4.8 se muestran los resultados del cálculo del promedio de cantidad de *code smells* incurridos en cada proyecto y para cada *code smell*.

De los datos obtenidos de cada estudiante, se deriva si el estudiante incurrió o no en cada *code smell* para cada proyecto ($CSIncurrido_{cpe}$) (ver cuadro 4.9,

Code smell	Proyecto							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	0	0	0	0	0	1	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	1	1	0
5	0	0	0	0	0	0	0	0
6	0	1	0	0	0	3	0	0
7	0	2	0	0	0	1	1	0
8	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	1
10	0	0	0	0	0	0	0	1
11	0	2	0	0	1	0	1	1
12	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0
15	0	0	0	0	2	0	4	0
16	0	0	0	0	0	0	0	0

Cuadro 4.7: Fase IV: Ejemplo para un estudiante

Code smell	Proyecto							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	0,18	0,42	0,04	0,02	0,09	0,11	0,07	0,27
2	0	0	0	0	0	0	0	0,11
3	0	0	0	0	0	0	0	0
4	0,11	0,53	0,02	0,11	0,11	0,40	0,29	0,96
5	0,02	0,40	0,04	0	0	0,02	0	0,18
6	0,40	0,89	0,36	0,51	0,38	0,71	0,78	1,67
7	0	0,27	0	0	0	0,04	0,07	0,02
8	0	0	0	0	0	0	0	0
9	0	0,07	0,13	0,16	0,07	0,18	0,29	0,73
10	0	0	0,02	0	0	0	0,13	0,60
11	0,02	0,62	0,69	0,04	0,18	0,04	0,60	2,29
12	0	0	0	0	0	0	0	0
13	0,24	0	0,02	0	0,04	0	0,02	0,42
14	0	0,04	0	0	0,02	0	0	0
15	0	0	0,04	0,04	0,09	0	0,13	0,11
16	0,09	0,13	0,38	0,13	0,09	0,07	0,24	0,69

Cuadro 4.8: Promedio de cantidad de *code smells* (por *code smell* y proyecto)

1 indica que incurrió y 0 indica que no incurrió). A partir de estos datos, se calcula el porcentaje de estudiantes que incurrieron al menos una vez en un *code smell* para cada *code smell* y proyecto de la siguiente forma:

$$PorcentajeCSIncurrido_{cp} = \frac{\sum_{e=1}^{\#estudiantes} CSIncurrido_{cpe}}{\#estudiantes} * 100 \qquad (4.2)$$

siendo c el número de code smell (1..16), p el número de proyecto (1..8) y e el número de estudiante (1..45). $CSIncurrido_{cpe}$ es 1 si aparece al menos una vez el *code smell* c en el código desarrollado por el estudiante e en el proyecto número p. En caso contrario es 0.

Code smell	**Proyecto**							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	0	0	0	0	0	1	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	1	1	0
5	0	0	0	0	0	0	0	0
6	0	1	0	0	0	1	0	0
7	0	1	0	0	0	1	1	0
8	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	1
10	0	0	0	0	0	0	0	1
11	0	1	0	0	1	0	1	1
12	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0
15	0	0	0	0	1	0	1	0
16	0	0	0	0	0	0	0	0

Cuadro 4.9: *Code smells* incurridos para un estudiante

En el cuadro 4.10 se muestran los resultados del cálculo del porcentaje de estudiantes que incurrieron al menos una vez en un *code smell* para cada *code smell* y proyecto.

A continuación se responden las dos preguntas de investigación considerando: (i) el promedio de la cantidad de *code smells* incurridos por los estudiantes para cada *code smell* y proyecto y (ii) el porcentaje de estudiantes que incurrieron en al menos un *code smell* para cada *code smell* y proyecto.

RQ1: ¿En cuáles *code smell* incurren los estudiantes durante el desarrollo de software?

En el cuadro 4.10 se observa que los estudiantes incurren en la mayoría de los *code smells* seleccionados a excepción del 3, 8 y 12, que no están presentes

Code smell	Proyecto							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	11 %	24 %	4 %	2 %	9 %	9 %	4 %	9 %
2	0 %	0 %	0 %	0 %	0 %	0 %	0 %	2 %
3	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %
4	16 %	51 %	2 %	11 %	11 %	44 %	27 %	71 %
5	2 %	22 %	4 %	0 %	0 %	2 %	0 %	2 %
6	24 %	64 %	18 %	29 %	22 %	44 %	33 %	51 %
7	0 %	22 %	0 %	0 %	0 %	4 %	4 %	2 %
8	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %
9	0 %	7 %	13 %	16 %	7 %	24 %	29 %	69 %
10	0 %	0 %	2 %	0 %	0 %	0 %	13 %	47 %
11	2 %	38 %	36 %	4 %	16 %	4 %	36 %	69 %
12	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %
13	2 %	0 %	2 %	0 %	2 %	0 %	2 %	11 %
14	0 %	4 %	0 %	0 %	2 %	0 %	0 %	0 %
15	0 %	0 %	4 %	2 %	7 %	0 %	7 %	9 %
16	7 %	9 %	9 %	9 %	9 %	4 %	13 %	20 %

Cuadro 4.10: Porcentaje de estudiantes que incurrieron al menos una vez en cada *code smell* (por *code smell* y proyecto)

en ninguno de los proyectos analizados. El *code smell* 2 solo está presente en el proyecto 8 pero el porcentaje es muy bajo.

Para los *code smells* 1, 4, 6, 11 y 16 se observa que fueron incurridos en todos los proyectos. El resto de los *code smells* (5, 7, 9, 10, 13, 14 y 15) presentan un comportamiento similar, es decir, se observa que en algunos proyectos los estudiantes incurren, en general, en un menor porcentaje de algunos de estos *code smells*, mientras que en otros proyectos no se presentan ocurrencias. En particular, para los *code smells* 9 y 10 se observa un aumento en el porcentaje en el proyecto 8.

Para el caso del *code smell* 6, que refiere a demasiadas sentencias anidadas, se observa que presenta el mayor porcentaje de estudiantes que los incurrieron en la mayoría de los proyectos, a excepción de los proyectos 3, 7 y 8. En los proyectos 3 y 7 el *code smell* 11, que hace referencia al largo en las líneas de código, es el que tiene mayor porcentaje de estudiantes que lo incurrieron. En el proyecto 8, el mayor porcentaje se presenta en en el *code smell* 4, el cual refiere a una alta complejidad cognitiva en el código. Este *code smell* también presenta el mayor porcentaje, junto con el *code smell* 6, en el proyecto 6.

Los *code smells* con mayor porcentaje de estudiantes que los incurrieron, se presentan en los proyectos 2 y 8. En particular, se puede observar que en

el proyecto 8 aumenta significativamente este porcentaje, respecto al resto de los proyectos, en los *code smells* 4, 9, 10 y 11. Este aumento hace pensar que el proyecto 8 resulta mas complejo para los estudiantes.

Respecto al promedio de la cantidad de *code smells*, se observa en el cuadro 4.8, que los *code smells* 6 y 11 presentan mayor promedio. Si se compara el promedio con el porcentaje, se cumple que, en general, hay coincidencia entre el *code smell* con mayor promedio y el *code smell* más incurrido por los estudiantes. Hay algunos casos donde, por ejemplo para el *code smell* 16 y proyecto 8, el porcentaje es bajo y el promedio es alto. También existen casos donde se observa para distintos *code smells* porcentajes similares pero promedios muy distintos, por ejemplo para los *code smell* 9 y 11 y proyecto 8. Estos resultados se deben a que existen estudiantes que incurren varias veces en un tipo de *code smell*.

En el cuadro 4.11 se presenta el porcentaje de estudiantes que incurrieron al menos una vez en un *code smell* para cada *code smell* y el promedio de la cantidad de *code smells* incurridos por los estudiantes para cada *code smell*, a lo largo del desarrollo de los proyectos. Este resultado está alineado con el presentado anteriormente, donde se observa que los *code smells* 4, 6, 9 y 11 tienen los mayores porcentaje de estudiantes que los incurrieron. Sin embargo, los mayores promedios se observan en el *code smell* 4, 6 y 11. Si bien el *code smell* 9 también presenta un alto promedio, se observa mayor promedio en el *code smell* 16 pero un menor porcentaje respecto al *code smell* 9 (casi la mitad) de estudiantes que lo incurrieron.

Este análisis mostró que en general los estudiantes incurren en la mayoría de los *code smells* seleccionados. En particular, el *code smell* 6 es el más incurrido por los estudiantes y el que presenta mayor promedio. Por otro lado se observó que en los proyectos 2 y 8 los valores del promedio y porcentaje aumentan significativamente respecto a los otros proyectos.

RQ2: ¿La representación del diseño detallado utilizando plantillas mejora la calidad interna del software?

Para responder a la pregunta se estudió cada plantilla, respecto a si entendemos que ayudarían a evitar la aparición de los *code smells* seleccionados, y se analizó la calidad interna de los proyectos entre grupos (*sinPlantRD*, *conPlantRD*) e intra grupo (*conPlantRD*).

Code smell	Porcentaje	Promedio
1	49 %	1,20
2	2 %	0,11
4	91 %	2,53
5	33 %	0,67
6	89 %	5,69
7	27 %	0,40
9	73 %	1,62
10	53 %	0,76
11	80 %	4,49
13	16 %	0,76
14	7 %	0,07
15	20 %	0,42
16	40 %	1,82

Cuadro 4.11: Porcentaje de estudiantes que incurrieron al menos una vez y promedio para cada *code smell* (por *code smell*)

El objetivo de la plantilla operacional es especificar un escenario que representa una secuencia de pasos, que describe una interacción entre un usuario y un sistema. El contenido puede verse como similar a los casos de uso. Por esta razón, creemos que los *code smells* 1, 5 y 14 podrían evitarse al describir estos escenarios, detallando los pasos y acciones.

El objetivo de la plantilla funcional es especificar el comportamiento de las invocaciones y retornos del programa, describiendo variables, funciones, clases y métodos, entre otros. Entendemos que especificar en detalle este artefacto podría prevenir la aparición de los *code smells* 10, 12, 15 y 16.

En la plantilla lógica se registra el pseudocódigo de cada método que aparece en la plantilla funcional. Esta descripción a un alto nivel es una forma de expresar los distintos pasos que va a realizar un programa, de la forma más parecida a un lenguaje de programación. Dado que uno de los objetivos principales es representar por pasos la solución a un problema o algoritmo, pensamos que la mayoría de los *code smells* podrían ser evitados con la utilización adecuada de esta plantilla. Los *code smells* que se descartan son el 8, ya que entendemos que no ayudaría a prevenir que los archivos presenten muchas líneas de código, y el 12, dado que si una función es vacía creemos que no se representaría con un pseudocódigo. Para los *code smells* 11 y 13, que refieren a convenciones de codificación o reglas de estilo, requieren que el programador conozca o defina estándares que ayuden a que el código sea más legible, por lo que consideramos que el uso de esta plantilla no evitaría la aparición de estos

code smells.

En la plantilla de estado se definen las transiciones y las condiciones de los estados internos del programa. El contenido es similar a los diagramas de máquinas de estado. Dado que se representa el estado, próximo estado, transiciones y acciones, pensamos que podrían evitarse *code smells* relacionados al uso de estructuras de control. Por este motivo consideramos que se podrían prevenir los *code smells* 1, 4, 5, 6 y 14 utilizando esta plantilla.

A partir de este análisis, se presenta en el cuadro 4.12 un resumen de los resultados obtenidos. Entendemos que a excepción del *code smell* 8 (que no fue incurrido por ningún estudiante), el resto de los *code smells* podrían ser evitados o disminuido su aparición por la utilización de alguna de estas plantillas.

Code smell	**Plantilla Operacional**	**Plantilla Funcional**	**Plantilla Lógica**	**Plantilla de Estado**
1 - Las declaraciones “if ... else if” deben terminar con la cláusula “else”	Sí	No	Sí	Sí
2 - Las declaraciones “switch”/“case” no deben estar anidadas	No	No	Sí	No
3 - Las declaraciones “switch”/“case” no deben tener demasiadas cláusulas “case”/“when”	No	No	Sí	No
4 - La complejidad cognitiva de las funciones o métodos no debe ser demasiado alta	No	No	Sí	Sí
5 - Las declaraciones colapsables “if” deben fusionarse	Sí	No	Sí	Sí
6 - Las declaraciones de flujo de control “if”, “for”, “while”, “switch” y “try” no deben anidarse demasiado	No	No	Sí	Sí
7 - Las expresiones no deben ser demasiado complejas	No	No	Sí	No
8 - Los archivos no deben tener demasiadas líneas de código	No	No	No	No
9 - Las funciones o métodos no deben tener demasiadas líneas de código	No	No	Sí	No
10 - Las funciones o métodos no deben tener demasiados parámetros	No	Sí	Sí	No
11 - Las líneas de código no deben ser demasiado largas	No	No	Sí	No
12 - Las funciones o métodos no deben ser vacíos	No	Sí	No	No
13 - Las declaraciones deben estar en líneas separadas	No	No	Sí	No
14 - Dos ramas en una estructura condicional no deben tener exactamente la misma implementación	Sí	No	Sí	Sí
15 - Los parámetros de una función o método no utilizados deben eliminarse	No	Sí	Sí	No
16 - Las variables locales no utilizadas deben eliminarse	No	Sí	Sí	No

Cuadro 4.12: Resultado del análisis respecto a si las plantillas de diseño podrían evitar los *code smells*

Análisis entre grupos

Para el análisis entre grupos (*sinPlantRD*, *conPlantRD*), se evaluó el porcentaje de estudiantes que incurren en al menos un *code smell* y el promedio de la cantidad de *code smells* incurridos por los estudiantes en los proyectos 5 a 8. Dado que los *code smells* 3, 8 y 12 no están presentes en ninguno de los proyectos analizados, no se presentan en el análisis. El cuadro 4.13 muestra el porcentaje de estudiantes que incurrieron en un *code smell* y el cuadro 4.14 muestra los resultados del promedio.

Code smell	**Grupo**	**Proyecto**			
		5	**6**	**7**	**8**
1	*sinPRD*	13 %	13 %	4 %	13 %
	conPRD	5 %	5 %	5 %	5 %
2	*sinPRD*	0 %	0 %	0 %	0 %
	conPRD	0 %	0 %	0 %	5 %
4	*sinPRD*	13 %	46 %	29 %	50 %
	conPRD	10 %	43 %	24 %	95 %
5	*sinPRD*	0 %	0 %	0 %	0 %
	conPRD	0 %	5 %	0 %	5 %
6	*sinPRD*	13 %	38 %	13 %	42 %
	conPRD	33 %	52 %	57 %	62 %
7	*sinPRD*	0 %	4 %	8 %	0 %
	conPRD	0 %	5 %	0 %	5 %
9	*sinPRD*	4 %	21 %	21 %	67 %
	conPRD	10 %	29 %	38 %	71 %
10	*sinPRD*	0 %	0 %	8 %	54 %
	conPRD	0 %	0 %	19 %	38 %
11	*sinPRD*	17 %	4 %	46 %	75 %
	conPRD	14 %	5 %	24 %	62 %
13	*sinPRD*	4 %	0 %	0 %	4 %
	conPRD	0 %	0 %	5 %	19 %
14	*sinPRD*	4 %	0 %	0 %	0 %
	conPRD	0 %	0 %	0 %	0 %
15	*sinPRD*	8 %	0 %	13 %	17 %
	conPRD	5 %	0 %	0 %	0 %
16	*sinPRD*	17 %	8 %	17 %	29 %
	conPRD	0 %	0 %	10 %	10 %
Total	*sinPRD*	67 %	50 %	75 %	92 %
	conPRD	52 %	71 %	76 %	100 %

Cuadro 4.13: Porcentaje de estudiantes que incurren en al menos un *code smell* (por *code smell*, grupo y proyecto)

Si se compara el porcentaje de estudiantes que incurrieron en los *code smells* en ambos grupos, se observa que en los proyectos donde se presenta mayor

Code smell	Grupo	Proyecto			
		5	**6**	**7**	**8**
1	*sinPRD*	0,13	0,13	0,04	0,46
	conPRD	0,05	0,10	0,10	0,05
2	*sinPRD*	0	0	0	0
	conPRD	0	0	0	0,24
4	*sinPRD*	0,13	0,33	0,33	0,63
	conPRD	0,10	0,48	0,24	1,33
5	*sinPRD*	0	0	0	0
	conPRD	0	0,05	0	0,38
6	*sinPRD*	0,21	0,46	0,21	0,63
	conPRD	0,57	1,00	1,43	2,86
7	*sinPRD*	0	0,04	0,13	0
	conPRD	0	0,05	0	0,05
9	*sinPRD*	0,04	0,08	0,21	0,75
	conPRD	0,10	0,29	0,38	0,71
10	*sinPRD*	0	0	0,08	0,75
	conPRD	0	0	0,19	0,43
11	*sinPRD*	0,17	0,04	0,75	2,38
	conPRD	0,19	0,05	0,43	2,19
13	*sinPRD*	0,08	0	0	0,13
	conPRD	0	0	0,05	0,76
14	*sinPRD*	0,04	0	0	0
	conPRD	0	0	0	0
15	*sinPRD*	0,13	0	0,25	0,21
	conPRD	0,05	0	0	0
16	*sinPRD*	0,17	0,13	0,33	1,08
	conPRD	0	0	0,14	0,24
Total	*sinPRD*	1,08	1,21	2,33	7,00
	conPRD	1,05	2,00	2,95	9,24

Cuadro 4.14: Promedio de la cantidad de *code smells* incurridos por los estudiantes (por *code smell*, grupo y proyecto)

porcentaje también existe un promedio alto de *code smells*. Existen algunas excepciones, por ejemplo el *code smell* 1, que presenta el mismo porcentaje de estudiantes que incurrieron en este *code smell* en los proyectos 5, 6 y 8 y grupo *sinPlantRD* pero el promedio es superior para el proyecto 8. Estos resultados se deben a que existen estudiantes que inciden más de una vez por proyecto en el mismo *code smell*.

Analizando el porcentaje de estudiantes que incurrieron al menos una vez en un *code smell* en ambos grupos *conPlantRD, sinPlantRD*, no se percibe que el uso de las plantillas mejore la calidad interna respecto a los *code smells* 6 y 9. En particular, para el *code smell* 6 se observa que el grupo *conPlantRD* incurre en un mayor porcentaje. Para todos los proyectos, el *code smell* 9 es incurrido en mayor porcentaje por el grupo *conPlantRD*, aunque la diferencia

con el grupo *sinPlantRD* es insignificante.

Para los *code smells* 4, 7, 10 y 13 se observa que para ciertos proyectos un grupo está mejor y para ciertos otros proyectos está mejor el otro grupo, aunque se percibe un aumento significativo en el code smell 4 y proyecto 8. Para los *code smells* 2 y 14, la diferencia es muy chica entre los grupos o no se incurrió en estos *code smells*. En definitiva, para ninguno de estos *code smells* se observan cambios al usar las plantillas.

Para el *code smell* 11, en general el grupo *sinPlantRD* presenta mayor porcentaje de estudiantes que lo incurren y mayor promedio, aunque se percibe un aumento de ambas variables en el proyecto 8. Por esta razón, no se observan cambios al usar las plantillas para este *code smell*.

Para el caso del *code smell* 1, se observa un porcentaje muy menor en todos los proyectos por parte del grupo que utiliza las plantillas (*conPlantRD*), a excepción del proyecto 7 donde ambos grupos se comportan casi igual. Desde el punto de vista de las plantillas, quizás alguna esté ayudando a los estudiantes a disminuir la introducción de este *code smell*.

Para los *code smells* 15 y 16, el grupo *conPlantRD* casi no incurre en ellos mientras que el grupo *sinPlantRD* sí lo presenta en la mayoría de los proyectos. El *code smell* 15 refiere a parámetros no utilizados en los métodos y el 16 a variables locales no utilizadas. Claramente estos tipos de *code smells* pueden ser evitados con buenos diseños de software. Desde el punto de vista del uso de las plantillas, quizás el desarrollo de pseudocódigo, que se representa en la plantilla lógica, o la plantilla funcional estén evitando que los estudiantes del grupo *conPlantRD* incurran en estos *code smells*.

El promedio y el porcentaje indican que los estudiantes incurren en mayor proporción en los tipos de *code smells* 4, 6 y 11. En general, el grupo *sinPlantRD* incurre en un mayor promedio en los tipos de *code smells* 1, 5, 10, 11, 14, 15 y 16 que el grupo *conPlantRD*, mientras que este último grupo presenta mayor promedio en los tipos de *code smells* 2, 4, 6, 9 y 13. Si bien el grupo *sinPlantRD* incurre en un mayor promedio en más *code smells*, la mayoría de los *code smells* incurridos por el grupo *conPlantRD* indican que el código es demasiado complejo y largo para ser comprendido.

Analizando estos resultados se puede pensar que los *code smells* 1, 15 y 16 podrían estar siendo evitados por el uso de las plantillas en el grupo *conPlantRD*, aunque es necesario analizar la evolución de estos *code smells* a lo largo del desarrollo de todos los proyectos. Para el resto de los *code smells*, no

se observa que el uso de las plantillas mejore la calidad interna de los proyectos.

Análisis en el grupo *conPlantRD*

Para el análisis del grupo *conPlantRD*, se evaluó el porcentaje de estudiantes que incurren en al menos un *code smell* y el promedio de *code smells* entre los proyecto 1 a 4 y 5 a 8. Dado que los *code smells* 3, 8 y 12 no están presentes en ninguno de los proyectos analizados, no se presentan en el análisis. El cuadro 4.15 muestra el porcentaje de estudiantes que incurrieron y el cuadro 4.16 muestra los resultados del promedio.

Code smell	**Proyecto**							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	19 %	19 %	10 %	0 %	5 %	5 %	5 %	5 %
2	0 %	0 %	0 %	0 %	0 %	0 %	0 %	5 %
4	24 %	43 %	5 %	10 %	10 %	43 %	24 %	95 %
5	0 %	24 %	10 %	0 %	0 %	5 %	0 %	5 %
6	38 %	67 %	29 %	29 %	33 %	52 %	57 %	62 %
7	0 %	19 %	0 %	0 %	0 %	5 %	0 %	5 %
9	0 %	10 %	19 %	14 %	10 %	29 %	38 %	71 %
10	0 %	0 %	5 %	0 %	0 %	0 %	19 %	38 %
11	0 %	29 %	29 %	0 %	14 %	5 %	24 %	62 %
13	5 %	0 %	5 %	0 %	0 %	0 %	5 %	19 %
14	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %
15	0 %	0 %	0 %	0 %	5 %	0 %	0 %	0 %
16	5 %	5 %	10 %	10 %	0 %	0 %	10 %	10 %
Total	52 %	100 %	57 %	48 %	52 %	71 %	76 %	100 %

Cuadro 4.15: Porcentaje de estudiantes que incurren en al menos un *code smell* en el grupo *conPlantRD* (por *code smell* y proyecto)

Si se compara el porcentaje de estudiantes que incurrieron en los *code smells* entre los proyectos 1 a 4 y 5 a 8, se observa que en general los proyectos donde se presenta mayor porcentaje de estudiantes que incurrieron, también existe un mayor promedio de *code smells*. Existen casos particulares como el *code smell* 4 y proyecto 8, donde el porcentaje es muy alto (95 %) pero el promedio es más bajo si se lo compara, por ejemplo, con el *code smell* 6 y proyecto 8.

Si se observa el porcentaje y promedio a lo largo del desarrollo de los 8 proyectos, estos no indican que el uso de las plantillas mejore la calidad interna, a excepción del *code smell* 1 que se va a analizar más en detalle.

Para los *code smell* 15 y 16 se observa que este grupo normalmente no incurrió (o lo hizo en muy bajo porcentaje). Observando los proyectos 1 a 4 y 5 a 8

Code smell	Proyecto							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	0,19	0,43	0,10	0	0,05	0,10	0,10	0,05
2	0	0	0	0	0	0	0	0,24
4	0,24	0,43	0,05	0,10	0,10	0,48	0,24	1,33
5	0	0,38	0,10	0	0	0,05	0	0,38
6	0,71	0,86	0,71	0,57	0,57	1,00	1,43	2,86
7	0	0,19	0	0	0	0,05	0	0,05
9	0	0,10	0,19	0,14	0,10	0,29	0,38	0,71
10	0	0	0,05	0	0	0	0,19	0,43
11	0	0,38	0,38	0	0,19	0,05	0,43	2,19
13	0,52	0	0,05	0	0	0	0,05	0,76
14	0	0	0	0	0	0	0	0
15	0	0	0	0	0,05	0	0	0
16	0,05	0,05	0,14	0,10	0	0	0,14	0,24
Total	1,71	3,29	1,76	0,90	1,05	2,00	2,95	9,24

Cuadro 4.16: Promedio de la cantidad de *code smell* incurridos por los estudiantes por el grupo *conPlantRD* (por *code smell* y proyecto)

por separado, no vemos una diferencia entre ellos. Es decir, el comportamiento de este grupo antes de usar plantillas y durante su uso no cambia para estos *code smells* . Entonces, la diferencia presentada en el análisis anterior entre los grupos (*conPlantRD*, *sinPlantRD*) no responde al uso de las plantillas.

Al igual que en el análisis anterior, se observa que tanto el porcentaje como el promedio indican que los estudiantes incurren en mayor proporción en el tipo de *code smell* 6. En este caso se observa que se cumple esto en casi todos los proyectos, a excepción del proyecto 8 donde se visualiza el mayor porcentaje en el *code smell* 4.

Para analizar el *code smell* 1, se muestra en el cuadro 4.17 la evolución de este *code smell* a lo largo de los proyectos para cada estudiante del grupo *conPlantRD*. Se observa que el 29 % de los estudiantes incurrieron en este *code smell* en los proyectos 1 a 4 y no incurrieron en los proyectos 5 a 8, el 57 % no incurrió en este *code smell* a lo largo de los proyectos y el 14 % incurrió en los proyectos 5 a 8 (en algunos casos también en los proyectos 1 a 4). Este resultado muestra que los estudiantes incurren en una menor cantidad en este *code smell*, por lo que no se puede asegurar, pero tampoco descartar, que este *code smell* puede ser evitado con el uso de las plantillas. De todas formas, es necesario analizar manualmente las plantillas entregadas por los estudiantes y tener entrevistas con ellos para conocer mejor si esto puede estar sucediendo por los motivos descritos. Esto aún no se ha realizado.

Juntando ambos análisis se desprende que en general el uso de las plantillas no mejora la calidad interna. En particular, el uso de las plantillas no tiene un efecto en los *code smells* en los que incurren los estudiantes al desarrollar software.

Estudiante	**Proyecto**							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	0	0	0	0	0	0	0	0
2	0	0	0	0	1	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	1	1	1	0	0	0	2	1
6	0	0	0	0	0	2	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0
9	1	2	0	0	0	0	0	0
10	1	0	1	0	0	0	0	0
11	0	4	0	0	0	0	0	0
12	1	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0
14	0	2	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0
19	0	1	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0

Cuadro 4.17: Cantidad de ocurrencia del *code smell* 1 por estudiante y proyecto

4.6. Relación entre los *code smells* y la enseñanza de grado

Realizamos una encuesta para conocer si los conocimientos impartidos en cursos obligatorios de la carrera Ingeniería en Computación, de la Universidad de la República, que son previas del curso *Principios y Fundamentos del Personal Software Process*, ayudan a evitar la aparición de un conjunto específico de *code smells* en el cual incurren los estudiantes de pregrado. Cada pregunta intentó responder si los conocimientos impartidos en una asignatura en particular ayudan a evitar que el estudiante incurra en un *code smell* en particular.

Los *code smells* seleccionados para esta encuesta fueron los elegidos en este trabajo, sin considerar aquellos que no fueron incurridos por ningún estudiante (*code smells* 3, 8 y 12).

Las asignaturas seleccionadas para realizar la encuesta fueron: *Programación 1* (P1), *Programación 2* (P2), *Programación 3* (P3), *Programación 4* (P4), *Taller de Programación* (TP) e *Introducción a la Ingeniería de Software* (IIS); las cuales son obligatorias en la currícula y requieren ser aprobadas para cursar *Principios y Fundamentos del Personal Software Process*. La encuesta fue contestada por el docente responsable de cada asignatura, a excepción de *Introducción a la Ingeniería de Software* que participaron dos docentes para llenar una única encuesta. En total se obtuvieron seis resultados, uno por cada asignatura seleccionada.

El cuadro 4.18 muestra los resultados obtenidos de la encuesta. Se puede observar que para las asignaturas *Programación 3* e *Introducción a la Ingeniería de Software*, los docentes consideran que los conocimientos impartidos en dichas asignaturas no ayudan a evitar la aparición de estos *code smells*.

Code smell	**Asignatura**					
	P1	**P2**	**P3**	**P4**	**TP**	**IIS**
1	No	No	No	No	No	No
2	No	No	No	No	No	No
4	No	No	No	Sí	No	No
5	No	Sí	No	No	No	No
6	No	No	No	No	No	No
7	No	Sí	No	Sí	No	No
9	No	Sí	No	Sí	Sí	No
10	No	No	No	Sí	Sí	No
11	No	No	No	No	No	No
13	Sí	Sí	No	Sí	No	No
14	Sí	Sí	No	Sí	No	No
15	Sí	Sí	No	Sí	Sí	No
16	Sí	Sí	No	Sí	Sí	No

Cuadro 4.18: ¿Se enseña cómo evitar los *code smells* en las asignaturas P1, P2, P3, P4, TP e IS?

Para los *code smells* 5, 7, 13, 14 y 15 los conocimientos impartidos en más de una asignatura podrían evitar estos *code smells*. Este resultado está alineado con nuestro análisis de los datos del curso, el cual reportó que estos *code smells* son incurridos en menor porcentaje por los estudiantes durante el desarrollo de los proyectos.

En algunas asignaturas también se imparten conocimientos sobre cómo evitar los *code smells* 9 y 10. Sin embargo, para este caso, en el análisis de los datos del curso encontramos un alto porcentaje de estudiantes que incurren en estos *code smells* (en particular en el Proyecto 8).

Para los *code smell* 15 y 16, la encuesta mostró que cuatro de seis asignaturas brindan conceptos que ayudarían a evitarlos. Este resultado va en la misma dirección que los obtenidos en nuestro análisis. Para ambos *code smells* existe un bajo porcentaje de estudiantes que los incurren.

Del resultado de la encuesta se observa que los *code smells* 1, 6 y 11 no están cubiertos por ninguna asignatura. Este resultado está alineado con nuestro estudio, en el cual se observó que estos *code smells* son incurridos en todos los proyectos. En particular, existe un alto porcentaje de estudiantes que incurren en los *code smells* 6 y 11 (Proyectos 2 y 8).

Por otro lado, solo una asignatura (*Programación 4*) imparte conocimientos sobre cómo evitar el *code smell* 4. Sin embargo, vimos en el análisis que los estudiantes incurren en un alto porcentaje (proyectos 2 y 8) en dicho *code smell*.

Para el caso particular del *code smell* 2, la encuesta indicó que ninguna asignatura brinda conocimientos para evitarlo. Sin embargo, se observa que este *code smell* fue incurrido, solo en el proyecto 8, en un muy bajo porcentaje por los estudiantes.

En general, la encuesta reveló que la mayoría de los *code smells* podrían ser evitados debido al conocimiento que se imparte en algunos de los cursos obligatorios. Sin embargo, los estudiantes incurren en estos *code smells* al desarrollar software.

4.7. Amenazas a la validez

Los datos extraídos para realizar este análisis son de tres estudios experimentales realizados entre los años 2015 a 2017. Estos estudios experimentales se plantearon como estudios preliminares para llevar a cabo investigaciones iniciales como el análisis realizado en esta tesis. El objetivo es poder formalizar experimentos controlados que puedan paliar algunas de las amenazas encontradas, como el análisis de los programas en diferentes lenguajes de programación y la eliminación de los *code smells* repetidos.

Este análisis sirve como puntapié inicial para generar nuevas hipótesis y

diseñar nuevos estudios experimentales que permitan estudiar en cuáles *code smells* incurren los estudiantes y si las plantillas del PSP, u otros artefactos, logran evitar alguno de ellos.

Con respecto al análisis, el proceso de selección, la extracción de datos y conclusiones para responder la primera pregunta de investigación, fue realizada por dos personas. Sin embargo, el resto del análisis fue ejecutado solo por la autora de la tesis, lo cual implica una amenaza a la validez interna.

Por último, la cantidad de estudiantes en el estudio constituye una amenaza a la conclusión, dado que se analizaron solo los proyectos de 45 estudiantes.

4.8. Conclusiones

En el contexto de los estudios experimentales encontramos que en general los estudiantes incurren en la mayoría de los *code smells* seleccionados.

De los 16 *code smells* seleccionados concluimos que, a excepción del *code smell* 8 o *Large Class*, el resto de los *code smells* podrían ser evitados o disminuido su aparición por la utilización de alguna de las plantillas de diseño.

Por otro lado, entendemos que la mayoría de los *code smells* podrían ser evitados debido al conocimiento que se imparte en algunos de los cursos obligatorios de nuestra Facultad.

Sin embargo, los estudiantes incurrieron en estos *code smell* al desarrollar los proyectos. Al menos, esto es el hallazgo de nuestro análisis. Por algún motivo, los estudiantes no logran incorporar estos conocimientos y habilidades al momento de diseñar y construir software.

Es más, encontramos que la representación del diseño utilizando plantillas no ayudó a desarrollar productos de software de mejor calidad. Los resultados muestran que el uso de plantillas no mejoró la calidad interna (medido como cantidad de *code smells* en el código) de los proyectos desarrollados por los estudiantes que las utilizaron.

Esto puede deberse a varios factores que se deberán analizar en el futuro. Podría ser, entre otros motivos, que no están habituados a estas plantillas y por ende no se obtuvo el beneficio esperado, podría ser que simplemente completaron las plantillas pero no se preocuparon en ese momento por pensar y desarrollar un diseño de calidad o podría ser que los estudiantes no saben qué y cómo diseñar; es decir, que no han adquirido dicha habilidad durante sus estudios de grado.

Capítulo 5

Conclusiones y Trabajo a Futuro

En este capítulo se presentan las conclusiones de la tesis y se plantean posibles trabajos a futuro.

5.1. Conclusiones

La construcción de software de calidad es una preocupación importante para la comunidad del software. Crear un diseño simple y eficiente puede ser una tarea muy compleja, en parte porque el diseño de software es un proceso creativo y difícil de comprender. Construir un buen diseño requiere un cierto nivel de desarrollo cognitivo que, al parecer, pocos estudiantes de pregrado alcanzan.

Las malas prácticas de diseño de software originan defectos de diseño, como *antipatterns* y *code smells*, los cuales afectan negativamente a los factores de calidad del software. Estas prácticas son causadas a menudo por la inexperiencia, el conocimiento insuficiente o la urgencia.

La búsqueda por desarrollar software de calidad ha dado lugar a un gran número de procesos de desarrollo, como es el *Personal Software Process* (PSP). Se ha demostrado que la inserción del PSP mejora la calidad de los productos desarrollados. En particular, el uso de las plantillas de diseño permite registrar el diseño de forma completa y precisa.

En el marco de esta tesis, se realizó un mapeo sistemático para conocer sobre las técnicas y herramientas de detección de *antipatterns*. No se efectuó un mapeo sistemático sobre *code smells*, dado que en la búsqueda de antecedentes encontramos una revisión secundaria sobre técnicas y herramientas de estos

defectos de diseño.

Se conformó un catálogo de *antipatterns* con una ficha para cada uno, incluyendo datos como: nombre, descripción, clasificación, referencias a la bibliografía del *antipattern*, técnicas de detección y características de las herramientas de detección. Con este catálogo se logró cumplir con el primer objetivo planteado en nuestro trabajo: conocer el cuerpo de conocimiento sobre las técnicas y herramientas de detección de *antipatterns*.

Del resultado de nuestro mapeo, encontramos que existen muchos *antipatterns* que refieren al mismo problema pero son conocidos con distinto nombre. Como consecuencia, no encontramos una nomenclatura unificada para hacer referencia a los *antipatterns*. Además, observamos que hay una atención particular por algunos *antipatterns* y un mayor interés en *antipatterns* orientados a objetos.

Por otro lado, detectamos que es complejo desarrollar técnicas o herramientas de detección de *antipatterns* que utilizan un enfoque automático. Además, hay varios autores que evalúan técnicas o herramientas de detección sobre diferentes tipos de *antipatterns*. Es decir, la misma técnica o herramienta es utilizada para detectar, por ejemplo, *antipatterns* enfocados al rendimiento y a arquitecturas orientadas a servicios. Estos resultados sugieren que la construcción de técnicas o herramientas está todavía en una etapa experimental.

Respecto al estudio secundario de *code smells*, observamos que se presentó una mayor cantidad de herramientas disponibles para descargar e instalar, que las encontradas para *antipatterns*. Este resultado puede deberse a que los *code smells* son detectados de forma más directa, es decir evaluando solo el código fuente, que los *antipatterns*. Por otro lado, consideramos que la herramienta *SonarQube* fue un importante anexo al resultado final de la revisión de *code smells*.

Para conocer los defectos de diseño en los que incurren estudiantes de pregrado y para estudiar los efectos de aplicar plantillas de diseño, se analizaron proyectos de estudiantes en el contexto de un curso. Los estudiantes desarrollan ocho proyectos pequeños siguiendo un proceso definido. Durante el desarrollo de los primeros cuatro proyectos, la fase de diseño no exige la entrega de la representación del diseño. Una vez finalizado el proyecto 4 se divide a los estudiantes en dos grupos. Los estudiantes de uno de los grupos debe representar su diseño utilizando un conjunto de plantillas de diseño, a partir del proyecto 5 y hasta el 8.

Para este análisis seleccionamos 16 *code smells*, tomando como punto de partida los resultados obtenidos del mapeo sistemático y la naturaleza de los proyectos. La herramienta utilizada para evaluar los proyectos fue *SonarQube*.

Observamos que en general los estudiantes incurren en la mayoría de los *code smells* seleccionados. En particular, el *code smell* 6 (las declaraciones de flujo de control no deben anidarse demasiado) es el más incurrido por los estudiantes y el que presenta mayor promedio, independientemente del proyecto.

Sin embargo, de los *code smells* seleccionados concluimos que, a excepción del *code smell* 8 o *Large Class* (los archivos no deben tener demasiadas líneas de código), el resto de los *code smells* podrían ser evitados o disminuido su aparición por la utilización de alguna de las plantillas de diseño. En particular, el *code smell* 8 no fue incurrido por ningún estudiante. Este resultado, puede deberse a que los proyectos desarrollados son pequeños y relativamente sencillos.

Por otro lado, concluimos que la mayoría de los *code smells* podrían ser evitados debido al conocimiento que se imparte en algunos de los cursos obligatorios. Sin embargo, los estudiantes incurrieron en estos *code smell* al desarrollar los proyectos.

Además, encontramos que el uso de las plantillas no mejoran la calidad interna de los proyectos por los estudiantes que las utilizaron. El uso de estas plantillas no evitan ni disminuyen la aparición de estos *code smell*.

Con este análisis se logró cumplir el segundo objetivo propuesto en esta tesis: conocer la calidad interna de los productos de software desarrollados por estudiantes de pregrado de nuestra Facultad, cuando utilizan y cuando no utilizan un conjunto específico de plantillas de soporte para la representación del diseño de software.

Estos resultados están alineados con estudios presentados por otros autores, donde se observa que los estudiantes presentan dificultades para diseñar un sistema de software. Es más, por algún motivo, los estudiantes de nuestra Facultad no logran incorporar los conocimientos brindados en diferentes cursos.

Los principales aportes de esta tesis son los siguientes:

- Un catálogo de *antipatterns* que puede ser utilizado en otras investigaciones y proporciona a los profesionales una visión general de estos defectos de diseño y su detección. Este catálogo sirve como guía en el diseño de software.

- Una máquina virtual con la plataforma *SonarQube* instalada, para la detección de *code smells* en programas implementados en Java, C, C++, C# y Ruby, que facilita la continuidad de este y otros análisis.
- Los resultados de este análisis y los datos crudos sobre los *code smells* en los que incurren los estudiantes sirve como puntapié inicial para generar nuevas hipótesis y diseñar nuevos estudios experimentales.

5.2. Trabajo a futuro

En la revisión secundaria es necesario notar que la literatura seleccionada es limitada. Si bien seleccionamos tres buscadores digitales que cubren la mayor cantidad de estudios presentados en revistas y actas de conferencias, en el área de Ingeniería de Software, creemos que sería muy beneficioso agregar otras fuentes de conocimiento; por ejemplo, incluir literatura gris y agregar otros buscadores digitales.

También podemos pensar en probar todas las herramientas encontradas en nuestra revisión secundaria, para tener un panorama más claro sobre las bondades y limitaciones de cada una de ellas, y así poder usarlas en futuros análisis.

Como forma de darle continuidad a la investigación realizada, sobre técnicas y herramientas de detección de *antipatterns*, pensamos en informatizar el catálogo. Por ejemplo, mediante el uso de ontologías y herramientas de soporte a las mismas. Este sistema podría ser usado por la comunidad para realizar consultas, actualizar información o agregar nuevos *antipatterns*. Además, se puede realizar otro mapeo sistemático que se enfoque en técnicas y herramientas de refactorización de *antipatterns*. Este insumo sería muy valioso para agregar al catálogo de *antipatterns* construido durante esta tesis.

Adicionalmente, creemos que se podría mejorar aún más la presentación del catálogo, por ejemplo, relacionando el lenguaje de programación con la herramienta de detección. Si bien esa información se presenta en el Anexo 1, sería muy útil poder visualizarla en cada ficha del catálogo.

Los estudios experimentales se consideran estudios preliminares y sería conveniente diseñar a futuro nuevos experimentos para analizar en cuáles *code smells* incurren los estudiantes.

Un aporte importante podría ser continuar con nuestro análisis, por ejemplo evaluar nuevos *code smells*, agregar nuevas métricas para analizar los proyectos

o distinguir por lenguaje de programación. En nuestro trabajo, calculamos la densidad de *code smells* [1] para cada *code smell* y proyecto, donde los proyectos elegidos fueron los implementados en Java. Los datos para continuar este análisis están publicados en https://www.fing.edu.uy/biblio/datos-crudos-para-el-estudio-de-los-efectos-del-diseño-sobre-la-aparición-de-code-smells.

Respecto a *antipatterns*, nos parece importante conocer en cuáles incurren los estudiantes. Por ejemplo, se podría evaluar proyectos de mediano porte desarrollados en el curso *Proyecto de Ingeniería de Software* (PIS) de nuestra Facultad. En el PIS, se conforman equipos de estudiantes que realizan proyectos para clientes reales. El objetivo del curso es afirmar y profundizar los conocimientos impartidos en el curso *Introducción a Ingeniería de Software*, contrastarlos con su aplicación práctica e integrarlos con conocimientos de otras materias, como diseño, programación, base de datos, arquitectura.

Finalmente, creemos conveniente explorar las causas por las cuales los estudiantes de nuestra Facultad no logran evitar ciertos *code smells*. En este sentido, se podría realizar una investigación en conjunto con docentes de distintas asignaturas para diseñar nuevos experimentos. Por ejemplo, se podría evaluar el código producido en cierta asignatura y agregar al análisis defectos de diseño relacionados al conocimiento que esta asignatura brinda.

[1] Densidad de *code smells*: Refiere a los *code smells* por KLOC (mil líneas de código) encontrados en un proyecto.

Referencias bibliográficas

Abbes, M., Khomh, F., Gueheneuc, Y., y Antoniol, G. (2011). An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. En *15th European Conference on Software Maintenance and Reengineering*, pages 181–190. IEEE.

Akroyd, M. (1996). Antipatterns session notes. *Object World*.

Alexander, C., Alexander, P., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., y Shlomo, A. (1977). *A Pattern Language: Towns, Buildings, Construction*. OUP USA.

Aras, M. y Selçuk, Y. (2016). Metric and rule based automated detection of antipatterns in object-oriented software systems. En *7th International Conference on Computer Science and Information Technology*, pages 1–6. IEEE.

Arcelli, D., Berardinelli, L., y Trubiani, C. (2015). Performance antipattern detection through fuml model library. En *Proceedings of the 2015 Workshop on Challenges in Performance Methods for Software Development*, pages 23–28.

Arcelli, D., Cortellessa, V., y Di Pompeo, D. (2018). Performance-driven software model refactoring. *Information and Software Technology*, 95:366–397.

Arcoverde, R., Garcia, A., y Figueiredo, E. (2011). Understanding the longevity of code smells: preliminary results of an explanatory survey. En *Proceedings of the 4th Workshop on Refactoring Tools*, pages 33–36.

Bagheri, H., Kang, E., Malek, S., y Jackson, D. (2015). Detection of design flaws in the android permission protocol through bounded verification.

Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 9109:73–89.

Baker, B. (1995). On finding duplication and near-duplication in large software systems. En *Proceedings of 2nd Working Conference on Reverse Engineering*, pages 86–95. IEEE.

Bavota, G., De Lucia, A., Di Penta, M., Oliveto, R., y Palomba, F. (2015). An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*.

Beck, K. y Cunningham, W. (1987). Using pattern languages for object oriented programs. En *Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

Bersani, M., Marconi, F., Tamburri, D., Jamshidi, P., y Nodari, A. (2016). Continuous architecting of stream-based systems. En *2016 13th Working IEEE/IFIP Conference on Software Architecture*, pages 146–151. IEEE.

Biray y Buzluca, F. (2015). A learning-based method for detecting defective classes in object-oriented systems. En *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops*, pages 1–8.

Bloch, J. y Steele, G. (2001). *Effective Java: Programming Language Guide*. Addison-Wesley Java series. Addison-Wesley.

Bourque, P. y Fairley, R. (2014). *Guide to the Software Engineering Body of Knowledge - SWEBOK v3.0*. IEEE Computer Society, 2014 version edition.

Brown, W., Malveau, R., McCormick, H., y Mowbray, T. (1998). *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc.

Brühlmann, A., Gîrba, T., Greevy, O., y Nierstrasz, O. (2008). Enriching reverse engineering with annotations. En *International Conference on Model Driven Engineering Languages and Systems*, pages 660–674. Springer.

Budi, A., Lo, D., Jiang, L., y Wang, S. (2011). Automated detection of likely design flaws in layered architectures. En *33th International Conference on Software Engineering*.

Carrington, D. y Kim, S. (2003). Teaching software design with open source software. En *33rd Annual Frontiers in Education, 2003. FIE 2003.*, volume 3, pages S1C–9. IEEE.

Chatzigeorgiou, A. y Manakos, A. (2010). Investigating the evolution of bad smells in object-oriented code. En *2010 Seventh International Conference on the Quality of Information and Communications Technology*, pages 106–115. IEEE.

Chen, T., Cooper, S., McCartney, R., y Schwartzman, L. (2005). The (relative) importance of software design criteria. En *ITiCSE*.

Cherbakov, L., Ibrahim, M., y Ang, J. (2006). Soa antipatterns: The obstacles to the adoption and successful realization of service-oriented architecture.

Corporation, R. S. (1997). *UML Extension for Objectory Process for Software Engineering: Version 1.1*. Rational Software.

Cortellessa, V., Di Marco, A., Eramo, R., Pierantonio, A., y Trubiani, C. (2009). Approaching the model-driven generation of feedback to remove software performance flaws. En *2009 35th Euromicro Conference on Software Engineering and Advanced Applications*, pages 162–169. IEEE.

Crasso, M., Zunino, A., Moreno, L., y Campo, M. (2009). Jeetuningexpert: A software assistant for improving java enterprise edition application performance. *Expert Systems with Applications*, 36(9):11718–11729.

Czibula, G., Marian, Z., y Czibula, I. (2015). Detecting software design defects using relational association rule mining. *Knowledge and Information Systems*, 42(3):545–577.

De Sanctis, M., Trubiani, C., Cortellessa, V., Di Marco, A., y Flamminj, M. (2017). A model-driven approach to catch performance antipatterns in adl specifications. *Information and Software Technology*, 83:35–54.

Dodani, M. (2006). Patterns of anti-patterns. *Journal of Object Technology*, 5(6):29–33.

Dudney, B., Krozak, J., Wittkopf, K., Asbury, S., y Osborne, D. (2002). *J2EE Antipatterns*. John Wiley & Sons, Inc., USA, 1 edition.

Eckerdal, A., McCartney, R., Moström, J., Ratcliffe, M., y Zander, C. (2006a). Can graduating students design software systems? En *SIGCSE'06*. ACM.

Eckerdal, A., McCartney, R., Moström, J., Ratcliffe, M., y Zander, C. (2006b). Categorizing student software designs: Methods, results, and implications. *Computer science education*, 16(3).

Erdemir, U., Tekin, U., y Buzluca, F. (2011). E-quality: A graph based object oriented software quality visualization tool. En *2011 6th International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–8. IEEE.

Fernandes, E., Oliveira, J., Vale, G., Paiva, T., y Figueiredo, E. (2016). A review-based comparative study of bad smell detection tools. En *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering - EASE '16*, pages 1–12.

Fontana, F., Roveda, R., Vittori, S., Metelli, A., Saldarini, S., y Mazzei, F. (2016). On evaluating the impact of the refactoring of architectural problems on software quality. En *Proceedings of the Scientific Workshop Proceedings of XP2016*, pages 1–8.

Fowler, M. (1997). *Analysis patterns: reusable object models*. Addison-Wesley Professional.

Fowler, M. (1999). *Refactoring: Improving the Design of Existng Code*. Addison-Wesley Professional.

Gamma, E., Helm, R., Johnson, R., y Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition.

Gamma, E., Helm, R., Johnson, R., y Vlissides, J. (2001). *Design Patterns: Abstraction and Reuse of Object-Oriented Design*. Springer Berlin Heidelberg, Berlin, Heidelberg.

Ganea, G. y Marinescu, R. (2015). Modeling design flaw evolution using complex systems. En *2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 433–436. IEEE.

Garousi, V., Felderer, M., y Mäntylä, M. (2019). Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology*, 106:101–121.

Grazioli, F., Vallespir, D., Pérez, L., y Moreno, S. (2014). The impact of the psp on software quality: Eliminating the learning effect threat through a controlled experiment. *Advances in Software Engineering*, 2014.

Hassaine, S., Khomh, F., Guéhéneuc, Y., y Hamel, S. (2010). Ids: An immune-inspired approach for the detection of software design smells. En *2010 Seventh International Conference on the Quality of Information and Communications Technology*, pages 343–348. IEEE.

Hayes, W. y Over, J. (1997). The personal software process (psp): An empirical study of the impact of psp on individual engineers. Technical Report CMU/SEI-97-TR-001, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Hecht, G., Benomar, O., Rouvoy, R., Moha, N., y Duchien, L. (2015). Tracking the software quality of android applications along their evolution (t). En *2015 30th IEEE/ACM International Conference on Automated Software Engineering*, pages 236–247.

Hu, C. (2013). The nature of software design and its teaching: An exposition. *ACM Inroads*, 4(2):62–72.

Humphrey, W. (1995). *A discipline for software engineering*. Addison-Wesley Longman Publishing Co., Inc.

Jebelean, C. (2004). Automatic detection of missing abstract-factory design pattern in object-oriented code. En *Proceedings of the International Conference on Technical Informatics*. Politehnica University in Timişoara.

Jones, S. (2006). *SOA anti-patterns*. URL https://www.infoq.com/articles/SOA-anti-patterns. Último acceso: mayo 2020.

Karasneh, B., Jolak, R., y Chaudron, M. (2015). Using examples for teaching software design: An experiment using a repository of uml class diagrams. En *2015 Asia-Pacific Software Engineering Conference*.

Kaur, H. y Kaur, P. (2014). Optimized unit testing for antipattern detection. En *Proceedings of the 2014 International Conference on Information and Communication Technology for Competitive Strategies*, pages 1–8.

Kessentini, M., Kessentini, W., Sahraoui, H., Boukadoum, M., y Ouni, A. (2011). Design defects detection and correction by example. En *2011 IEEE 19th International Conference on Program Comprehension*, pages 81–90. IEEE.

Khan, A. (2012). Impact of personal software process on software quality. *IOSR Journal of Computer Engineering*, 1:21–25.

Khomh, F., Di Penta, M., y Gueheneuc, Y.-G. (2009). An exploratory study of the impact of code smells on software change-proneness. En *2009 16th Working Conference on Reverse Engineering*, pages 75–84. IEEE.

Khomh, F., Di Penta, M., Guéhéneuc, Y., y Antoniol, G. (2012). An exploratory study of the impact of antipatterns on class change and fault-proneness. *Empirical Software Engineering*, 17:243–275.

Khomh, F., Vaucher, S., Guéhéneuc, Y., y Sahraoui, H. (2011). A gqm-based bayesian approach for the detection of antipatterns. *Journal of Systems and Software*, pages 559–572.

Kis, M. (2002). Information security antipatterns in software requirements engineering. En *9th Conference of Pattern Languages of Programs*, volume 11.

Kitchenham, B., Budge, D., y Brereton, P. (2015). *Evidence-Based Software Engineering and Systematic Reviews*. Taylor & Francis.

Kral, J. y Zemlicka, M. (2007). The most important service-oriented antipatterns. En *2nd International Conference on Software Engineering Advances*, pages 29–29.

Kramer, J. (2007). Is abstraction the key to computing? *Commun. ACM*, 50.

Kreimer, J. (2005). Adaptive detection of design flaws. *Electronic Notes in Theoretical Computer Science*, 141(4 SPEC. ISS.):117–136.

Lakos, J. (1996). *Large-scale C++ software design*. Addison-Wesley.

Lanza, M. y Marinescu, R. (2006). *Object-Oriented Metrics in Practice*. Springer-Verlag.

Larman, C. (1998). *Applying UML and patterns*. Prentice Hall Englewood Cliffs, NJ.

Leung, F. y Bolloju, N. (2005). Analyzing the quality of domain models developed by novice systems analysts. En *38th Hawaii International Conference on System Sciences*.

Li, W. y Shatnawi, R. (2007). An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software*, 80:1120–1128.

Linder, S., Abbott, D., y Fromberger, M. (2006). An instructional scaffolding approach to teaching software design. *Journal of Computing Sciences in Colleges*, 21.

Loftus, C., Thomas, L., y Zander, C. (2011). Can graduating students design: revisited. En *Proceedings of the 42nd ACM technical symposium on Computer science education*. ACM.

Maiga, A., Ali, N., Bhattacharya, N., Sabané, A., Guéhéneuc, Y., y Aimeur, E. (2012a). Smurf: A svm-based incremental anti-pattern detection approach. En *2012 19th Working Conference on Reverse Engineering*, pages 466–475. IEEE.

Maiga, A., Ali, N., Bhattacharya, N., Sabané, A., Guéhéneuc, Y., Antoniol, G., y Aimeur, E. (2012b). Support vector machines for anti-pattern detection. En *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 278–281, New York, NY, USA. ACM.

Marinescu, R. (2002). *Measurement and quality in object-oriented design*. PhD thesis, Politehnica University in Timişoara.

Marinescu, R. (2005). Measurement and quality in object-oriented design. En *21st IEEE International Conference on Software Maintenance*, pages 701–704. IEEE.

Martin, R. (2000). Design principles and design patterns. *Object Mentor*, 1(34):597.

Moha, N., Guéhéneuc, Y., Duchien, L., y Le Meur, A.-F. (2010). Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36.

Moreno, S. y Vallespir, D. (2018). ¿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. En *Conferencia Iberoamericana de Ingeniería de Software 2018*.

Murphy, M. (2014). *Vulnerabilities with custom permissions*. URL http://commonsware.com/blog/2014/02/12/vulnerabilities-custom-permissions.html. Último acceso: mayo 2020.

Nahar, N. y Sakib, K. (2015). Automatic recommendation of software design patterns using anti-patterns in the design phase: A case study on abstract factory. En *QuASoQ/WAWSE/CMCE@ APSEC*, pages 9–16.

Nahar, N. y Sakib, K. (2016). Acdpr: A recommendation system for the creational design patterns using anti-patterns. En *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, volume 4, pages 4–7.

Nayrolles, M., Moha, N., y Valtchev, P. (2013). Improving soa antipatterns detection in service based systems by mining execution traces. En *2013 20th Working Conference on Reverse Engineering*, pages 321–330. IEEE.

Ouni, A., Daagi, M., Kessentini, M., Bouktif, S., y Gammoudi, M. (2017a). A machine learning-based approach to detect web service design defects. En *IEEE International Conference on Web Services*, pages 532–539. IEEE.

Ouni, A., Kessentini, M., Inoue, K., y Cinnéide, M. (2017b). Search-based web service antipatterns detection. *IEEE Transactions on Services Computing*, 10(4):603–617.

Ouni, A., Kessentini, M., y Sahraoui, H. (2014). Multiobjective optimization for software refactoring and evolution. *Advances in Computers*, 94:103–167.

Palma, F., Nayrolles, M., Moha, N., Guéhéneuc, Y., Baudry, B., y Jézéquel, J. (2013). Soa antipatterns: An approach for their specification and detection. *International Journal of Cooperative Information Systems*, 22(4).

Palomba, F., De Lucia, A., Bavota, G., y Oliveto, R. (2014). Anti-pattern detection: Methods, challenges, and open issues. En *Advances in Computers*, volume 95, pages 201–238. Elsevier.

Paulk, M. (2006). Factors affecting personal software quality. *Institute for Software Research*, pages 9–13.

Peiris, M. y Hill, J. (2014). Towards detecting software performance anti-patterns using classification techniques. *ACM SIGSOFT Software Engineering Notes*, 39:1–4.

Peiris, M. y Hill, J. (2016). Automatically detecting.excessive dynamic memory allocations"software performance anti-pattern. En *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, pages 237–248.

Peldszus, S., Kulcsár, G., Lochau, M., y Schulze, S. (2016). Continuous detection of design flaws in evolving object-oriented programs using incremental multi-pattern matching. En *2016 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 578–589. IEEE.

Pérez, J. (2011). *Refactoring planning for design smell correction in object-oriented software*. PhD thesis, Escuela Técnica Superior de Ingeniería Informática.

Petersen, K., Feldt, R., Mujtaba, S., y Mattsson, M. (2008). Systematic mapping studies in software engineering. *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*, 17.

Pulawski, L. (2012). Automatic forecasting of design anti-patterns in software source code. En *CS&P*, pages 312–323.

Rayside, D. y Mendel, L. (2007). Object ownership profiling: a technique for finding and fixing memory leaks. En *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 194–203.

Riel, A. (1996). *Object-Oriented Design Heuristics*. Addison-Wesley.

Rombach, D., Münch, J., Ocampo, A., Humphrey, W., y Burton, D. (2008). Teaching disciplined software development. *Journal of Systems and Software*, 81(5):747–763.

Rosenberg, D. y Scott, K. (1999). *Use case driven object modeling with UML: a practical approach*. Addison-Wesley.

Salehie, M., Li, S., y Tahvildari, L. (2006). A metric-based heuristic framework to detect object-oriented design flaws. En *14th IEEE International Conference on Program Comprehension*, pages 159–168. IEEE.

Sfayhi, A. y Sahraoui, H. (2011). What you see is what you asked for: An effort-based transformation of code analysis tasks into interactive visualization scenarios. En *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, pages 195–203. IEEE.

Sharma, V. y Anwer, S. (2014). Performance antipatterns: Detection and evaluation of their effects in the cloud. En *IEEE International Conference on Services Computing*, pages 758–765. IEEE.

Siau, K. y Tan, X. (2005). Improving the quality of conceptual modeling using cognitive mapping techniques. *Data & Knowledge Engineering*, 55(3). Quality in conceptual modeling.

Sindhgatta, R., Sengupta, B., y Ponnalagu, K. (2009). Measuring the quality of service oriented design. En Baresi, L., Chi, C.-H., y Suzuki, J., editors, *Service-Oriented Computing*, pages 485–499, Berlin, Heidelberg. Springer Berlin Heidelberg.

Smith, C. y Williams, L. (2002). *Performance solutions: a practical guide to creating responsive, scalable software*, volume 23. Addison-Wesley Reading.

Smith, C. y Williams, L. (2003). More new software performance antipatterns: Even more ways to shoot yourself in the foot. En *Computer Measurement Group Conference*, pages 717–725. Citeseer.

Smith, C. U. y Williams, L. G. (2000). Software performance antipatterns. *Proceedings of the 2nd international workshop on Software and performance*, pages 127–136.

Sommerville, I. (2016). *Software Engineering*. Pearson.

Stoianov, A. y Şora, I. (2010). Detecting patterns and antipatterns in software using prolog rules. En *2010 International Joint Conference on Computational Cybernetics and Technical Informatics*, pages 253–258. IEEE.

Taenzer, D., Ganti, M., y Podar, S. (1989). Problems in object-oriented software reuse. En *ECOOP*, volume 89, pages 25–38.

Tahvildari, L. y Kontogiannis, K. (2004). Improving design quality using meta-pattern transformations: a metric-based approach. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(4/5):331–361.

Tate, B. (2002). *Bitter Java*. Manning.

Tate, B., Clark, M., y Linskey, P. (2003). *Bitter EJB*. Manning Publications Co.

Tekin, U. y Buzluca, F. (2014). A graph mining approach for detecting identical design structures in object-oriented design models. *Science of Computer Programming*, 95:406–425.

Tenenberg, J. (2005). Students designing software: a multi-national, multi-institutional study. *Informatics in Education*, 4.

Torkamani, M. y Bagheri, H. (2014). A systematic method for identification of anti-patterns in service oriented system development. *International Journal of Electrical and Computer Engineering*, 4(1):16–23.

Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J. M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., et al. (2014). Storm@ twitter. En *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 147–156.

Tourwe, T. y Mens, T. (2003). Identifying refactoring opportunities using logic meta programming. En *7th European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.*, pages 91–100.

Trifu, A., Seng, O., y Genssler, T. (2004). Automated design flaw correction in object-oriented systems. En *8th European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.*, pages 174–183. IEEE.

Trubiani, C., Bran, A., Hoorn, A., Avritzer, A., y Knoche, H. (2018). Exploiting load testing and profiling for performance antipattern detection. *Information and Software Technology*, 95:329–345.

Trubiani, C. y Koziolek, A. (2011). Detection and solution of software performance antipatterns in palladio architectural models. En *Proceedings of the 2nd ACM/SPEC International Conference on Performance engineering*, pages 19–30.

Ujhelyi, Z., Szőke, G., Horváth, Á., Csiszár, N., Vidács, L., Varró, D., y Ferenc, R. (2015). Performance comparison of query-based techniques for anti-pattern detection. *Information and Software Technology*, 65:147–165.

Vaucher, S., Khomh, F., Moha, N., y Guéhéneuc, Y. (2009). Tracking design smells: Lessons from a study of god classes. En *2009 16th Working Conference on Reverse Engineering*, pages 145–154. IEEE.

Vidal, S. A., Marcos, C., y Díaz-Pace, J. A. (2016). An approach to prioritize code smells for refactoring. *Automated Software Engineering*, 23(3):501–532.

Wang, H., Kessentini, M., Hassouna, T., y Ouni, A. (2017). On the value of quality of service attributes for detecting bad design practices. En *2017 IEEE International Conference on Web Services*, pages 341–348.

Webster, B. (1995). *Pitfalls of object-oriented development*. M & T Books.

Yamashita, A. y Moonen, L. (2013). Exploring the impact of inter-smell relations on software maintainability: An empirical study. En *35th International Conference on Software Engineering*, pages 682–691. IEEE.

ANEXOS

Anexo 1

Resultados del mapeo sistemático

En este anexo se presentan los estudios seleccionados del mapeo sistemático, ordenado por fecha de publicación ascendente. Para cada estudio se indica el año de publicación del estudio, la librería digital donde se publicó el estudio, la referencia al estudio, el tipo de contribución, el tipo de técnica de detección, la clasificación del *antipattern* y los *antipatterns* detectados. Para cada *antipattern* detectado, se presenta el nombre del *antipattern*, como lo menciona el estudio, y entre paréntesis el nombre más utilizado del *antipattern*.

Para los estudios que presentan una herramienta, se indican las principales características de la herramienta: nombre, disponibilidad, licencia y lenguaje de programación capaz de analizar la herramienta.

En el estudio presentado por Fontana et al. (2016) se presentan cuatro herramientas de detección, por lo que para cada herramienta se realiza la extracción de los datos correspondientes. Los estudios publicados por Erdemir et al. (2011) y Tekin y Buzluca (2014) utilizan la herramienta *E-Quality* para detectar diferentes *antipatterns*.

Año	Librería digital	Referencia	Tipo de contribución	Tipo de técnica de detección	Características principales de la herramienta de detección				Clasificación del *antipattern*	*Antipatterns* detectados
					Nombre	Disponibilidad	Licencia	Lenguaje de programación		
2004	Scopus, IEEEXplore	Trifu et al. (2004)	Técnica y herramienta de detección	Basado en heurística	jGoose	Herramienta propuesta en la literatura pero no disponible en línea	NA	Java	Diseño orientado a objetos	God Class (The Blob)
2005	Scopus	Kreimer (2005)	Técnica y herramienta de detección	Basado en aprendizaje automático	IYC ("It's Your Code")	Herramienta propuesta en la literatura pero no disponible en línea	-	Java	Diseño orientado a objetos	Big Class (The Blob)
2005	Scopus, IEEEXplore	Marinescu (2005)	Técnica y herramienta de detección	Basado en heurística	ProDeOOS	Herramienta en línea pero no disponible para descargar	Software propietario	Java, C++	Diseño orientado a objetos	God Class (The Blob), God Package, Misplaced Class, Inflation of Atomic Packages, Lack of bridge, Lack of strategy, Lack of state, Lack of singleton, Lack of facade (Wide Subsystem Interface), ISP Violation (Swiss Army Knife)
2006	Scopus	Salehie et al. (2006)	Técnica de detección	Basado en heurística	-	-	-	-	Diseño orientado a objetos	God Class (The Blob)
2007	ACM Digital Library	Rayside y Mendel (2007)	Técnica de detección	Basado en heurística	-	-	-	-	Rendimiento	Construction of Zombie References, Extending a mutable base class, Tangle (Tangled Ownership Contexts), Bloated Facade, Failure to Release Dormant References
2008	Scopus	Brühlmann et al. (2008)	Técnica y herramienta de detección	Basado en heurística	Metanool	Herramienta propuesta en la literatura pero no disponible en línea	-	Java	Diseño orientado a objetos	God class (The Blob), Brain class
2009	Scopus, IEEEXplore	Vaucher et al. (2009)	Técnica de detección	Basado en aprendizaje automático	-	-	-	-	Diseño orientado a objetos	God Class (The Blob)

Continua en la siguiente página

Cuadro 1.1 – *Continuación de la página anterior*

Año	Librería digital	Referencia	Tipo de contribución	Tipo de técnica de detección	Características principales de la herramienta de detección				Clasificación del *antipattern*	*Antipatterns* detectados
					Nombre	Disponibilidad	Licencia	Lenguaje de programación		
2009	Scopus, IEEEXplore	Cortellessa et al. (2009)	Técnica de detección	Basado en modelo	-	-	-	-	Rendimiento	The Blob, Traffic Jam, The Ramp, Concurrent Processing Systems, Pipe and Filter, Extensive Processing
2009	Scopus, IEEEXplore	Crasso et al. (2009)	Técnica y herramienta de detección	Basado en heurística	JEETuning Expert	Herramienta propuesta en la literatura pero no disponible en línea	-	Java	Rendimiento	Round tripping, Multiple EJB accesses per request, Redundant JNDI lookups, Sending excessive amount of objects
2010	Scopus, IEEEXplore	Stoianov y Şora (2010)	Técnica de detección	Basado en lógica relacional	-	-	-	-	Diseño orientado a objetos	The Blob, Poltergeist, Constant interface, Yoyo Problem
2010	Scopus, IEEEXplore	Moha et al. (2010)	Técnica de detección	Basado en heurística	-	-	-	-	Diseño orientado a objetos	The Blob, Functional Decomposition, Spaghetti Code, Swiss Army Knife
2010	IEEEXplore	Hassaine et al. (2010)	Técnica de detección	Basado en aprendizaje automático	-	-	-	-	Diseño orientado a objetos	The Blob, Functional Decomposition, Spaghetti Code
2011	Scopus	Budi et al. (2011)	Técnica de detección	Basado en aprendizaje automático	-	-	-	-	Arquitectura en capas	Breaking Robustness Rules, Breaking Well-Formedness Rules
2011	Scopus, IEEEXplore	Erdemir et al. (2011)	Técnica y herramienta de detección	Basado en heurística	E-Quality	Herramienta en línea pero no disponible para descargar	Software gratuito y de código abierto	Java	Diseño orientado a objetos	God class (The Blob), Brain class, Law of Demeter violation
2011	Scopus	Kessentini et al. (2011)	Técnica de detección	Basado en heurística	-	-	-	-	Diseño orientado a objetos	The Blob, Spaghetti Code, Functional Decomposition
2011	IEEEXplore	Sfayhi y Sahraoui (2011)	Técnica y herramienta de detección	Basado en modelo	Verso	Herramienta propuesta en la literatura pero no disponible en línea	-	NA	Diseño orientado a objetos	The Blob
2011	Scopus	Trubiani y Koziolek (2011)	Técnica y herramienta de detección	Basado en modelo	Extension PCM Bench	Herramienta disponible en línea para descargar e instalar	Software gratuito y de código abierto	NA	Rendimiento	The Blob, One Lane Bridge, Empty Semi Trucks, Circuitous Treasure Hunt, Traffic Jam, Concurrent Processing Systems, Pipe and Filter, Extensive Processing

Continua en la siguiente página

Cuadro 1.1 – *Continuación de la página anterior*

Año	Librería digital	Referencia	Tipo de contribución	Tipo de técnica de detección	Características principales de la herramienta de detección				Clasificación del *antipattern*	*Antipatterns* detectados
					Nombre	Disponibilidad	Licencia	Lenguaje de programación		
2012	Scopus, IEEEXplore	Maiga et al. (2012a)	Técnica de detección	Basado en aprendizaje automático	-	-	-	-	Diseño orientado a objetos	The Blob, Functional Decomposition, Spaghetti Code, Swiss Army Knife
2012	Scopus	Pulawski (2012)	Técnica de detección	Basado en aprendizaje automático	-	-	-	-	Diseño orientado a objetos	God object (The Blob), Yoyo Problem, Base bean
2013	Scopus	Palma et al. (2013)	Técnica de detección	Basado en heurística	-	-	-	-	Arquitectura orientada a servicios	Chatty Service (Chatty Web Service), Sand Pile, Nobody Home, Multiservice, Tiny Service, The Knot, Duplicated Service, Bottleneck Service, Service Chain, God Component, Bloated Service, Stovepipe Service
2013	Scopus, IEEEXplore	Nayrolles et al. (2013)	Técnica y herramienta de detección	Basado en heurística	SOMAD (Service Oriented Mining for Antipattern Detection)	Herramienta disponible en línea para descargar e instalar	Software gratuito y de código abierto	Java	Arquitectura orientada a servicios	Chatty Service (Chatty Web Service), Multiservice, Tiny Service, The Knot, Bottleneck Service, Service Chain
2014	Scopus, ACM Digital Library	Kaur y Kaur (2014)	Técnica y herramienta de detección	Basado en heurística	Antipattern Testing Tool	Herramienta propuesta en la literatura pero no disponible en línea	-	Java	Diseño orientado a objetos	The Blob, Unused code (Lava Flow)
2014	Scopus, IEEEXplore	Tekin y Buzluca (2014)	Técnica y herramienta de detección	Basado en heurística	E-Quality	Herramienta en línea pero no disponible para descargar	Software gratuito y de código abierto	Java	Diseño orientado a objetos	Software clones (Cut and Paste Structures)
2014	Scopus	Ouni et al. (2014)	Técnica de detección	Basado en heurística	-	-	-	-	Diseño orientado a objetos	The Blob, Functional Decomposition, Spaghetti Code, Swiss Army knife

Continua en la siguiente página

Cuadro 1.1 – *Continuación de la página anterior*

Año	Librería digital	Referencia	Tipo de contribución	Tipo de técnica de detección	Características principales de la herramienta de detección				Clasificación del *antipattern*	*Antipatterns* detectados
					Nombre	Disponibilidad	Licencia	Lenguaje de programación		
2014	Scopus	Torkamani y Bagheri (2014)	Técnica de detección	Basado en lista de verificación	-	-	-	-	Arquitectura orientada a servicios	Point to Point Web Services, Too many Cooks in the SOA, CRUD Interface (Maybe It is Not RPC), Chatty Interface Services (Chatty Web Service), SOA Over standardized (Standardization Paralysis), Vendor Lock In, Sand Pile, On-Line Only, Splitting Hairs, Nobody Home, UBER service
2014	Scopus, IEEEXplore	Sharma y Anwer (2014)	Técnica y herramienta de detección	Basado en heurística	PAD (Performance Antipattern Detector)	Herramienta propuesta en la literatura pero no disponible en línea	-	Java	Rendimiento	The Blob, Empty Semi Trucks, Circuitous Treasure Hunt
2014	ACM Digital Library	Peiris y Hill (2014)	Técnica de detección	Basado en aprendizaje automático	-	-	-	-	Rendimiento	One Lane Bridge
2014	ACM Digital Library	Fontana et al. (2016)	Herramienta de detección	Basado en heurística	SonarQube	Herramienta disponible en línea para descargar e instalar	Software gratuito y de código abierto/Software de código abierto y no gratuito	More than 20 programming languages, including C, Java, C#, C++, Ruby	Diseño orientado a objetos	Cyclic dependencies
2014	ACM Digital Library	Fontana et al. (2016)	Herramienta de detección	Basado en heurística	inFusion (InF)	Herramienta disponible en línea para descargar e instalar	Software propietario	Java, C, C++	Diseño orientado a objetos	Stable Abstraction Breaker, Unstable dependencies, Cyclic dependencies
2014	ACM Digital Library	Fontana et al. (2016)	Herramienta de detección	Basado en heurística	Structure101 (S101)	Herramienta disponible en línea para descargar e instalar	Software propietario	Java, .NET	Diseño orientado a objetos	Tangle, The Blob (Fat class)

Continua en la siguiente página

Cuadro 1.1 – *Continuación de la página anterior*

Año	**Librería digital**	**Referencia**	**Tipo de contribución**	**Tipo de técnica de detección**	**Características principales de la herramienta de detección**				**Clasificación del *antipattern***	***Antipatterns* detectados**
					Nombre	**Disponibilidad**	**Licencia**	**Lenguaje de programación**		
2014	ACM Digital Library	Fontana et al. (2016)	Herramienta de detección	Basado en heurística	Structural Analysis for Java (SA4J)	Herramienta en línea pero no disponible para descargar	Software gratuito y de código abierto	Java	Diseño orientado a objetos	Tangle
2015	Scopus	Bagheri et al. (2015)	Técnica y herramienta de detección	Basado en lógica relacional	Alloy Analyzer	Herramienta disponible en línea para descargar e instalar	Software gratuito y de código abierto	Alloy	Seguridad en Android	Custom permission vulnerability, URI Permission Flaw, Improper Delegation
2015	Scopus, IEEEXplore	Ganea y Marinescu (2015)	Técnica de detección	Basado en modelo	-	-	-	-	Diseño orientado a objetos	God Class (The Blob)
2015	Scopus	Nahar y Sakib (2015)	Técnica y herramienta de detección	Basado en heurística	ADPR (Antipattern based Design Pattern Recommender)	Herramienta en línea pero no disponible para descargar	Software gratuito y de código abierto	Java	Diseño orientado a objetos	Missing Abstract Factory
2015	IEEEXplore	Hecht et al. (2015)	Técnica de detección	Basado en aprendizaje automático	-	-	-	-	Diseño orientado a objetos	God Class (The Blob)
2015	ACM Digital Library	Arcelli et al. (2015)	Técnica de detección	Basado en modelo	-	-	-	-	Rendimiento	The Blob
2016	Scopus, IEEEXplore	Bersani et al. (2016)	Técnica y herramienta de detección	Basado en modelo	OSTIA (On-the-fly Static Topology Inference Analysis)	Herramienta disponible en línea para descargar e instalar	Software gratuito y de código abierto	Java	Grandes volúmenes de datos	Multi-Anchoring, Cycle-in Topology, Persistent Data, Computation Funnel
2016	IEEEXplore	Nahar y Sakib (2016)	Técnica y herramienta de detección	Basado en heurística	ACDPR (Antipattern based Creational Design Pattern Recommender)	Herramienta propuesta en la literatura pero no disponible en línea	-	Java	Diseño orientado a objetos	Missing Abstract Factory, Missing Factory Method, Missing Builder, Missing Prototype, Missing Singleton (Lack of singleton)

Continua en la siguiente página

Cuadro 1.1 – *Continuación de la página anterior*

Año	Librería digital	Referencia	Tipo de contribución	Tipo de técnica de detección	Características principales de la herramienta de detección				Clasificación del *antipattern*	*Antipatterns* detectados
					Nombre	Disponibilidad	Licencia	Lenguaje de programación		
2016	Scopus, IEEEXplore, ACM Digital Library	Peldszus et al. (2016)	Técnica y herramienta de detección	Basado en heurística	HULK	Herramienta disponible en línea para descargar e instalar	Software gratuito y de código abierto	Java	Diseño orientado a objetos	The Blob, Swiss Army Knife, Spaghetti Code
2016	Scopus, IEEEXplore	Aras y Selçuk (2016)	Técnica de detección	Basado en heurística	-	-	-	-	Diseño orientado a objetos	The Blob, Swiss Army Knife, Lava Flow
2016	Scopus, ACM Digital Library	Peiris y Hill (2016)	Técnica y herramienta de detección	Basado en heurística	EMAD (Excessive Memory Detector de asignación)	Herramienta propuesta en la literatura pero no disponible en línea	-	C, C++	Rendimiento	Excessive dynamic memory allocation
2016	Scopus, IEEEXplore	Ouni et al. (2017b)	Técnica de detección	Basado en heurística	-	-	-	-	Arquitectura orientada a servicios	Maybe It is Not RPC (CRUD Interface), Chatty Web Service, God object Web service (Multiservice)
2017	Scopus	De Sanctis et al. (2017)	Técnica y herramienta de detección	Basado en modelo	panda-aemilia	Herramienta disponible en línea para descargar e instalar	Software gratuito y de código abierto	-	Rendimiento	The Blob, One Lane Bridge, Traffic Jam, The Ramp, Pipe and Filter, Extensive Processing
2017	Scopus	Ouni et al. (2017a)	Técnica de detección	Basado en aprendizaje automático	-	-	-	-	Arquitectura orientada a servicios	CRUDy Interface (Maybe It is Not RPC), Chatty Web Service, Multiservice
2017	IEEEXplore	Wang et al. (2017)	Técnica de detección	Basado en heurística	-	-	-	-	Arquitectura orientada a servicios	CRUDy Interface (Maybe It is Not RPC), Chatty Web Service, God object Web service (Multiservice)
2018	Scopus	Arcelli et al. (2018)	Técnica de detección	Basado en modelo	-	-	-	-	Rendimiento	The Blob, Empty Semi Trucks, Concurrent Processing Systems, Pipe and Filter, Extensive Processing, Tower of Babel
2018	Scopus	Trubiani et al. (2018)	Técnica y herramienta de detección	Basado en heurística	PADprof	-	-	Java	Rendimiento	Circuitous Treasure Hunt, Extensive Processing

Cuadro 1.1: Estudios seleccionados del mapeo sistemático

Anexo 2

Catálogo de *antipatterns*

En este anexo se presenta el catálogo de *antipatterns* construido durante la revisión de la literatura, ordenado alfabéticamente por el nombre del *antipattern*.

Name Base Bean

Description Derived classes should be substitutable for their base classes. That is, a user of a base class should continue to function properly if a derivative of that base class is passed to it. Apply inheritance only to be able to use its methods is not only a bad design solution but it violates the Liskov Substitution Principle (Martin, 2000). The Liskov Substitution Principle (LSP) states: "Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing".

References Pulawski (2012), Martin (2000)

Classification Object Oriented Design

Also know as -

References to techniques Pulawski (2012)

Type of detection techniques Machine learning-based

Name Bloated Facade

Description The facade design pattern provides a unified interface to a subsystem. A facade may be considered bloated under the following conditions: it is dynamically allocated; it refers to some large, dynamically allocated, substructures, in addition to smaller substructures. It has clients "C" that only need some of the smaller structures; the clients "C" live longer than some other clients "D" that use the larger substructures. In cases such as this, the long-living clients C cause the larger substructures to become junk.

References Rayside y Mendel (2007)

Classification Performance

Also know as -

References to techniques Rayside y Mendel (2007) **Type of detection techniques** Heuristic-based

Name Bloated Service

Description Is an antipattern related to service implementation where services in Service Oriented Architecture become "blobs" (The Blob) with one large interface and lots of parameters. Bloated Service performs heterogeneous operations with low cohesion among them. It results in a system with less maintainability, testability, and reusability within other business processes. It requires the consumers to be aware of many details (i.e. parameters) to invoke or customize them.

References Palma et al. (2013)

Classification Service Oriented Architecture

Also know as -

References to techniques Palma et al. (2013) **Type of detection techniques** Heuristic-based

Name Bottleneck Service

Description Is a service that is highly used by other services or clients. It has a high incoming and outgoing coupling. Its response time can be high because it may be used by too many external clients, for which clients may need to wait to get access to the service. Moreover, its availability may also be low due to the traffic.

References Nayrolles et al. (2013), Palma et al. (2013)

Classification Service Oriented Architecture

Also know as -

References to techniques Nayrolles et al. (2013), Palma et al. (2013) **Type of detection techniques** Heuristic-based

Name of the tools SOMAD **User Licenses** Free and open-source software

Programming languages Java

Name Brain Class

Description This design disharmony is about complex classes that tend to accumulate an excessive amount of intelligence, usually in the form of several methods affected by "Brain method" (Lanza y Marinescu, 2006).
This recalls The Blob disharmony, because those classes also have the tendency to centralize the intelligence of the system. It looks like the two disharmonies are quite similar. This is partially true, because both refer to complex classes. Yet the two problems are distinct. The fingerprint of "The Blob" is not just its complexity, but the fact that the class relies for part of its behavior on encapsulation breaking, as it directly accesses many attributes from other classes. On the other hand, the "Brain Class" detection strategy is trying to complement "The Blob" strategy by catching those excessively complex classes that are not detected as "blob classes" either because they do not abusively access data of "satellite" classes, or because they are a little more cohesive.

References Lanza y Marinescu (2006)

Classification Object Oriented Design

Also know as -

References to techniques Erdemir et al. (2011), Brühlmann et al. (2008) **Type of detection techniques** Heuristic-based

Name of the tools Metanool, E-quality **User Licenses** Free and open-source software

Programming languages Java

Name Breaking Robustness Rules

Description Robustness analysis provides a set of rules that indicate all valid and invalid interactions among different class stereotypes (Rosenberg y Scott, 1999). The rules are paraphrased as follows, where actors represent users of a system which could be humans or classes/objects outside the system under analysis.
The rules are as follows: Actors can only call boundary objects; Boundary objects can only call controllers or actors; Entity objects can only call controllers; Controllers can call entities, other controllers, andboundaries, but not actors.

References Budi et al. (2011)

Classification Layer Architecture

Also know as -

References to techniques Budi et al. (2011)

Type of detection techniques Machine learning-based

Name Breaking Well-Formedness Rules

Description The well-formedness rules (Corporation, 1997) are rephrased as follows: Actors can only call boundary objects; Boundary objects can call entities, controllers, otherboundaries, and actors; Entity objects can only call other entities; Controllers can call entities, other controllers, andboundaries, but not actors.

References Budi et al. (2011)

Classification Layer Architecture

Also know as -

References to techniques Budi et al. (2011)

Type of detection techniques Machine learning-based

Name Chatty Web Service

Description The Web Service is defined with many finegrained operations, and clients end up invoking many operations in sequence to perform an overall process. The operations may be supporting CRUD-type (Create, Read, Update and Delete) requirements and be manifested as attribute-level accessors or mutators. Alternately, the operations may each represent a very fine-grained step within an overall process sequence.

References Dudney et al. (2002)

Classification Service Oriented Architecture

Also know as Fine-Grained Interface, Fine-Grained Service, Chatty Interface, Chatty Service

References to techniques Ouni et al. (2017a), Ouni et al. (2017b), Wang et al. (2017), Torkamani y Bagheri (2014), Nayrolles et al. (2013), Palma et al. (2013)

Type of detection techniques Heuristic-based, Machine learning-based, Checklist-based

Name of the tools SOMAD

User Licenses Free and open-source software

Programming languages Java

Name Circuitous Treasure Hunt

Description The impact on performance is the large amount of database processing required each time the "ultimate results" are needed. It is especially problematic when the data is on a remote server, and each access requires transmitting all the intermediate queries and their results via a network, and perhaps through middleware and other servers in a multitier environment.

References Smith y Williams (2000)

Classification Performance

Also know as -

References to techniques Trubiani et al. (2018), Sharma y Anwer (2014), Trubiani y Koziolek (2011)

Type of detection techniques Heuristic-based, Model-based

Name of the tools PAD, Extension PCM Bench, PADprof

User Licenses Free and open-source software

Programming languages Java

Name Computation Funnel

Description Storm is a technology developed at Twitter (Toshniwal et al., 2014) in order to face the problem of processing of streaming of data. A Storm topology is a computational graph composed by nodes of two types: spouts and bolts, where spouts are nodes that have outgoing edges only and bolts are nodes that can have either incoming or outgoing edges respectively. A computational funnel emerges when there is not a path from data source (spout) to the bolts that sends out the tuples off the topology to another topology through a messaging framework or through a storage. This circumstance should be dealt with since it may compromise the availability of results under the desired performance restrictions.

References Bersani et al. (2016)

Classification Big Data

Also know as -

References to techniques Bersani et al. (2016)

Type of detection techniques Model-based

Name of the tools OSTIA

User Licenses Free and open-source software

Programming languages Java

Name Concurrent Processing Systems

Description Occurs when processing cannot make use of available processors. This antipattern represents a manifestation of the "Unbalanced Processing" antipattern (Smith y Williams, 2003). It occurs when processes cannot make effective use of available processors either because of: (i) a non-balanced assignment of tasks to processors; (ii) single-threaded code.

References Smith y Williams (2003)

Classification Performance

Also know as -

References to techniques Arcelli et al. (2018), Trubiani y Koziolek (2011), Cortellessa et al. (2009)

Type of detection techniques Model-based

Name of the tools Extension PCM Bench

User Licenses Free and open-source software

Programming languages NA

Name Constant interface

Description It is a poor use of interface which is occurred when declaring a constant in the interface without any methods. That is the interface class should be used to refer to instances of classes and it is inappropriate to be used for any other purpose.

References Bloch y Steele (2001)

Classification Object Oriented Design

Also know as -

References to techniques Stoianov y Şora (2010) **Type of detection techniques** Relational logic-based

Name Construction of Zombie References

Description Is a reference to an object that is already junk. Unlike a Failure to Release Dormant References, a "Zombie Reference" is never used. The solution is to not create the "Zombie Reference" in the first place.

References Rayside y Mendel (2007)

Classification Performance

Also know as -

References to techniques Rayside y Mendel (2007) **Type of detection techniques** Heuristic-based

Name Custom permission vulnerability

Description If two applications define the same custom permission (via the "permission" element in the manifest, with the same "android:name" value), whichever app is installed first is the one whose definition is used. This, it does open up significant risks. An attacker can gain access to a defender's secured components simply by: declaring the same custom permission, requesting that custom permission via "uses-permission", and being installed first.

References Murphy (2014)

Classification Security in Android

Also know as -

References to techniques Bagheri et al. (2015)

Type of detection techniques Relational logic-based

Name of the tools Alloy Analyzer

User Licenses Free and open-source software

Programming languages Alloy

Name Cut and Paste Structures

Description This antipattern is identified by the presence of several similar segments of code interspersed throughout the software project. Usually, the project contains many programmers who are learning how to develop software by following the examples of more experienced developers. However, they are learning by modifying code that has been proven to work in similar situations, and potentially customizing it to support new data types or slightly customized behavior. This creates code duplication (code smell: "Duplicated Code"), which may have positive short-term consequences such as boosting line count metrics, which may be used in performanc se evaluations. Furthermore, it is easy to extend the code as the developer has full control over the code used in his or her application and can quickly meet short-term modifications to satisfy new requirements.

References Brown et al. (1998), Baker (1995)

Classification Object Oriented Design

Also know as Software design clones, Software clones, Reused design structures between projects, Copy paste modify structures, Identical design structures, Clipboard Coding, Software Cloning, Software Propagation

References to techniques Tekin y Buzluca (2014)

Type of detection techniques Heuristic-based

Name of the tools E-Quality

User Licenses Free and open-source software

Programming languages Java

Name Cycle-in Topology

Description Storm is a technology developed at Twitter (Toshniwal et al., 2014) in order to face the problem of processing of streaming of data. A Storm topology is a computational graph composed by nodes of two types: spouts and bolts, where spouts are nodes that have outgoing edges only and bolts are nodes that can have either incoming or outgoing edges respectively. Technically, it is possible to have cycle in "Storm" topologies. An infinite cycle of processing would create an infinite tuple tree and make it impossible for Storm to ever acknowledge spout emitted tuples. Therefore, cycles should be avoided or resulting tuple trees should be investigated additionally to make sure they terminate at some point and under a specified series of conditions. The antipattern itself may lead to infrastructure overloading and increased costs.

References Bersani et al. (2016)

Classification Big Data

Also know as -

References to techniques Bersani et al. (2016)

Type of detection techniques Model-based

Name of the tools OSTIA

User Licenses Free and open-source software

Programming languages Java

Name Cyclic Dependency

Description Refers to a subsystem that is involved in a chain of relations that break the desirable acyclic nature of a subsystems dependency structure.

References Lakos (1996)

Classification Object Oriented Design

Also know as -

References to techniques Fontana et al. (2016)

Type of detection techniques Heuristic-based

Name of the tools SonarQube, inFusion

User Licenses Free and open-source software, Open-source and not free software, Proprietary software

Programming languages More than 20 programming languages, including C, C++, Java, C#, C++, Ruby

Name Duplicated Service

Description Corresponds to a set of highly similar services. Because services are implemented multiple times as a result of the silo approach (it occurs when several departments or groups within an organization do not want to share information or knowledge with other individuals in the same organization), there may have common or identical methods with the same names and–or parameters

References Cherbakov et al. (2006)

Classification Service Oriented Architecture

Also know as -

References to techniques Palma et al. (2013) **Type of detection techniques** Heuristic-based

Name Empty Semi Trucks

Description Occurs in software systems where an excessive number of requests is required to perform a task, such as retrieval of information from a database. The problem may be due to inefficient use of available bandwidth, an inefficient interface, or both.
This manifestation of the antipattern occurs in message-based systems when a very small amount of information is sent in each message. The amount of processing overhead is the same regardless of the size of the message. With smaller messages, this processing is required many more times than necessary.

References Smith y Williams (2003)

Classification Performance

Also know as -

References to techniques Arcelli et al. (2018), Sharma y Anwer (2014), Trubiani y Koziolek (2011) **Type of detection techniques** Heuristic-based, Model-based

Name of the tools PAD, Extension PCM Bench **User Licenses** Free and open-source software

Programming languages Java

Name Excessive dynamic memory allocation

Description Occurs when an application unnecessarily creates and destroys large numbers of objects during its execution. The overhead required to create and destroy these objects has a negative impact on performance.

References Smith y Williams (2000)

Classification Performance

Also know as -

References to techniques Peiris y Hill (2016)

Type of detection techniques Heuristic-based

Name of the tools EMAD

User Licenses -

Programming languages C, C++

Name Extending a mutable base class

Description By extending a mutable base class, the programmer may unwittingly inherit unnecessary fields that retain unnecessary references to other objects. The fix is usually to implement an appropriate interface instead.

References Rayside y Mendel (2007)

Classification Performance

Also know as -

References to techniques Rayside y Mendel (2007)

Type of detection techniques Heuristic-based

Name Extensive Processing

Description Occurs when extensive processing in general impedes overall response time. This antipattern represents a manifestation of the "Unbalanced Processing" antipattern (Smith y Williams, 2003). It occurs when a long running process monopolizesa processor and prevents a set of other jobs to be executed until it finishes its computation. The processor is removed from the pool, but unlike the "pipe and filter" antipattern, other work does not have to pass through this stage before proceeding. This is particularly problematic if the extensive processing is on the processing path that is executed for the most frequent workload.

References Smith y Williams (2003)

Classification Performance

Also know as -

References to techniques Arcelli et al. (2018), Trubiani et al. (2018), De Sanctis et al. (2017), Trubiani y Koziolek (2011), Cortellessa et al. (2009)

Type of detection techniques Heuristic-based, Model-based

Name of the tools Extension PCM Bench, panda-aemilia, PADprof

User Licenses Free and open-source software

Programming languages Java

Name Failure to Release Dormant References

Description Is a reference created when its target was active, but that persists after its target becomes junk. This is the most common and well-known memory management antipattern.

References Tate (2002)

Classification Performance, Object Oriented Design

Also know as Leaking listeners

References to techniques Rayside y Mendel (2007)

Type of detection techniques Heuristic-based

Name Functional Decomposition

Description Is the result of experienced, nonobject-oriented developers who design and implement an application in an object-oriented language. When developers are comfortable with a "main" routine that calls numerous subroutines, they may tend to make every subroutine a class, ignoring class hierarchy altogether (and pretty much ignoring object orientation entirely). The resulting code resembles a structural language such as Pascal or Fortran in class structure. It can be incredibly complex, as smart procedural developers devise very clever ways to replicate their time-tested methods in an object-oriented architecture.

References Brown et al. (1998), Akroyd (1996)

Classification Object Oriented Design

Also know as No Object Oriented AntiPattern

References to techniques Ouni et al. (2014), Maiga et al. (2012a), Kessentini et al. (2011), Hassaine et al. (2010), Moha et al. (2010)

Type of detection techniques Heuristic-based, Machine learning-based

Name God Component

Description Occurs when a component encapsulates a multitude of services. This component represents high responsibility enclosed by many methods with many different types of parameters to exchange. It may have a high coupling with the communicating services. Being at the component-level, God Component is at a higher level of abstraction than the Multiservice, which is at the service-level, and usually aggregates a set of services.

References Palma et al. (2013)

Classification Service Oriented Architecture

Also know as -

References to techniques Palma et al. (2013)

Type of detection techniques Heuristic-based

Name God Package

Description The packages tend to become very large and non-cohesive. Another symptom of this flaw is the large number of clients of the package (i.e. classes from other packages) that use excessively this package. The consequence is that clients of this package must check out everything in that package even if that change does not affect them in any way.

References Marinescu (2002)

Classification Object Oriented Design

Also know as -

References to techniques Marinescu (2005)

Type of detection techniques Heuristic-based

Name of the tools ProDeOOS

User Licenses Proprietary software

Programming languages Java, C++

Name Improper Delegation

Description A malicious application may be able to indirectly invoke a component, without having a permission to do so, by interacting with third component that possess the permission.

References Bagheri et al. (2015)

Classification Security in Android

Also know as -

References to techniques Bagheri et al. (2015)

Type of detection techniques Relational logic-based

Name of the tools Alloy Analyzer

User Licenses Free and open-source software

Programming languages Alloy

Name Inflation of Atomic Packages

Description The packages tend to become very small, almost "atomic". In contrast to the God Package flaw, clients of packages affected by this flaw tend to use classes from a large number of other packages.

References Marinescu (2002)

Classification Object Oriented Design

Also know as -

References to techniques Marinescu (2005)

Type of detection techniques Heuristic-based

Name of the tools ProDeOOS

User Licenses Proprietary software

Programming languages Java, C++

Name Lack of Bridge

Description Bridge design pattern (Gamma et al., 1994) is used to decouple an abstraction from its implementation, so that the two may vary independently. When we have several implementations for a certain abstraction, the usual solution is to use inheritance: an abstract class defines the interface and concrete subclasses provide the implementations. However, this solution is inflexible because the abstraction and its implementations are statically linked. Thus, the following problems may appear: (i) if we have a lot of abstractions, organized in a rich hierarchy, by adding the implementation subclasses this hierarchy becomes very complex; (ii) because of the static link between abstraction and implementation, modifying the abstraction implies the modification of all its implementations; (iii) when we add a new abstraction, it is necessary to write all the subclasses that provide the implementations for this abstraction.

References Marinescu (2002)

Classification Object Oriented Design

Also know as -

References to techniques Marinescu (2005)

Type of detection techniques Heuristic-based

Name of the tools ProDeOOS

User Licenses Proprietary software

Programming languages Java, C++

Name Lack of Singleton

Description In proper application of Singleton design pattern (Gamma et al., 1994), singleton class has the responsibility of keeping one single instance of it, ensuring central control to object creation. In this antipattern, a conditional checking is performed before instantiating the singleton class every time by the classes that require the singleton object.

References Marinescu (2002), Nahar y Sakib (2016)

Classification Object Oriented Design

Also know as Missing Singleton

References to techniques Marinescu (2005), Nahar y Sakib (2016)

Type of detection techniques Heuristic-based

Name of the tools ProDeOOS, ACDPR

User Licenses Proprietary software

Programming languages Java, C++

Name Lack of State

Description The state design pattern (Gamma et al., 1994) allow an object to alter its behavior when its internal state changes. The object will appear to change its class. Thus, the following problem may appear: large conditional statements, which are monolithic and tend to make the code less explicit, whichin turn makes them difficult to modify and extend. The State patternoffers a better way to structure state-specific code. The logic thatdetermines the state transitions doesn't reside in monolithicif or switch statements but instead is partitionedbetween the State subclasses.

References Marinescu (2002)

Classification Object Oriented Design

Also know as -

References to techniques Marinescu (2005)

Type of detection techniques Heuristic-based

Name of the tools ProDeOOS

User Licenses Proprietary software

Programming languages Java, C++

Name Lack of Strategy

Description When we need several algorithms to solve a certain problem, the Strategy design pattern (Gamma et al., 1994) offers the solution of defining a family of algorithms; these algorithms are encapsulated and organized in a class hierarchy, so they can be easily interchanged. Thus, the following problems may appear: (i) All the algorithms are encapsulated in one big class; this class will be very complex, with some methods having large conditional structures, namely those methods in which the algorithms are implemented; (ii) another possibility is the appearance of a class hierarchy (probably shallow and wide), where each subclass offers a different implementation of the algorithm. It is very likely that the only difference between the base class and its subclasses the overriding of the methods that implement the algorithm.

References Marinescu (2002)

Classification Object Oriented Design

Also know as -

References to techniques Marinescu (2005)

Type of detection techniques Heuristic-based

Name of the tools ProDeOOS

User Licenses Proprietary software

Programming languages Java, C++

Name Lava Flow

Description Is commonly found in systems that originated as research but ended up in production. It is characterized by the lava like "flows" of previous developmental versions strewn about the code landscape, which have now hardened into a basaltlike, immovable, generally useless mass of code that no one can remember much. This is the result of earlier (perhaps Jurassic) developmental times when, while in a research mode, developers tried out several ways of accomplishing things, typically in a rush to deliver some kind of demonstration, there by casting sound design practices to the winds and sacrificing documentation. The result is several fragments of code, wayward variable classes, and procedures that are not clearly related to the overall system. In fact, these flows are often so complicated in appearance and spaghetti like that they seem important, but no one can really explain what they do or why they exist.

References Brown et al. (1998)

Classification Object Oriented Design

Also know as Dead code, Unused code

References to techniques Aras y Selçuk (2016), Kaur y Kaur (2014)

Type of detection techniques Heuristic-based

Name of the tools Antipattern Testing Tool

User Licenses -

Programming languages Java

Name Law of Demeter violation

Description Demeter's law is known as "don't talk to strangers" because: (i) each unit should have only limited knowledge about other units (only units "closely" related to the current unit); (ii) each unit should only talk to its friends (don't talk to strangers); (iii) only talk to your immediate friends.

References Larman (1998)

Classification Object Oriented Design

Also know as -

References to techniques Erdemir et al. (2011)

Type of detection techniques Heuristic-based

Name of the tools E-quality

User Licenses Free and open-source software

Programming languages Java

Name Maybe It is Not RPC

Description In this antipattern, a Web Service implements one or more RPC-style (Remote Procedure Call implies that SOAP (Simple Object Access Protocol) body contains an element with the name of the method or operation being invoked; this element in turn contains an element for each parameter of that method/operation) operations that essentially support CRUD-type (Create, Read, Update and Delete) actions for significant business entities, for example, "createOrder" or "updateInvoice". These operations will likely need to specify a significant number of parameters and/or complexity in those parameters (for example, custom types) to facilitate the passing of the necessary attributes of a business entity. Alternately, a document exchange process may be implemented through RPC by attaching the document as a SOAP attachment. Essentially, the approach is still RPC, but all the operation-specific data is being passed as an XML attachment.

References Dudney et al. (2002)

Classification Service Oriented Architecture

Also know as CRUD Interface, CRUDy Interface, Procedural Programming

References to techniques Wang et al. (2017), Ouni et al. (2017a), Ouni et al. (2017b), Torkamani y Bagheri (2014)

Type of detection techniques Heuristic-based, Machine learning-based, Checklist-based

Name Misplaced Class

Description The detection strategy is based on the principles of package cohesion (Martin, 2000): (i) Number of classes from other packages on which the measured class depends on; (ii) relative number of dependencies that a class has on its own package; (iii) number of other packages on which a class depends.

References Marinescu (2002)

Classification Object Oriented Design

Also know as -

References to techniques Marinescu (2005)

Type of detection techniques Heuristic-based

Name of the tools ProDeOOS

User Licenses Proprietary software

Programming languages Java, C++

Name Missing Abstract Factory

Description This antipattern arises when groups or family of classes are instantiated directly without using factories for the instantiation. Thus, the following problems may appear: (i) clients depend on a fixed family of products; (ii) hard to introduce a new family; (iii) no restriction to use products from diferent families.

References Nahar y Sakib (2015), Jebelean (2004)

Classification Object Oriented Design

Also know as -

References to techniques Nahar y Sakib (2016), Nahar y Sakib (2015)

Type of detection techniques Heuristic-based

Name of the tools ADPR

User Licenses -

Programming languages Java

Name Missing Builder

Description Is concerned with single class rather than multiple ones. So, despite covering the whole class relationships structure, it requires to analyze the internal structure of classes individually. The structure is, there are multiple constructors of a class with different parameters. Existence of these multiple different parameterized constructors assure that the single object has different representations revealing the potential of using Builder pattern. The behavior lies in the parameter lists of the multi-constructor classes, where with inclusion of every constructor, a new parameter is introduced. This behavior shows that the object can be divided into individual parts based on those parameters, as required by Builder.

References Nahar y Sakib (2016)

Classification Object Oriented Design

Also know as Telescoping Constructor

References to techniques Nahar y Sakib (2016) **Type of detection techniques** Heuristic-based

Name of the tools ACDPR **User Licenses** -

Programming languages Java

Name Missing Factory Method

Description This antipattern is similar to the Missing Abstract Factory, but with absence of class families. In "Missing Factory Method", different classes are instantiated in different execution paths (conditional instantiation). Unlike "Missing Abstract Factory", where families of classes are instantiated in different execution paths, here only single classes are instantiated in those execution paths. The behavioral matching is also similar to "Missing Abstract Factory". The identified classes from lifelines of each sequence diagram are marked to be in the same execution paths. Classes instantiated in different execution paths are of the same type in "Missing Factory Method".

References Nahar y Sakib (2016)

Classification Object Oriented Design

Also know as -

References to techniques Nahar y Sakib (2016) **Type of detection techniques** Heuristic-based

Name of the tools ACDPR **User Licenses** -

Programming languages Java

Name Missing Prototype

Description The missing Prototype basically reveals the usage of multiple instances of same objects, which recommends the cloning of those objects. Similar to Builder, semantic of missing Prototype is not required. The prototype classes contain clone methods to copy instances of own-selves, that can be invoked by other classes. A clone method is called by another class only when it requires multiple instances of the prototype class. Thus, the structure focuses on these other classes to find whether multiple variables of a class type are present. The structural matching is done by checking if a class contains multiple variables having another class type. As structure defines, in Prototype, a class must have multiple variables having the type of prototype class. The behavior verifies that, truly prototype classes are instantiated in those variables (in case of inheritance, sub-classes can get instantiated in a variable of super-class type).

References Nahar y Sakib (2016)

Classification Object Oriented Design

Also know as -

References to techniques Nahar y Sakib (2016) **Type of detection techniques** Heuristic-based

Name of the tools ACDPR **User Licenses** -

Programming languages Java

Name Multi-Anchoring

Description Storm is a technology developed at Twitter (Toshniwal et al., 2014) in order to face the problem of processing of streaming of data. A Storm topology is a computational graph composed by nodes of two types: spouts and bolts, where spouts are nodes that have outgoing edges only and bolts are nodes that can have either incoming or outgoing edges respectively. In order to guarantee fault-tolerant stream processing, tuples processed by bolts need to be anchored with the unique id of the bolt and be passed to multiple acknowledgers (or "ackers" in short) in the topology. In this way, ackers can keep track of tuples in the topology. Multiple ackers can indeed cause much overhead and influence the operational performance of the entire topology.

References Bersani et al. (2016)

Classification Big Data

Also know as -

References to techniques Bersani et al. (2016) **Type of detection techniques** Model-based

Name of the tools OSTIA **User Licenses** Free and open-source software

Programming languages Java

Name Multiple EJB accesses per request

Description The multiple EJB (Enterprise Java Beans) accesses per request antipattern is considered a special case of Round tripping, which requires that, at least, one access is intended for modifying the "bean". This causes many remote calls.

References Crasso et al. (2009)

Classification Performance

Also know as -

References to techniques Crasso et al. (2009)

Type of detection techniques Heuristic-based

Name of the tools JEETuning Expert

User Licenses -

Programming languages Java

Name Multiservice

Description A multiservice is a service with a large number of public interface methods that implement processes related to many different key abstractions or entities. The result is a service that is coupling to many underlying business or technical components, and also is utilized by many clients, making development and maintenance difficult, and reducing flexibility and reuse.

References Dudney et al. (2002)

Classification Service Oriented Architecture

Also know as The God Object, God object Web service , Big Ball of Mud

References to techniques Ouni et al. (2017a), Ouni et al. (2017b), Wang et al. (2017), Nayrolles et al. (2013), Palma et al. (2013)

Type of detection techniques Heuristic-based, Machine learning-based

Name of the tools SOMAD

User Licenses Free and open-source software

Programming languages Java

Name Nobody Home

Description Nobody Home corresponds to a service, defined but actually never used by clients. Thus, the methods from this service are never invoked, even though it may be coupled to other services. But still they require deployment and management, despite of their no usage.

References Jones (2006)

Classification Service Oriented Architecture

Also know as -

References to techniques Torkamani y Bagheri (2014), Palma et al. (2013) **Type of detection techniques** Heuristic-based, Checklist-based

Name On-Line Only

Description Practice indicates that there are frequent situations when some parts of the system must or should be run in batch mode. The main reasons are: (i) some legacy systems are batch systems; (ii) some activities require a lot of time (as they are computationally complex) or need real-world (i.e. user) responses; (iii) there can be software engineering reasons to use batch systems (reduction of development effort, security, etc.). So the integration of batch systems is necessary. The integration can be via data stores implemented as services. There is a prejudice that it is an obsolete technique. This antipattern results into expensive and unstable solutions.

References Kral y Zemlicka (2007)

Classification Service Oriented Architecture

Also know as -

References to techniques Torkamani y Bagheri (2014) **Type of detection techniques** Checklist-based

Name One Lane Bridge

Description Occurs at a point in execution where only one, or a few, processes may continue to execute concurrently (i.e. when accessing a database). Other processes are delayed while they wait for their turn.

References Smith y Williams (2000)

Classification Performance

Also know as -

References to techniques De Sanctis et al. (2017), Peiris y Hill (2014), Trubiani y Koziolek (2011)

Type of detection techniques Machine learning-based, Model-based

Name of the tools Extension PCM Bench, panda-aemilia

User Licenses Free and open-source software

Programming languages Java

Name Persistent Data

Description Storm is a technology developed at Twitter (Toshniwal et al., 2014) in order to face the problem of processing of streaming of data. A Storm topology is a computational graph composed by nodes of two types: spouts and bolts, where spouts are nodes that have outgoing edges only and bolts are nodes that can have either incoming or outgoing edges respectively. This antipattern covers the circumstance wherefore if two processing elements need to update a same entity in a storage, there should be a consistency mechanism in place.

References Bersani et al. (2016)

Classification Big Data

Also know as -

References to techniques Bersani et al. (2016)

Type of detection techniques Model-based

Name of the tools OSTIA

User Licenses Free and open-source software

Programming languages Java

Name Pipe and Filter

Description Occurs when the slowest filter in a "pipe and filter" architecture causes the system to have unacceptable throughput. This antipattern represents a manifestation of the "Unbalanced Processing" antipattern (Smith y Williams, 2003). It occurs when the throughput of the overall systemis jeopardized by the slowest filter. This is due to a stage in a pipeline, whichis significantly slower than all the others, therefore constituting a bottleneckin the whole process in which most stages have to wait the slowest one toterminate.

References Smith y Williams (2003)

Classification Performance

Also know as -

References to techniques Arcelli et al. (2018), De Sanctis et al. (2017), Trubiani y Koziolek (2011), Cortellessa et al. (2009)

Type of detection techniques Model-based

Name of the tools Extension PCM Bench, panda-aemilia

User Licenses Free and open-source software

Programming languages NA

Name Point to Point Web Services

Description Web Service calls are invoked directly, using a URI which is hard-coded into the WSDL. After a period of time the multiple web service calls and dependencies lead to the Spaghetti infrastructure. When one service is changed a number of other services either fail or behave in unpredictable ways. There is a lack of clarity as to how one service depends on another and what the impact of change is across these services.

References Jones (2006)

Classification Service Oriented Architecture

Also know as -

References to techniques Torkamani y Bagheri (2014)

Type of detection techniques Checklist-based

Name Poltergeist

Description Poltergeists are classes with limited responsibilities and roles to play in the system; therefore, their effective life cycle is quite brief. Poltergeists clutter software designs, creating unnecessary abstractions; they are excessively complex, hard to understand, and hard to maintain.

References Brown et al. (1998), Fowler (1997), Akroyd (1996), Riel (1996)

Classification Object Oriented Design

Also know as Gypsy, Proliferation of Classes, Big DoIt Controller Class

References to techniques Stoianov y Şora (2010) **Type of detection techniques** Relational logic-based

Name Redundant JNDI lookups

Description Communication overhead when there are redundant accesses to a JNDI (Java Naming and Directory Interface) server. Every time a JEE (Java Enterprise Edition) client needs a service, the former must access to the JNDI server. Clients use a JNDI server to look up available services, such as EJBs, JDBC (Java Data Base Connectivity) data sources and JMS (Java Message Service) connection factories, from anywhere in the network. The lookup process is expensive because clients usually run on a host different from the host that executes the JNDI server.

References Crasso et al. (2009)

Classification Performance

Also know as -

References to techniques Crasso et al. (2009) **Type of detection techniques** Heuristic-based

Name of the tools JEETuning Expert **User Licenses** -

Programming languages Java

Name Round tripping

Description A major cause of poor performance in distributed applications is a large communication overhead between the distributed objects. One of the main characteristics of remote entity/session beans is that other distributed components can get access to them as if they were deployed at the same host. The straightest way for retrieving the details of a bean is through its "getters" methods, nevertheless this requires multiple fine grained remote calls. In the context of JEE (Java Enterprise Edition) applications, a careless use of EJB (Enterprise Java Beans) technology renders too many occurrences of this antipattern at the expense of the overall system performance. Round tripping can be seen as a special case of the well-known Empty Semi Trucks antipattern, which occurs when an excessive number of quests is required to perform a task due to inefficient use of available bandwidth.

References Tate et al. (2003)

Classification Performance

Also know as -

References to techniques Crasso et al. (2009) **Type of detection techniques** Heuristic-based

Name of the tools JEETuning Expert **User Licenses** -

Programming languages Java

Name Sand Pile

Description A frequent implementation of SOA is the technique "one elementary service (i.e. pay-a-bill) per one software component". The result is a large number of small components sharing data. It causes inefficiencies and big maintenance problems.

References Kral y Zemlicka (2007)

Classification Service Oriented Architecture

Also know as -

References to techniques Torkamani y Bagheri (2014), Palma et al. (2013) **Type of detection techniques** Heuristic-based, Checklist-based

Name Sending excessive amount of objects

Description Communication overhead occurs when a huge volume of data is transmitted to the clients. Commonly, Web applications provide facilities for searching and browsing large query result sets. This, besides requiring a long time to transmit the result sets over the network, demands a long time from clients to process these sets afterward. In consequence, these facilities are expensive when large result sets are transmitted from the business tier to the client tier.

References Crasso et al. (2009)

Classification Performance

Also know as -

References to techniques Crasso et al. (2009)

Type of detection techniques Heuristic-based

Name of the tools JEETuning Expert

User Licenses -

Programming languages Java

Name Service Chain

Description The Service Chain appears when clients request consecutive service invocations to fulfill their goals. This kind of dependency chain reflects the subsequent invocation of services.

References Palma et al. (2013)

Classification Service Oriented Architecture

Also know as -

References to techniques Nayrolles et al. (2013), Palma et al. (2013),

Type of detection techniques Heuristic-based

Name of the tools SOMAD

User Licenses Free and open-source software

Programming languages Java

Name Spaghetti Code

Description Appears as a program or system that contains very little software structure. Coding and progressive extensions compromise the software structure to such an extent that the structure lacks clarity, even to the original developer, if he or she is away from the software for any length of time. If developed using an object-oriented language, the software may include a small number of objects that contain methods with very large implementations that invoke a single, multistage process flow. Furthermore, the object methods are invoked in a very predictable manner, and there is a negligible degree of dynamic interaction between the objects in the system. The system is very difficult to maintain and extend, and there is no opportunity to reuse the objects and modules in other similar systems.

References Brown et al. (1998)

Classification Object Oriented Design

Also know as -

References to techniques Peldszus et al. (2016), Ouni et al. (2014), Maiga et al. (2012a), Kessentini et al. (2011), Hassaine et al. (2010), Moha et al. (2010)

Type of detection techniques Heuristic-based, Machine learning-based

Name of the tools HULK

User Licenses Free and open-source software

Programming languages Java

Name Splitting Hairs

Description Splitting into two separate tiers of Service and Process, with separate rules and governance results in divergent solutions.

References Jones (2006)

Classification Service Oriented Architecture

Also know as -

References to techniques Torkamani y Bagheri (2014)

Type of detection techniques Checklist-based

Name Stable Abstraction Breaker

Description Stable Abstraction Breaker refers to a subsystem for which its stability level is not proportional with its abstractness.

References Fontana et al. (2016)

Classification Object Oriented Design

Also know as SAP Breaker

References to techniques Fontana et al. (2016) **Type of detection techniques** Heuristic-based

Name of the tools inFusion **User Licenses** Proprietary software

Programming languages C, C++, Java

Name Standardization Paralysis

Description XML enables an easy development of languages transformable very quickly into standards. Many people believe that the standardization based on (quickly changing and obsoleting) standards is a proper solution. The result is that the developers often strongly depend on software vendors and that any solution based on legacies is almost impossible. Examples of this antipattern are the frequent use of SOAP (Simple Object Access Protocol) only as a XML message mediator and the e-government in one country where all the newly incorporated systems must be certified for compatibility with a very quickly changing set of rules what blocks the e-government development.

References Kral y Zemlicka (2007)

Classification Service Oriented Architecture

Also know as SOA Over standardized

References to techniques Torkamani y Bagheri (2014) **Type of detection techniques** Checklist-based

Name Stovepipe Service

Description Is an antipattern with large number of private or protected methods that primarily focus on performing infrastructure and utility functions (i.e. logging, data validation, notifications, etc.) and few business processes (i.e. data type conversion), rather than focusing on main operational goals (i.e. very few public methods). This may result in services with duplicated code, longer development time, inconsistent functioning, and poor extensibility.

References Palma et al. (2013)

Classification Service Oriented Architecture

Also know as -

References to techniques Palma et al. (2013) **Type of detection techniques** Heuristic-based

Name Swiss Army Knife

Description Is an excessively complex class interface. The designer attempts to provide for all possible uses of the class. In the attempt, he or she adds a large number of interface signatures in a futile attempt to meet all possible needs. Real-world examples of Swiss Army Knife include from dozens to thousands of method signatures for a single class. The designer may not have a clear abstraction or purpose for the class, which is represented by the lack of focus in the interface. Swiss Army Knives are prevalent in commercial software interfaces, where vendors are attempting to make their products applicable to all possible applications.

References Brown et al. (1998)

Classification Object Oriented Design

Also know as Kitchen Sink, ISP (Interface Segregation Principle) Violation

References to techniques Aras y Selçuk (2016) , Peldszus et al. (2016), Ouni et al. (2014), Maiga et al. (2012a), Moha et al. (2010), Marinescu (2005)

Type of detection techniques Heuristic-based, Machine learning-based

Name of the tools HULK, ProDeOOS

User Licenses Free and open-source software, Proprietary software

Programming languages Java, C++

Name Tangle

Description In Object Oriented Design, is a large group of objects or package whose relationships are so interconnected that a change in any one of them could affect all the others. Large "Tangles" are a major cause of instability in large systems ("Tangles" exist both class and package level).
From a performance perspective, "Tangled ownership contexts" is a design violation: the programmer expected some structure to be encapsulated, but discovers that it is tangled with some other structure. For example, a programmer who writes a compiler may intend that parse trees from different compilations should not refer to each other in the heap. There is no one simple solution to this complex antipattern, but object ownership profiling can provide vital information to diagnose and repair it.

References Fontana et al. (2016), Rayside y Mendel (2007)

Classification Object Oriented Design, Performance

Also know as Tangled Ownership Contexts

References to techniques Fontana et al. (2016), Rayside y Mendel (2007)

Type of detection techniques Heuristic-based

Name of the tools Structure101, SA4J

User Licenses Proprietary software, Free and open-source software

Programming languages Java, .NET

Name The Blob

Description The Blob is found in designs where one class monopolizes the processing, and other classes primarily encapsulate data. This antipattern is characterized by a class diagram composed of a single complex controller class surrounded by simple data classes. The key problem here is that the majority of the responsibilities are allocated to a single class. In general, The Blob is a procedural design even though it may be represented using object notations and implemented in object oriented languages. A procedural design separates process from data, whereas an object oriented design merges process and data models, along with partitions. The Blob contains the majority of the process, and the other objects contain the data. Architectures with The Blob have separated process from data; in other words, they are procedural style rather than object oriented architectures.
From a performance perspective, this antipattern creates problems bycausing excessive message traffic. In the behavioral form of the problem, the excessive traffic occurs as the "god" class requests and updates the data it needs to control the system from subordinate classes. In the data form, the problem is reversed as subordinates request and update data in the "god" class. In both cases, the number of messages required to perform a function is larger than it would be in a design that assigned related data and behavior to thesame class.

References Palomba et al. (2014), Lanza y Marinescu (2006), Smith y Williams (2000), Brown et al. (1998), Akroyd (1996), Riel (1996)

Classification Object Oriented Design, Performance

Also know as God Class, Big Class, Winnebago, God Object, Fat Class, Blob Controller, Blob Data-Container

References to techniques Arcelli et al. (2018), De Sanctis et al. (2017), Aras y Selçuk (2016), Fontana et al. (2016), Peldszus et al. (2016), Arcelli et al. (2015), Ganea y Marinescu (2015), Hecht et al. (2015), Kaur y Kaur (2014), Ouni et al. (2014), Sharma y Anwer (2014), Maiga et al. (2012a), Pulawski (2012), Erdemir et al. (2011), Kessentini et al. (2011), Sfayhi y Sahraoui (2011), Trubiani y Koziolek (2011), Hassaine et al. (2010), Moha et al. (2010), Stoianov y Şora (2010), Cortellessa et al. (2009), Vaucher et al. (2009), Brühlmann et al. (2008), Salehie et al. (2006), Kreimer (2005), Marinescu (2005), Trifu et al. (2004)

Type of detection techniques Heuristic-based, Machine learning-based, Model-based, Relational logic-based

Name of the tools ProDeOOS, IYC, Metanool, Antipattern Testing Tool, Verso, HULK, jGoose, E-quality, PAD, Extension PCM Bench, panda-aemilia, Structure101

User Licenses Free and open-source software, Proprietary software

Programming languages Java, C++, .NET

Name The Knot

Description Is a set of very low cohesive services, which are tightly coupled. These services are thus less reusable. Due to this complex architecture, the availability of these services may be low, and their response time high.

References Palma et al. (2013)

Classification Service Oriented Architecture

Also know as -

References to techniques Nayrolles et al. (2013), Palma et al. (2013) **Type of detection techniques** Heuristic-based

Name of the tools SOMAD **User Licenses** Free and open-source software

Programming languages Java

Name The Ramp

Description Any situation in which the amount of processing required to satisfy a request increases over time will produce the behavior.

References Smith y Williams (2003)

Classification Performance

Also know as -

References to techniques De Sanctis et al. (2017), Cortellessa et al. (2009) **Type of detection techniques** Model-based

Name of the tools panda-aemilia **User Licenses** Free and open-source software

Programming languages NA

Name Tiny Service

Description A tiny service is a service that incompletely represents an abstraction, implementing only a subset of the necessary methods, resulting in the need for multiple services to completely implement the abstraction. The result is that clients have to utilize several services to implement a significant business process, and need to implement the sequencing and workflow.

References Dudney et al. (2002)

Classification Service Oriented Architecture

Also know as Refactor Mercilessly

References to techniques Nayrolles et al. (2013), Palma et al. (2013) **Type of detection techniques** Heuristic-based

Name of the tools SOMAD **User Licenses** Free and open-source software

Programming languages Java

Name Too many Cooks in the SOA

Description Multiple services with the same functional behavior are added to the registry without each group checking for service functionality beforehand. It is the proliferation of services that are identical to each other but are supported by different teams or within the same team in the enterprise. Projects are more interested in building services than in re-using them; the numbers of services proliferates while the amount of functionality increases slowly.

References Jones (2006)

Classification Service Oriented Architecture

Also know as A maze of twisty services, All alike

References to techniques Torkamani y Bagheri (2014) **Type of detection techniques** Checklist-based

Name Tower of Babel

Description Occurs when processes use different format of data and they spend too many times in convert them to an internal format. This antipattern occurs in complex, distributed data-oriented systems in which the same information is often translated into an exchange format (by a sending process) and then parsed and translated into an internal format (by receiving processes). The performance loss in this case is clearly due to the excessive overhead caused by the translation and parsing operations which may be executed several times in the whole execution process.

References Arcelli et al. (2018)

Classification Performance

Also know as -

References to techniques Arcelli et al. (2018) **Type of detection techniques** Model-based

Name Traffic Jam

Description Occurs when one problem causes a backlog of jobs that produces wide variability in response time which persists long after the problem has disappeared.

References Smith y Williams (2002)

Classification Performance

Also know as -

References to techniques De Sanctis et al. (2017), Trubiani y Koziolek (2011), Cortellessa et al. (2009) **Type of detection techniques** Model-based

Name of the tools Extension PCM Bench, panda-aemilia **User Licenses** Free and open-source software

Programming languages NA

Name UBER service

Description The antipattern of creating a very limited set of services, or even just one, to try and provide all functionality within the enterprise SOA, related to Singleton SOA antipattern. This causes the functionality and operations being added to a service, to be mushed into a group of functionality that may or may not be related, creating tight coupling between them.

References Jones (2006)

Classification Service Oriented Architecture

Also know as -

References to techniques Torkamani y Bagheri (2014) **Type of detection techniques** Checklist-based

Name Unstable Dependency

Description Describe a subsystem that depends on other subsystems that are less stable than itself.

References Fontana et al. (2016)

Classification Object Oriented Design

Also know as -

References to techniques Fontana et al. (2016) **Type of detection techniques** Heuristic-based

Name of the tools inFusion **User Licenses** Proprietary software

Programming languages C, C++, Java

Name URI Permission Flaw

Description A malicious application can obtain a URI permission to a part of a content provider that it is not authorized to access. This vulnerability is due to another flaw in the Android permission protocol: granted URI permissions are not revoked when the associated content provider is uninstalled, leaving dangling permissions that can be exploited for a similar type of attack.

References Bagheri et al. (2015)

Classification Security in Android

Also know as -

References to techniques Bagheri et al. (2015)

Type of detection techniques Relational logic-based

Name of the tools Alloy Analyzer

User Licenses Free and open-source software

Programming languages Alloy

Name Vendor Lock In

Description A software project adopts a product technology and becomes completely dependent upon the vendor's implementation. When upgrades are done, software changes and interoperability problems occur, and continuous maintenance is required to keep the system running. In addition, expected new product features are often delayed, causing schedule slips and an inability to complete desired application software features.

References Brown et al. (1998)

Classification Service Oriented Architecture

Also know as Product Dependent Architecture, Bondage and Submission, Connector Conspiracy

References to techniques Torkamani y Bagheri (2014)

Type of detection techniques Checklist-based

Name Wide Subsystem Interface

Description This flaw is inspired by the motivation for the Facade design pattern (Gamma et al., 1994). Similar to classes, we can also speak about the interface of a subsystem. This interface consists of the classes that are accessed or, more precisely, are accessible from outside the package. The flaw refers to the situation where this interface is very wide, this causing a very tight coupling between the package and the rest of the world, which is undesirable. In most of the cases the interface can be reduced by introducing a Facade class. If this does not help it might be a sign that the God Package flaw has also crept in. In this case the possibility of relocating some of the classes should be considered.

References Marinescu (2002)

Classification Object Oriented Design

Also know as Lack of facade

References to techniques Marinescu (2005)

Type of detection techniques Heuristic-based

Name of the tools ProDeOOS

User Licenses Proprietary software

Programming languages Java, C++

Name Yoyo Problem

Description A project has a Yoyo Problem if the programmer has to keep flipping between many different class definitions in order to follow the control flow of the program and the structure is hard to understand due to excessive fragmentation. This problem appears for example when we have a very deep inheritance graph, and the implementation of methods in derived classes calls many methods defined at a higer level in the inheritance hierarchy.

References Taenzer et al. (1989)

Classification Object Oriented Design

Also know as -

References to techniques Pulawski (2012), Stoianov y Şora (2010)

Type of detection techniques Machine learning-based, Relational logic-based

Anexo 3

SonarQube

En este anexo se describen los principales aspectos de la herramienta *SonarQube* y una guía de instalación y configuración de esta herramienta para detectar *code smells*, en los lenguajes analizados en esta tesis. Este trabajo fue realizado en la Actividad Integradora: *Análisis estático de código utilizando SonarQube*, que dirigí durante el transcurso de mi tesis.

3.1. Herramienta

SonarQube[1] es una herramienta de código abierto para la gestión de la calidad del software, dedicada a analizar y a medir de forma continua la calidad del código. Utiliza analizadores y reglas para procesar el código fuente y detectar errores típicos de programación, *bugs*, *code smells* y vulnerabilidades. Como resultado brinda información sobre la calidad del código y realiza sugerencias sobre las partes que son mejorables. Actualmente soporta más de 20 lenguajes de programación entre los que se encuentran Java, C#, C, C++, PL/SQL, Cobol, Ruby, Python, JavaScript, entre otros. También brinda la posibilidad de crear plugins en un lenguaje específico o establecer reglas propias. Se integra con Maven[2], Ant[3] y herramientas de integración continua como Atlassian Bamboo[4], Azure DevOps[5], TeamCity[6] y Jenkins[7], lo que permite automatizar

[1] https://www.sonarqube.org
[2] https://maven.apache.org
[3] https://ant.apache.org
[4] https://www.atlassian.com
[5] https://azure.microsoft.com
[6] https://www.jetbrains.com/teamcity
[7] https://jenkins.io

el análisis cada vez que se sube el código. Soporta numerosas herramientas de control de versiones como Git[1], Subversion[2], CVS[3] y Mercurial[4]. También permite analizar el código fuente mientras se está programando con SonarLint[5], que es una herramienta que se integra como extensión a Eclipse[6], IntelliJ[7], Visual Studio[8], entre otros IDEs.

La herramienta presenta las siguientes ediciones:

- Community Edition: Utilizado por más de 85.000 organizaciones. Incluye *SonarQube*, más de 60 plugins, SonarLint, soporte para 14 lenguajes, DevOps integration, y soporte para 5 IDEs.
- Developer Edition (requiere licencia): A lo que ofrece la Community Edition agrega análisis de ramas y pull requests, detección de inyección de vulnerabilidades, notificaciones en SonarLint y soporte para 20 lenguajes entre los que se incluyen C y C++.
- Enterprise Edition (requiere licencia): A lo que ofrece la Developer Edition agrega gobernanza, informes ejecutivos, soporte para 24 lenguajes y soporte técnico experto.
- Data Center Edition (requiere licencia): Diseñado para alta disponibilidad. A lo que ofrece la Enterprise Edition agrega redundancia de componentes, integridad de datos y soporte técnico experto.

3.1.1. Conceptos principales

A continuación se describen los conceptos principales de *SonarQube*.

Issues

Los *Issues* en *SonarQube* se dividen en 3 tipos:

- *bugs*: problema que representa algo que está mal en el código y necesita ser arreglado.
- *Code smell*: problema relacionado con la mantenibilidad del código. Indican debilidades en el diseño que pueden estar frenando el desarrollo o

[1] https://git-scm.com
[2] https://subversion.apache.org
[3] http://www.nongnu.org/cvs
[4] https://www.mercurial-scm.org
[5] https://www.sonarlint.org
[6] https://www.eclipse.org
[7] https://www.jetbrains.com/idea
[8] https://visualstudio.microsoft.com/es

aumentando el riesgo de errores o fallas en el futuro. Son un indicador de factores que contribuyen a la deuda técnica. Algunos tipos de *code smells* son el incumplimiento de estándares, código duplicado, código muerto, falta de pruebas unitarias, mala distribución de la complejidad, clases largas, métodos largos, entre otros.

- *Vulnerability*: problema relacionado con la seguridad del software.

Rules

Una regla es una norma de codificación o práctica que debe seguirse. En *SonarQube* existen 4 tipos de reglas para detectar diferentes tipos de problemas:

- *Code smell* (Maintainability domain)
- *Bugs* (Reliability domain)
- *Vulnerability* (Security domain)
- *Security Hotspot* (Security domain)

Estas reglas son proporcionadas por los plugins y se ejecutan sobre el código fuente para generar evidencias sobre su calidad. Tienen una gran cobertura de estándares de calidad como CWE[1], CERT[2], MISRA[3], SANS[4] y OWASP[5]. Debido a que los plugins se liberan con mucha más frecuencia que la plataforma *SonarQube*, es recomendable revisar el centro de actualizaciones de la herramienta para ver si hay versiones más nuevas.

Metric

Las métricas pueden tener valores o medidas variables a lo largo del tiempo. Ejemplos: número de archivos, número de líneas de código, complejidad, densidad de líneas duplicadas, cobertura de línea por pruebas, etc.

Measure

Es el valor de una métrica para un archivo o proyecto dado en un momento dado. Por ejemplo, 125 líneas de código en la clase *MyClass* o densidad de

[1] https://cwe.mitre.org
[2] https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards
[3] https://www.misra.org.uk
[4] https://www.sans.org/top25-software-errors
[5] https://www.owasp.org

líneas duplicadas de 30 % en el proyecto *MyProject*.

Snapshot

Es un conjunto de medidas y problemas de un proyecto dado en un momento dado. En cada análisis se genera un snapshot y se guarda en la base de datos.

Technical Debt

SonarQube considera como deuda técnica el tiempo que se tendría que invertir para corregir los *code smells* encontrados. Al tener asignado un peso temporal, puede servir para tener una idea del por qué se tarda tanto en incluir ciertos cambios y si es necesario asignar recursos para refactorizar, o incluso considerar no seguir invirtiendo en nuevas funcionalidades.

Technical Debt Ratio

Representa la relación entre el costo para desarrollar un software y el costo para arreglarlo. La fórmula es la siguiente: *deuda técnica / costo de desarrollo*. El costo de desarrollo a su vez puede ser expresado como: *costo para desarrollar 1 línea de código * número de líneas de código*. Por defecto, *SonarQube* establece el costo para desarrollar 1 línea de código en 0.06 días. Este valor es configurable desde la pantalla de administración de la herramienta.

***SQALE Rating* (Software Quality Assessment based on Lifecycle Expectations)**

Este concepto está relacionado con el ratio de la deuda técnica y sirve para transmitir de forma rápida y visual la calidad del proyecto con una calificación entre la A y la E. Si este valor es menor a 5 %, la calificación es A y significa que el proyecto está sano, entre el 6 % y el 10 % la calificación es B y significa que el proyecto está en unos parámetros aceptables, entre el 11 % y el 20 % la calificación es una C y significa que la salud ya no es buena y hay que empezar a tomar medidas correctivas, entre el 21 % y el 50 % la calificación es una D y significa que se deben tomar medidas urgentes para llevar el proyecto a zonas más saludables, más del 50 % es una E y significa que está en zona de rescate y se debe tomar una decisión entre apostar por un rescate, con todos los esfuerzos que esto significa de tiempo y dinero, o dejar que el proyecto entre en "banca rota" para buscar otro camino fuera del mismo. Desde la pantalla

de administración de *SonarQube* se pueden ajustar estos valores.

3.1.2. Principales funcionalidades

En la página de inicio del proyecto se puede ver un resumen de la calidad del proyecto. Muestra la cantidad de *bugs*, vulnerabilidades, *code smells*, y deuda técnica.

Fix the Water Leak

El paradigma de la fuga de agua es una manera simple pero poderosa para administrar la calidad del código: el código nuevo (agregado o modificado) debe ponerse bajo control antes que cualquier otra cosa. Una vez que la fuga esté bajo control, la calidad del código comenzará a mejorar sistemáticamente. *SonarQube* integra este concepto en los proyectos mostrando en un área destacada la cantidad de *bugs*, vulnerabilidades, *code smells* y deuda técnica del código nuevo.

Apply Quality Gates

Para aplicar completamente una práctica de calidad de código en todos los equipos de desarrollo se debe configurar un Quality Gate. Este es un concepto central de *SonarQube* que permite establecer un conjunto de requisitos para indicar si una nueva versión de un proyecto puede entrar en producción o no. Por ejemplo que el código nuevo no tenga vulnerabilidades, que la pruebas tengan una cobertura mayor al 50 %, y que el código duplicado no supere el 3 %.

Detect Issues

La página *Issues* del proyecto brinda poder completo para analizar en detalle cuáles son los principales problemas, dónde están ubicados y cuándo se agregaron a nuestro código:

- *Bugs*: *SonarQube* permite encontrar los errores más complicados navegando fácilmente a través del código mientras los señala en cada ubicación. Detecta problemas tanto en un código que es incorrecto como en un código que no tiene el comportamiento deseado.
- *Code smells*: Un código con *code smells* hace probablemente lo que debería, pero será difícil de mantener. En el peor de los casos, será tan

confuso que los desarrolladores pueden introducir errores inadvertidamente, por ejemplo: código duplicado, código no cubierto por pruebas unitarias y código demasiado complejo. *SonarQube* navega a través del código señalando la ubicación y cuándo se agregó cada *code smell* al código. Brinda sugerencias para su corrección y hace una estimación del esfuerzo en minutos.

- *Vulnerability*: *SonarQube* ayuda a encontrar y rastrear vulnerabilidades en el código. Por ejemplo inyección de SQL, hard-coded passwords y errores mal administrados.

Show Activity and History

En esta sección, se puede profundizar en los detalles de la historia de un proyecto de manera muy sencilla y precisa para comprender mejor lo que sucedió en el pasado.

Rule Management

Los analizadores de código de *SonarQube* incluyen perfiles de calidad predeterminados que ofrecen conjuntos de reglas que funcionan para la mayoría de los proyectos. Si es necesario pueden configurarse fácilmente para ajustarse a cada proyecto.

Explore execution paths

SonarQube se basa en varios motores de flujo de datos que permiten a los analizadores explorar todos los caminos de ejecución posibles para detectar los errores más difíciles. Incluso una función simple que contiene solo 10 ramas diferentes puede llevar a 100 caminos de ejecución posibles en tiempo de ejecución. Comprobar manualmente que esos 100 caminos de ejecución no llevan a errores es realmente muy difícil.

3.1.3. Arquitectura

La plataforma *SonarQube* consiste en cuatro componentes: servidor, base de datos, plugins instalados en el servidor y analizadores. En la figura 3.1 se visualizan estos componentes y la interacción entre ellos.

1. **Servidor *SonarQube***

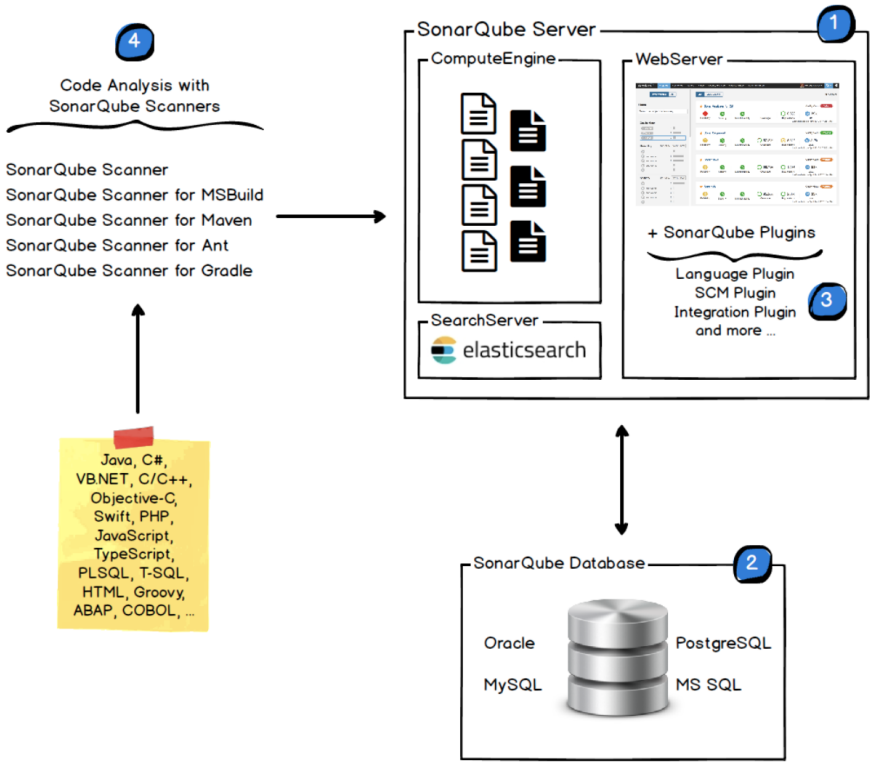

Figura 3.1: Arquitectura de SonarQuebe

- Web Server: Permite configurar el servidor, ajustar los perfiles de calidad, instalar plugins, administrar proyectos y ver los resultados del análisis.
- Search Server basado en Elasticsearch: Utilizado para las búsquedas desde la interfaz de usuario
- Compute Engine Server: Responsable de procesar los informes de análisis de código y guardarlos en la base de datos de *SonarQube*.

2. Base de datos
   - Guarda la configuración del servidor (security, plugins settings, etc) y las snapshots de calidad de proyectos, vistas, etc.
3. Plugins
   - Se pueden instalar en el servidor plugins para los lenguajes, SCM, integración, autenticación, gobierno, etc.
4. Analizadores
   - Los Sonar-Scanners ejecutan el análisis de código línea por línea.

Cuando el análisis finaliza, envían el resultado al servidor *SonarQube*.

3.2. Instalación y configuración

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 SO.
- Java: Oracle JRE 8, OpenJDK 8.
- Base de datos: PostgreSQL (9.3 - 9.6, 10, 11), Microsoft SQL Server (12, 13, 14), Oracle (11G, 12G, XE Edition), MySQL (5.6, 5.7).
- Navegador web: Habilitar JavaScript en el navegador.

Instalación

A continuación se describen los pasos necesarios para instalar y configurar *SonarQube* en Windows de la edición *Community Edition*.

1. Descargar e instalar *SonarQube*
   Descargar la última versión y descomprimir el archivo descargado en una carpeta. A modo de ejemplo, se toma como ubicación el directorio "c:/sonarqube".
2. Configurar la base de datos
   SonarQube viene por defecto configurado para utilizar H2, una base de datos embebida. H2 solo esta recomendada para pruebas, por lo tanto para este trabajo se utiliza como base de datos PostgreSQL 11.

   a) Crear usuario y base de datos en PostgreSQL, como se muestra a continuación:
   Abrir una consola de comandos y 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;
-q
```

b) Asociar la base de datos a *SonarQube*, como se muestra a continuación:

- Editar el archivo *sonar.properties* ubicado en "c:/sonarqube/conf/" para configurar el acceso a la base de datos.
- Descomentar las siguientes líneas y asignar la base de datos, el usuario y la contraseña creados en el paso anterior.

```
# User credentials.
# Permissions to create tables, indices and triggers must be granted to JDBC user.
# The schema must be created first.
sonar.jdbc.username=sonar
sonar.jdbc.password=sonar

#—— PostgreSQL 9.3 or greater
# By default the schema named "publicïs used. It can be overridden with the parameter "currentSchema".
sonar.jdbc.url=jdbc:postgresql://localhost/sonar
```

3. Iniciar el Servidor *SonarQube*

- Abrir una consola de comandos e ir a la carpeta "c:/sonarqube/bin/windows-x86-64".
- Ejecutar el comando *StartSonar.bat*.
  SonarQube al iniciar hace la conexión a la base de datos y verifica que existan las tablas necesarias para su funcionamiento, si no existen las crea.
- Confirmar que el servidor inició correctamente al ver el siguiente mensaje:

```
jvm 1 — 2018.12.16 10:55:45 INFO app[][o.s.a.SchedulerImpl] SonarQube is up
```

4. Acceder al servidor web

- Verificar que se haya iniciada 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)*.

3.2.1. Análisis de código Java

1. Instalar plugin SonarJava
   SonarJava[1] contiene las reglas que permiten encontrar *code smells*, *bugs*

[1] https://www.sonarsource.com/products/codeanalyzers/sonarjava.html

y vulnerabilidades en el código Java. Se encuentra instalado de manera predeterminada en el servidor *SonarQube*, por lo que no es necesario instalarlo.

2. Instalar y configurar analizador Sonar-Scanner
   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*.

   a) Descargar Sonar-Scanner:
   Descargar la opción de Sonar-Scanner para Windows y descomprimir el archivo en una carpeta. A modo de ejemplo, se toma como ubicación "c:/sonar-scanner/"

   b) Configurar Sonar-Scanner

   - Editar el archivo *sonar-scanner.properties* ubicado en "c:/sonar-scanner-windows/conf" para configurar los datos del servidor.
   - Indicar los datos del servidor web *SonarQube*.

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

   c) Configurar el proyecto a analizar
   Crear el archivo *sonar-project.properties* en la carpeta raíz del proyecto con la siguiente información:

```
# 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 file. Replace "\" 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=.
```

3. Ejecutar análisis

   - Abrir una consola de comandos y ubicarse en el directorio donde está el archivo *sonar-project.properties* del proyecto que se quiere analizar.

- Ejecutar el comando sonar-scanner.
  Al finalizar se puede consultar toda la información referente al análisis en el servidor *SonarQube*, donde también se ven todos los proyectos que se han analizado a lo largo del tiempo.

3.2.2. Análisis de código Ruby

1. Instalar plugin SonarRuby
   SonarRuby[1] contiene las reglas que permiten encontrar *code smells*, *bugs* y vulnerabilidades en el código Ruby. Para instalarlo se deben seguir los siguientes pasos:
   - Acceder al servidor *SonarQube*.
   - Iniciar sesión como administrador (admin/admin).
   - Seleccionar la opción *Marketplace* de la sección *Administration*.
   - Buscar el plugin SonarRuby e instalarlo.
   - Reiniciar el servidor.
2. Instalar y configurar analizador Sonar-Scanner
   Realizar los pasos indicados en 3.2.1: Instalar y configurar analizador Sonar-Scanner
3. Ejecutar análisis
   Realizar los pasos indicados en 3.2.1: Ejecutar análisis.

3.2.3. Análisis de código C/C++

1. Instalar plugin SonarCFamily para C/C++
   SonarCFamily[2] para C/C++ contiene las reglas que permiten encontrar *code smells*, *bugs* y vulnerabilidades en el código C/C++. Este plugin no está disponible en la edición *Community Edition*, sí está disponible en la edición *Developer Edition* que requiere licencia. Las alternativas analizadas son las siguientes:

   a) Utilizar una licencia de evaluación de *Developer Edition*
   Fue solicitada a SonarSource (empresa que desarrolla *SonarQube*) una licencia de evaluación con fines académicos, respondieron que

[1] https://www.sonarsource.com/products/codeanalyzers/sonarruby.html
[2] https://www.sonarsource.com/products/codeanalyzers/sonarcfamilyforcpp.html

solo envían licencias de evaluación a las empresas. Por lo tanto se descartó esta opción.

b) Utilizar SonarCloud[1]

SonarCloud es un servicio en la nube que permite usar en forma libre *SonarQube* con todos sus plugins, incluido *SonarCFamily for C/C++*. Para utilizarlo hay que subir el proyecto a un repositorio público como GitHub (o Bitbucket, Azure DevOps). Luego vincular el repositorio en GitHub con SonarCloud. SonarCloud descarga el código, lo analiza y lo publica para todos los usuarios. Esta modalidad soluciona el acceso a *SonarCFamily for C/C++* pero establece una forma de uso que no puede aplicarse a este trabajo ya que para el análisis de código se utilizan programas extraídos del curso *Principios y fundamentos del Proceso Personal de Software* de la Facultad de Ingeniería, Universidad de la República, y no se puede publicar el código de los estudiantes sin su consentimiento. Por lo tanto se descartó esta opción.

c) Utilizar SonarLint + SonarCloud

SonarLint es una herramienta de apoyo para los desarrolladores que se integra como extensión a los IDEs (Eclipse, IntelliJ, Visual Studio, entre otros). Para utilizarlo se debe vincular a un servidor *SonarQube* (o SonarCloud), para que descargue las reglas que permiten detectar problemas en el código. Desde el IDE se puede ver el resultado del análisis, pero no se puede subir al servidor *SonarQube* (o SonarCloud) ya que SonarLint no dispone de esta funcionalidad. Esta es finalmente la opción que se implementa para analizar el código en C y C++ ya que SonarLint permite utilizar todos los plugins disponibles en SonarCloud (incluido *SonarCFamily for C/C++*) sin necesidad de subir el código a un repositorio público. Como consecuencia no se podrá hacer reportes y tener el análisis en la base de datos de *SonarQube*.

2. Instalar y configurar SonarLint en Eclipse (lenguaje Inglés)

   a) Descargar e instalar Eclipse CDT para C/C+

   b) Instalar SonarLint

   - Acceder a Eclipse

[1] https://sonarcloud.io

- Seleccionar la opción *Marketplace* de la sección *Help*.
- Buscar SonarLint e instalarlo.
- Reiniciar Eclipse.

c) Vincular SonarLint con SonarCloud

- Desde Eclipse, abrir el cuadro de diálogo *Connect to a SonarQube Server* desde la sección *File*.
- Seleccionar la opción para conectarse a SonarCloud y continuar.
- Ingresar el token de autenticación para usar SonarCloud, o hacer clic en *Generate token* para generar uno.
- Se abrirá el navegador y cargará la página de inicio de sesión de SonarCloud. Iniciar sesión utilizando una cuenta de GitHub (o Bitbucket, Azure DevOps).
- En la página *Security* de SonarCloud, asignar un nombre al token y seleccionar *Generate*. Se devolverá el token creado.
- Seleccionar el enlace *Organizations* de la barra de navegación principal para agregar una organización.
- Crear una nueva organización.
- En la página *Create Organization* seleccionar la opción *Create manually*.
- Ingresar el nombre de la organización y continuar.
- Confirmar en la siguiente pantalla seleccionado *Create Organization*.
- Desde la página *Projects* de la organización, seleccionar *Analize new project*.
- Seleccionar *Setup manually*.
- Ingresar los datos: *Project key*, *Display name* y seleccionar *Set Up* para crear el proyecto dentro de la organización.
- De vuelta en Eclipse, en el cuadro de diálogo *Connect to a SonarQube Server*, agregar el token generado y continuar.
- Ingresar el nombre de la organización creada en SonarCloud y continuar.
- Ingresar un nombre para la conexión y continuar.

Como resultado se agrega a SonarLint una conexión a SonarCloud.

3. Configurar el proyecto a analizar

Los proyectos en Eclipse deben ser del tipo *C++ Project* y tener asociados un compilador. Las herramientas necesarias para compilar código en C/C++ (gcc-core, gcc-g++, make) se pueden descargar con Cygwin.

- Crear un protecto C++:
- Seleccionar opción *Toolchains Cygwin GCC*.
- Copiar los fuentes dentro del proyecto.
- Configurar SonarLint en el proyecto y seleccionar la opción *Bind to SonarQube or SonarCloud* de SonarLint.
- Seleccionar la conexión creada en el paso anterior.
- Ingresar el nombre del proyecto creado en el paso anterior.

4. Ejecutar análisis
   Seleccionar la opción *Analize* de SonarLint. Al finalizar se podrá ver toda la información referente al análisis en la ventana SonarLint Report.

3.2.4. Análisis de código C#

1. Instalar plugin SonarC#
   SonarC# está basado en el compilador *Microsoft Roslyn*, utiliza las técnicas más avanzadas (pattern matching, dataflow analysis) para analizar el código y encontrar *code smells*, *bugs* y vulnerabilidades en el código C#. Se encuentra instalado de manera predeterminada en el servidor *SonarQube*, por lo que no es necesario instalarlo.
2. Instalar y configurar analizador Sonar-Scanner for MSBuild
   Sonar-Scanner for MSBuild[1] es el cliente para proyectos en C#. 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*.

   a) Instalar MSBuild Tools

   b) Instalar SonarScanner for MSBuild

   - Abrir una consola de comandos y ejecutar:

   ```
   dotnet tool install –global dotnet-sonarscanner
   ```

   c) Configurar Sonar-Scanner for MSBuild

[1] https://docs.sonarqube.org/display/SCAN/Analyzing+with+SonarQube+Scanner+for+MSBuild

- Editar el archivo *sonar-scanner.properties* indicando los datos del servidor *SonarQube*.

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

3. Ejecutar análisis
   - Abrir una consola de comandos y ubicarse en el directorio donde está la solución del proyecto a analizar (.sln o .csproj).
   - Ejecutar los comandos

```
dotnet sonarscanner begin /k:"keyDeNuestroProyecto"
dotnet build
dotnet sonarscanner end
```

Al finalizar se podrá consultar toda la información referente al análisis en el servidor *SonarQube*, y ver todos los proyectos que se han analizando a lo largo del tiempo.

Anexo 4

Plantillas de diseño del PSP

En este anexo se presentan las plantillas de diseño definidas en el *Personal Software Process* (Humphrey, 1995) y utilizadas en el curso *Principios y Fundamentos del Personal Software Process*.

Operational Specification Template Instructions

Purpose	- To hold descriptions of the likely operational scenarios followed during program use - To ensure that all significant usage issues are considered during program design - To specify test scenarios
General	- Use this template for complete programs, subsystems, or systems. - Group multiple small scenarios on a single template, as long as they are clearly distinguished and have related objectives. - List the major scenarios and reference other exception, error, or special cases under comments. - Use this template to document the operational specifications during planning, design, test development, implementation, and test. - After implementation and testing, update the template to reflect the actual implemented product.
Header	- Enter your name and the date. - Enter the program name and number. - Enter the instructor's name and the programming language you are using.
Scenario Number	Where several scenarios are involved, reference numbers are needed.
User Objective	List the users' likely purpose for the scenario, for example, to log onto the system or to handle an error condition.
Scenario Objective	List the designer's purpose for the scenario, for example, to define common user errors or to detail a test scenario.
Source	- Enter the source of the scenario action. - Example sources could be user, program, and system.
Step	Provide sequence numbers for the scenario steps. These facilitate reviews and inspections.
Action	Describe the action taken, such as - Enter incorrect mode selection. - Provide error message.
Comments	List significant information relating to the action, such as - User enters an incorrect value. - An error is possible with this action.

Figura 4.1: Instrucciones de la plantilla operacional

Operational Specification Template

Student		**Date**	
Program		**Program #**	
Instructor		**Language**	

Scenario Number		**User Objective**	
Scenario Objective			
Source	**Step**	**Action**	**Comments**

Figura 4.2: Plantilla operacional

Functional Specification Template Instructions

Purpose	- To hold a part's functional specifications - To describe classes, program modules, or entire programs
General	- Use this template for complete programs, subsystems, or systems. - Use this template to document the functional specifications during planning, design, test development, implementation, and test. - After implementation and testing, update the template to reflect the actual implemented product.
Header	- Enter your name and the date. - Enter the program name and number. - Enter the instructor's name and the programming language you are using.
Class Name	- Enter the part or class name and the classes from which it directly inherits. - List the class names starting with the most immediate. - Where practical, list the full inheritance hierarchy.
Attributes	- Provide the declaration and description for each global or externally visible variable or parameter with any constraints. - List pertinent relationships of this part with other parts together with the multiplicity and constraints.
Items	- Provide the declaration and description for each item. - Precisely describe the conditions that govern each item's return values. - Describe any initialization or other key item responsibilities.
Example Items	An item could be a class method, procedure, function, or database query, for example.

Figura 4.3: Instrucciones de la plantilla funcional

Functional Specification Template

Student		**Date**	
Program		**Program #**	
Instructor		**Language**	

Class Name	
Parent Class	

Attributes	
Declaration	**Description**

Items	
Declaration	**Description**

Figura 4.4: Plantilla funcional

Logic Specification Template Instructions

Purpose	- To contain the pseudocode for a program, component, or system - To enable precise and complete program implementation - To facilitate thorough design and implementation reviews and inspections
General	- Use this template to document the program's detailed logic. - After implementation and testing, update the template to reflect the actual implemented product. - During detailed design, write the pseudocode needed to describe all of the program's logic. - Use plain language and avoid using programming instructions wherever practical.
Header	- Enter your name and the date. - Enter the program name and number. - Enter the instructor's name and the programming language you are using.
Design References	List the references used to produce the program's logical design. - the Operational, Functional, and State templates - the program's requirements - any other pertinent source
Parameters	- Where needed, define any parameters or abbreviations used. - Avoid duplicating definitions on other templates and reference these other definitions where they are needed.

Figura 4.5: Instrucciones de la plantilla lógica

Logic Specification Template

Student		Date	
Program		Program #	
Instructor		Language	

Design

References

Parameters

Figura 4.6: Plantilla lógica

State Specification Template Instructions

Purpose	- To hold the state and state transition specifications for a system, class, or program - To support state-machine analysis during design, design reviews, and design inspections
General	- This form shows each system, program, or routine state, the attributes of that state, and the transition conditions among the states. - Use this template to document the state specifications during planning, design, test development, implementation, and test. - After implementation and testing, update the template to reflect the actual implemented product.
Header	- Enter your name and the date. - Enter the program name and number. - Enter the instructor's name and the programming language you are using.
State Name	- Name all of the program's states. - Also enter each state name in the header space at the top of each "States/ Next States" section of the template.
State Name Description	- Describe each state and any parameter values that characterize it. - For example, if a state is described by SetSize=10 and SetPosition=3, list SetSize=10 and SetPosition=3.
Function/Parameter	- List the principal functions and parameters. - Include all key variables or methods used to define state transitions or actions.
Function/Parameter Description	- For each function, provide its declaration, parameters, and returns. - For each parameter, define its type and significant values.
Next State	- For each state, list the names of all possible next states. - Include the state itself.
Transition Condition	List the conditions for transition to each next state. - Use a mathematical or otherwise precise notation. - If the transition is impossible, list "impossible," with a note saying why.
Action	List the actions taken with each state transition.

Figura 4.7: Instrucciones de la plantilla de estado

State Specification Template

Student		Date	
Program		Program #	
Instructor		Language	

State Name	Description
Function/Parameter	**Description**

States/Next States	Transition Condition	Action

Figura 4.8: Plantilla de estado