Revista Cubana de Ciencias Informáticas
Vol. 12, No. 4, Octubre-Diciembre, 2018
ISSN: 2227-1899 | RNPS: 2301
Pág. 156-176
http://rcci.uci.cu

Tipo de artículo: Artículo de revisión
Temática: Ingeniería y gestión de software
Recibido: 02/10/2017 | Aceptado: 15/10/2018

Código con "mal olor": un mapeo sistemático

Bad smell code: a systematic mapping

Oscar Ancán*, Carlos Cares, Ania Cravero

Depto. de Ciencias de Computación e Informática, Universidad de La Frontera, Temuco, Chile

*Autor para correspondencia: oscar.ancan@ufrontera.cl

Resumen
El concepto de código con mal olor (*code smells*) fue introducido a fines de los años 90 y es una manera de referirse a ciertas características subjetivas en el código fuente que podrían repercutir en problemas de operación y mantención. Para corregir estos problemas se ha desarrollado, dentro de la Ingeniería de Software, toda un área de estudio denominada reconstrucción (*refactoring*), centrada principalmente en estrategias de corrección de código anómalo. El objetivo de este artículo es develar el estado actual de los estudios relacionados con los malos olores en el código fuente, considerando principalmente su detección. Se realizó una revisión basada en el protocolo de mapeo sistemático. Las estrategias de búsqueda recuperaron un conjunto de 215 documentos, de los cuales 30 fueron seleccionados para su análisis. Se definieron cuatro categorías de clasificación: método de detección, tipo de aporte, lenguaje estudiado y tipo de olor. Los resultados de la revisión indican un predominio de estudios que detectaban malos olores sobre código escrito en *Java*. Se pudo constatar que la literatura analizada carece de estudios que reporten métodos, herramientas y estrategias de detección en las categorías "abusadores de la orientación a objetos" e "inhibidores", mientras que la mayor concentración de artículos están en las categoría de olor denominada "hipertrofias" y "prescindibles", cuyos principales métodos de detección corresponden a métricas y análisis de logs.

Palabras claves: Código Anómalo, Mantenibilidad, Mapeo Sistemático, Reconstrucción

Abstract
The concept of code smell was introduced in the late 90's and is a way of referring to certain subjective characteristics in the source code that could have repercussions on operation and maintenance. In order to correct these problems, a whole study area called refactoring has been developed within Software Engineering, which focuses mainly on strategies for correcting this type of anomalous code. The objective of this article is to unveil the current status of studies related to code smells, considering mainly their detection. A review was conducted based on the systematic mapping protocol. The search strategies retrieved a set of 215 documents, of which 30 were selected for analysis. Four categories of classification were defined: detection method, contribution type, programming language and code smell category. The results of the review indicate a predominance of studies that detected code smells in Java. It was found that the literature analyzed lacks studies that report methods, tools and detection strategies in the categories "object-oriented abusers" and "change preventers", while the highest concentration of articles are in the code smells categories called "bloaters" and "dispensable", whose main detection methods correspond to metrics and log analysis.

Keywords: *Code Smell, Maintainability, Refactoring, Systematic Mapping*

Introducción

Disponer de código fuente de mala calidad en proyectos de software implica problemas en su compresión (Abbes et al., 2011) y propensión a fallos frente a eventuales cambios (Khomh et al., 2012). A nivel de costos, diversos estudios revelan evidencia de cómo el código difícil de comprender, y eventualmente defectuoso, impacta en los costos de mantención. Uno de los primeros estudios en el área indican que alrededor del 80 % del presupuesto es gastado en mantención (Lientz et al., 1978). Estudios posteriores mantienen esta cifra (Banker et al., 1993; Pigoski, 1996) y otros incluso la suben a un 90 % (Erlikh, 2000). Un estudio relativamente reciente, ha mostrado la tendencia de estos costos mostrando además que, con el pasar de los años, es ascendente (Engelbertink and Vogt, 2010). Estos hechos han motivado un interés relacionado con el análisis del código fuente y cómo dicho análisis podría, en cierta medida, adelantarse a potenciales fallos del software (Fowler et al., 1999; Mäntylä and Lassenius, 2006).

La reconstrucción, o *refactoring*, surge como una técnica para mitigar y/o corregir problemas relacionados con el código fuente de mala calidad y que impactan directamente en la capacidad de ser corregido ante eventuales fallas o extendido debido a nuevos requerimientos (International Organization for Standardization, 2014). Si bien una refactorización puntual no impacta en gran medida en la mantenibilidad del software si lo hace un proceso de refactorización que involucre todo el software (Szőke et al., 2017), por lo que un gran número de estudios de reconstrucción de software se relacionan estrechamente con la calidad del código.

En este artículo, se realiza una revisión de los potenciales síntomas de código defectuoso u "olores", descritos inicialmente en (Fowler et al., 1999), con el propósito de indagar el estado actual de este tópico, específicamente en mecanismos de análisis de código, que permitan anticiparse a problemas más profundos a nivel estructural, focalizándose en su detección.

El método utilizado para la recopilación y selección de artículos es el mapeo sistemático, si bien este protocolo se genera en las ciencias de la salud, ha sido adaptado para estudios en Ingeniería de Software (Petersen et al., 2007). El proceso cuenta básicamente de cinco pasos, cuatro productos intermedios y un producto final, que es el mapa del estado del arte en el tópico deseado.

Este estudio se estructura de acuerdo a las siguientes secciones. La sección 2 expone antecedentes y conceptos básicos considerados en esta investigación. En la sección 3 se describen los resultados obtenidos en cada una de las fases del procedimiento utilizado. En el apartado 4 se analizan los artículos seleccionados y finalmente en la sección 5 se explican la conclusiones del estudio.

Conceptos Básicos

Las anomalías en el código fuente son un síntoma que puede dar indicios de problemas más profundos en la mantenibilidad del software. Así lo exponen diversos estudios que hablan del impacto de una mala codificación en la evolución del software a través del tiempo, en términos de la comprensión y la trazabilidad de éste (Fontana et al., 2013; Stolee and Elbaum, 2013). Dichas anomalías han sido denominadas "malos olores" aludiendo de esta manera a una evaluación de la estructura y su eventual impacto en errores y/o en problemas de mantención. Estos "olores" fueron planteadas inicialmente por Fowler y Beck (Fowler et al., 1999) y, posteriormente, su marco conceptual se complementó con anomalías o malos "síntomas" a nivel de diseño de software (Martin, 2008).

Si bien Fowler et al. (1999) plantea una clasificación de alto nivel, otros autores han planteado dudas respecto a la real clasificación de los "olores" proponiendo enfoques alternativos para establecer una taxonomía (Mäntylä and Lassenius, 2006; Mantyla et al., 2003; Moha et al., 2010). Por ejemplo en (Moha et al., 2010) se propone una clasificación basada en la división de "olores" que ocurren dentro de las clases y entre clases y, a su vez, considera los niveles estructurales y léxicos, para cada uno de los niveles superiores. El principal problema con dicha clasificación es que omite algunos "olores" inicialmente planteados en Fowler (Fowler et al., 1999) como *Clases Alternativas con Diferentes Interfaces (Alternative Classes with different Interfaces)* o *Intermediario (Middle Man)*. Por otro lado en (Mäntylä and Lassenius, 2006) se expone una clasificación de 6 niveles: **Hipertrofia** (*Bloater*), **Prescindibles** (*Dispensable*), **Abuso de la Orientación a Objetos** (*Object-Oriented Abusers*), **Acopladores** (*Couplers*), **Inhibidores de Cambios** (*Change Preventers*) y finalmente la categoría **Otros** (ver Tabla 1). En el resto del estudio se utilizará ésta taxonomía debido a que su aplicación en el mapeo sistemático resulta más simple de implementar que la clasificación propuesta en (Fowler et al., 1999).

Métricas

Fowler et al. (1999) plantea 22 "olores" en el contexto de Programación Orientada a Objetos como una guía para los desarrolladores de software. No obstante, dichos "olores" carecen de una métrica y una especificación formal de detección (Singh and Kahlon, 2011). Por otra parte, las métricas tradicionales, como la complejidad ciclomática (CYCLO) (McCabe, 1976) o la cantidad de líneas de código (LoC) han demostrado ser incapaces de representar la calidad interna del software y la necesidad de ser reconstruido.

Por lo tanto una métrica de manera aislada no entregará pistas relevantes que indiquen al ingeniero de software la causa del problema, de hecho sólo indicará que existe una anomalía (Marinescu, 2004). De manera complementaria, otros autores han estudiado las posibles correlaciones existentes entre "olores" y un grupo

Revista Cubana de Ciencias Informáticas
Vol. 12, No. 4, Octubre-Diciembre, 2018
ISSN: 2227-1899 | RNPS: 2301
Pág. 156-176
http://rcci.uci.cu

Tabla 1. Clasificación basada en (Mäntylä and Lassenius, 2006) y (Fowler et al., 1999)

Tipo	Definición	Olor
Hipertrofia	Son códigos, métodos y clases que han aumentado de manera excesiva su tamaño y resultan complejas de trabajar.	Método largo (*Long method*), Clase Extensa (*Large class*), Lista larga de parámetros (*Long parameter list*), Obsesión primitiva (*Primitive obsession*), Agrupaciones de datos (*Data Clumps*)
Prescindibles	Es una sección de código innecesaria cuya ausencia haría el código más limpio y fácil de entender.	Generalidad especulativa (*Speculative generality*), Clase perezosa (*Lazy class*), Clase de datos (*Data class*), Código duplicado (*Duplicated code*)
Abusadores de la orientación a objetos	Corresponde a la aplicación incompleta o incorrecta de los principios de programación orientada a objetos	Atributo temporal (*Temporary field*), Sentencia switch (*Switch statements*), Legado rechazado (*Refused Bequest*), Clases alternativas con diferentes interfaces (*Alternative classes with different interfaces*)
Acopladores	Incluye excesivo acoplamiento entre clases	Envidia de funcionalidades (*Feature envy*), Cadenas de mensajes (*Message chains*), Intermediario (*Middle man*), Intimidad inapropiada (*Inappropiate intimacy*)
Inhibidores de Cambio	Al momento de realizar un cambio en una sección de código se deben realizar muchos cambios en otros lugares también	Cambio divergente (*Divergent change*), Cirugía con escopeta (*Shotgun surgery*), Jerarquía de herencia paralela (*Parallel inheritance hierarchies*)
Otros	Elementos no terminados y comentarios irrelevantes, redundantes o absurdos	Bibliotecas incompletas (*Incomplete library class*), Comentarios innecesarios (*Comments*)

de métricas de calidad interna (LoC, número de métodos, clases, paquetes, CYCLO entre otras) dando pie a posibles mecanismos que faciliten la mantención del software, considerando por ejemplo la frecuencia en que aparece un determinado "olor" y su vínculo con la métrica de calidad de software respectiva (Fontana et al., 2013; Yamashita and Counsell, 2013).

Herramientas de Detección

Gran parte de las herramientas de detección de "olores" funcionan en base a métricas y su respectivo umbral de acción (Munro, 2005). Dichos umbrales han sido estudiados ampliamente y consideran valores límite que se deben establecer de manera previa a la detección. De este modo, los resultados obtenidos de un análisis resultan subjetivos al depender de los umbrales de detección, arrojando resultados diferentes dependiendo de las reglas que se utilicen. Por lo tanto, la determinación de, si alguna pieza de código contiene un "olor", aún es algo arbitrario y, hasta ahora no está del todo normado (Fontana et al., 2015).

Pese a lo anterior, existe un gran número de herramientas de software desarrolladas para la detección automática de "olores", sin embargo, todas ellas difieren en la cantidad de "olores" que detectan, su capacidades de procesamiento y enfoque (Hamid et al., 2013; Tomas et al., 2013; Fontana et al., 2011).

Revista Cubana de Ciencias Informáticas
Vol. 12, No. 4, Octubre-Diciembre, 2018
ISSN: 2227-1899 | RNPS: 2301
Pág. 156-176
http://rcci.uci.cu

Actualmente, en el mercado, existe una gran variedad de herramientas que permiten analizar diferentes características internas de un software; basta indagar en extensas revisiones como las planteadas en (Garousi Yusifolu et al., 2015) y en (Tomas et al., 2013) para notar la cantidad de herramientas disponibles. Estudios como los anteriores no indican sobre cuál "olor" actúa la herramienta y se limitan a indicar, de manera muy resumida, sus funcionalidades, como por ejemplo: JDeodorant[1], PMD[2], Checkstyle[3], Decor, STENCH BLOSSOM[4], SonarQube[5], ConcernMeBS[6] y FindBug[7].

Método de investigación

La construcción del mapa del Estado del Arte se ha realizado siguiendo el protocolo de búsqueda bibliográfica refinado por Petersen (Petersen et al., 2007). En la Figura 1 se muestra un esquema de este protocolo. En la parte superior de la figura se exponen los pasos y en la parte inferior los productos. El proceso comienza con la definición de las preguntas de investigación, lo que resulta en el alcance de la investigación. Esto básicamente incluye los años a estudiar y la amplitud del marco conceptual así como el tipo de fuente bibliográfica. El segundo paso considera la ejecución de la búsqueda, lo que produce un conjunto de artículos. El tercer paso, es básicamente la revisión y aplicación de los criterios de inclusión/exclusión definidos en el alcance. El cuarto paso se alimenta de manera del alcance y las preguntas de investigación lo que se cruza con las palabras claves y resúmenes de los trabajos para dar paso a un conjunto de categorías de clasificación para los artículos encontrados. Finalmente, el quinto paso, corresponde a la clasificación de los artículos en las categorías previamente identificadas, produciendo el mapa objetivo del estudio.

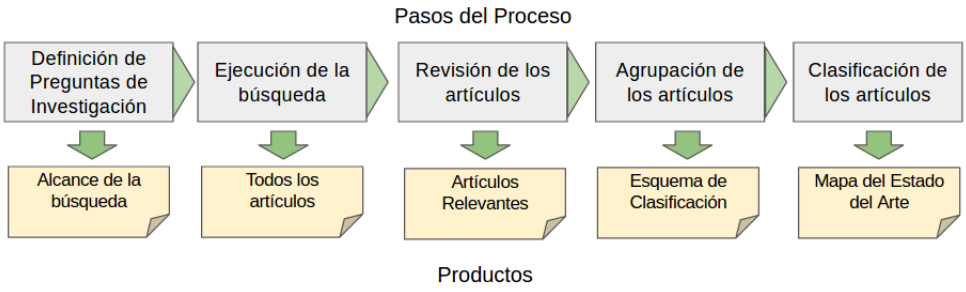

Figura 1. Pasos y productos del protocolo de Petersen et al. (2007)

[1]Más información en http://users.encs.concordia.ca/ nikolaos/jdeodorant/
[2]Más información en https://pmd.github.io/
[3]Más información en http://checkstyle.sourceforge.net/
[4]Más información en http://multiview.cs.pdx.edu/
[5]Más información en https://www.sonarqube.org/
[6]Más información en https://sourceforge.net/projects/concernmebs/
[7]Más información en http://findbugs.sourceforge.net/

Revista Cubana de Ciencias Informáticas
Vol. 12, No. 4, Octubre-Diciembre, 2018
ISSN: 2227-1899 | RNPS: 2301
Pág. 156-176
http://rcci.uci.cu

Definición de las preguntas de investigación y alcance

Las preguntas de investigación formuladas tienen un enfoque pragmático en el sentido de reflejar los modelos, métodos y herramientas orientados a mejorar o facilitar el proceso de mantención de un producto de software. Las preguntas formuladas fueron:

PI1 ¿Cuál es el método de detección de "olores" más utilizado en los trabajos seleccionados?

PI2 ¿Cuál es el tipo de aporte de los estudios seleccionados, en términos de modelo, estrategia y herramienta?

PI3 ¿Qué lenguajes de programación son predominantes como foco de estudio de "olores" en el código?

PI4 ¿Cuál es la categoría de "olores" menos estudiada? por ejemplo Hipertrofia, Prescindibles, Abuso de la Orientación a Objetos, Acopladores etc.

En términos de alcance, se considera revisar estudios experimentales y también no experimentales de tipo transversal. Para estos últimos se consideran los trabajos de tipo exploratorios, descriptivos y correlacionales. Se consideran estudios que arrojen resultados sobre nuevas métricas asociadas a alguna dimensión específica de análisis de código. También se incluirán reportes que entreguen indicios sobre el vínculo entre "olores" y métricas. Finalmente y de manera complementaria, se abordan resultados de experimentos con herramientas de automatización de métricas. También en (Petersen et al., 2007) se plantea que el alcance implica la población de artículos a revisar, para este caso corresponderá a los estudios que aborden la temática de "olores" y su detección.

Ejecución de la búsqueda

La búsqueda se llevó a cabo mediante la integración de las siguientes palabras claves extraídas en base a las preguntas de investigación: *code smell*, *detection*, *metrics*, *refactoring*, *maintainability*. Luego de ajustar algunos criterios la cadena de búsqueda resultó ser la siguiente:

("*code smells*" *OR* "*bad smells*" *OR* "*bad code smells*" *OR* "*bad smells in code*") *AND* "*detection*" *AND* (*metrics OR measure*) *AND refactoring AND maintainability*

De manera complementaria, las fuentes de recolección de artículos fueron las siguientes: Elsevier, IEEE Digital Library, ACM, Springer. Adicionalmente, se ejecutó una búsqueda en Google Scholar, debido a que también se deseaba incorporar un espectro más amplio de fuentes.

Revisión de los artículos

El proceso de revisión de los artículos consideró tres etapas, donde cada etapa tenía por propósito refinar los resultados inicialmente obtenidos.

En primer lugar se revisó el total de artículos obtenidos luego de ejecutar la cadena de búsqueda en los distintos buscadores, además de eliminar repetidos. Dicha búsqueda tenía por objeto recopilar el mayor número de artículos relacionados con "olores" mediante el análisis sobre el título, resumen y palabras claves. Este primer apartado, consideró incluir sólo artículos, publicaciones en revistas y congresos que aborden la temática de "olores" y mecanismos de detección.

En segundo lugar, se centró la búsqueda en el resumen y las conclusiones con el fin de seleccionar aquellos aspectos relacionados con los objetivos inicialmente planteados. De acuerdo a lo anterior, se excluyeron los siguientes artículos: (1) Artículos que hablan de "olores" pero que consideran sólo un "olor" (2) Trabajos que hablan sobre técnicas de detección vinculadas a reconstrucción en general.

El proceso final de selección se realizó en base a la cantidad de citas de cada artículo, donde un mayor índice H consideró su inclusión. A modo de resumen, se incluye la Tabla 2 que presenta los resultados de la búsqueda y selección de acuerdo a todas las iteraciones antes descritas.

Tabla 2. Resultados de la Búsqueda y Selección

	IEEE	**ACM**	**Elsevier**	**Springer**	**G.Scholar**	**Total**
Búsqueda Inicial	53	26	15	24	97	215
Con Criterios	30	17	10	9	12	78
Selección Final	12	7	3	4	4	30

De la selección final, 4 artículos pertenecen a revistas y/o congresos indexados por Google Scholar. De manera completaría, en la Tabla 3 se adjunta el listado con los 30 artículos seleccionados de los cuales alrededor del 37 % fueron publicados en el años 2015 y un porcentaje menor en el año 2014 (17 %) y 2016 (13 %) (Figura 2).

Esquema de Clasificación

Luego del proceso de filtrado de artículos, se procedió a establecer una clasificación en base a cuatros elementos centrales. Dichos elementos se desprendieron de los objetivos específicos del estudio (ver Figura 3).

Método de Detección: corresponde a los métodos con los que es posible detectar un "olor". Para este caso de han considerado cuatro métodos: *análisis de logs*, *basado en visualización*, Valor límite o umbral de acción de la *métrica* y finalmente basados en *estadística y/o probabilidad*. El análisis de *logs* contempla la detección

Tabla 3. Artículos Seleccionados

Autores	Año	Título
N Moha (Moha et al., 2010)	2010	DECOR: A method for the specification and detection of code and design smells
Schumacher J (Schumacher et al., 2010)	2010	Building empirical support for automated code smell detection
FA Fontana (Fontana et al., 2011)	2011	An experience report on using code smells detection tools
Satwinder Singh (Singh and Kahlon, 2011)	2011	Effectiveness of encapsulation and object-oriented metrics to refactor code and identify error prone classes using bad smells
FA Fontana (Fontana et al., 2012)	2012	Automatic detection of bad smells in code: An experimental assessment.
H Liu (Liu et al., 2012)	2012	Schedule of bad smell detection and resolution: A new way to save effort
A Hamid (Hamid et al., 2013)	2013	A comparative study on code smell detection tools
S AyshwaryaLakshmi (Ayshwaryalakshmi et al., 2013)	2013	Agent based tool for topologically sorting badsmells and refactoring by analyzing complexities in source code
P Tomas (Tomas et al., 2013)	2013	Open source tools for measuring the Internal Quality of Java software products. A survey
F Palomba (Palomba et al., 2013)	2013	Detecting Bad Smells in Source Code using Change History Information
Boussaa M (Boussaa et al., 2013)	2013	Competitive coevolutionary code-smells detection
CF Ramos (Conceição et al., 2014)	2014	Streamlining Code Smells: Using Collective Intelligence and Visualization
D Sahin (Sahin et al., 2014)	2014	Code-smell detection as a bilevel problem
FA Fontana (Fontana et al., 2015)	2015	Automatic metric thresholds derivation for code smell detection
M Hozano (Hozano et al., 2015)	2015	Using developers feedback to improve code smell detection
S Fu (Fu and Shen, 2015)	2015	Code Bad Smell Detection through Evolutionary Data Mining
G Rasool (Rasool and Arshad, 2014)	2015	A review of code smell mining techniques
L Amorim (Amorim et al., 2015)	2015	Experience report: Evaluating the effectiveness of decision trees for detecting code smells
A Ouni (Ouni et al., 2015)	2015	Improving multi-objective code-smells correction using development history
H Liu (Liu et al., 2015)	2015	Dynamic and Automatic Feedback-Based Threshold Adaptation for Code Smell Detection
F. Palomba (Palomba et al., 2015b)	2015	Anti-Pattern Detection: Methods, Challenges, and Open Issues.
FA Fontana (Arcelli Fontana et al., 2015)	2015	Comparing and experimenting machine learning techniques for code smell detection
Walter B (Walter et al., 2015)	2015	Including structural factors into the metrics-based code smells detection
F. Palomba (Palomba et al., 2015a)	2015	Mining version histories for detecting code smells
FA Fontana (Fontana et al., 2016)	2016	Antipattern and Code Smell False Positives: Preliminary Conceptualization and Classification
Mkaouer M (Mkaouer, 2016)	2016	Interactive code smells detection: An initial investigation
Kim T (Kim et al., 2013)	2016	Specification and automated detection of code smells using OCL
Sirikul K (Sirikul and Soomlek, 2016)	2016	Automated detection of code smells caused by null checking conditions in Java programs
Rasool G (Rasool and Arshad, 2017)	2017	A Lightweight Approach for Detection of Code Smells
Hozano M (Kaur and Jain, 2017)	2017	Evaluating the Accuracy of Machine Learning Algorithms on Detecting Code Smells for Different Developers

Revista Cubana de Ciencias Informáticas
Vol. 12, No. 4, Octubre-Diciembre, 2018
ISSN: 2227-1899 | RNPS: 2301
Pág. 156-176
http://rcci.uci.cu

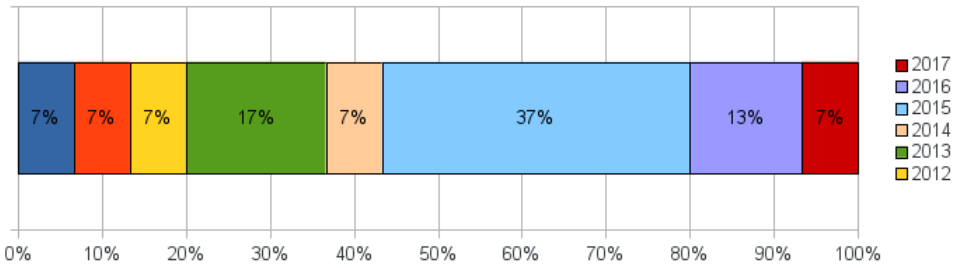

Figura 2. Artículos por año

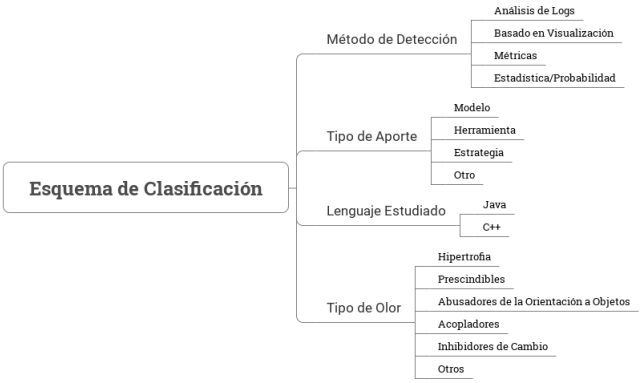

Figura 3. Esquema de Clasificación

de "olores" mediante la revisión de los *logs* de cambios de un repositorio de software como SVN o Git. Los umbrales de funcionamiento de métricas corresponden a valores de ajuste con los cuales una métrica es capaz de detectar la presencia de un síntoma de fallo. Basado en estadística y/o probabilidad significa que se considera la aplicación de funciones estadísticas para procesar determinados comportamientos en el software. Esta último también incluye algoritmos de *machine learning* basados en análisis probabilístico. Finalmente, la detección mediante visualización corresponde al despliegue de un diagrama, principalmente de grafos, para realizar seguimiento a secciones del código susceptibles a problemas.

Tipo de Aporte: corresponde la contribución que realiza el estudio al sector de investigación. En particular, se consideran 4 tipos de aporte: *modelo*, *estrategias*, *herramientas* y *otros*. Modelo corresponde a una representación teórica de un sistema o una interacción compleja entre piezas anómalas de software. Estrategia, incluye

descripciones y procedimientos para llevar a cabo el proceso de detección o aplicación de determinada métrica de software. Herramienta, indica si el artículo en cuestión utilizó para su estudio algún software o biblioteca de tipo comercial o libre en la generación de los resultados del estudio. La categoría "otro" considera algunas ramas de la computación como optimización, agentes, herramientas de inteligencia de negocios y similares.

Lenguaje Estudiado: son los tipos de lenguaje de programación utilizados, tanto para detectar ciertos "olores" o bien como foco de análisis del código fuente. Se considera *Java* y *C++*.

Tipo de "Olor": agrupa los olores según la taxonomía presentada en (Mäntylä and Lassenius, 2006). Ver tabla 1.

Mapa Sistemático

Mediante la extracción de los datos se procedió a generar los mapas en base a tablas desglosadas que contabilizan la cantidad de estudios de acuerdo al esquema de clasificación. Se debe mencionar que, durante el proceso de conteo, algunos artículos ingresaron a varias categorías al mismo tiempo, por lo que al sumar de manera directa las cantidades, el resultado superará los 30 estudios seleccionados.

Todos los mapas presentan el cruce respecto a las categorías de "olores" mediante un gráfico de burbujas, permitiendo cuantificar de manera directa las cantidades de estudios relacionadas con cada objetivo. El gráfico de burbujas expone, mediante el tamaño de su circunferencia, el número de artículos vinculados a una determinada categoría.

Por ejemplo, si tomamos el cruce de lenguaje de programación y la categorización de "olor" (Figura 4) podemos observar que 25 artículos incorporan *Java* para estudiar algún "olor" de la categoría **hipertrofia**, mientras que para el mismo tipo de "olor" existen sólo tres estudios que lo hicieron con lenguaje *C++*.

En relación a los métodos de detección de "olores" (Figura 5) las categorías menos estudiadas fueron las de *visualización*(10 artículos) y las de *análisis de logs*(11 artículos). Del mismo modo las categorías más estudiadas corresponden a **hipertrofia** con umbrales de *métricas*.

Respecto a los tipos de aportes de los diversos estudios seleccionados (Figura 6), 13 artículos proponen *estrategias* para detectar "olores" y nuevamente lo realizan sobre la misma categoría de **hipertrofia**. Sólo existe dos estudios relacionado con **abuso de orientación a objetos** y corresponde a *modelos*.

Finalmente, en la Figura 7 se presenta el cruce de todas las categorías antes expuestas: *Método de detección*, *Tipo de aporte* y *Lenguaje Estudiado*, con los tipos de "olores". Un número considerable de estudios se en-

Revista Cubana de Ciencias Informáticas
Vol. 12, No. 4, Octubre-Diciembre, 2018
ISSN: 2227-1899 | RNPS: 2301
Pág. 156-176
http://rcci.uci.cu

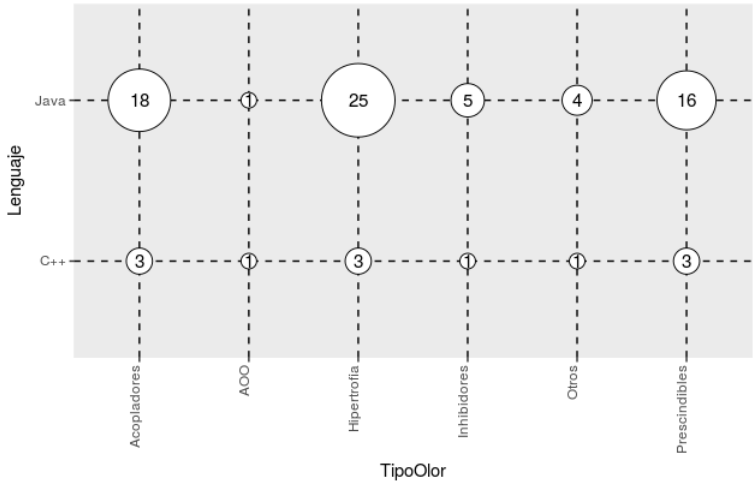

Figura 4. Mapa Tipo de olor v/s Lenguajes de Programación

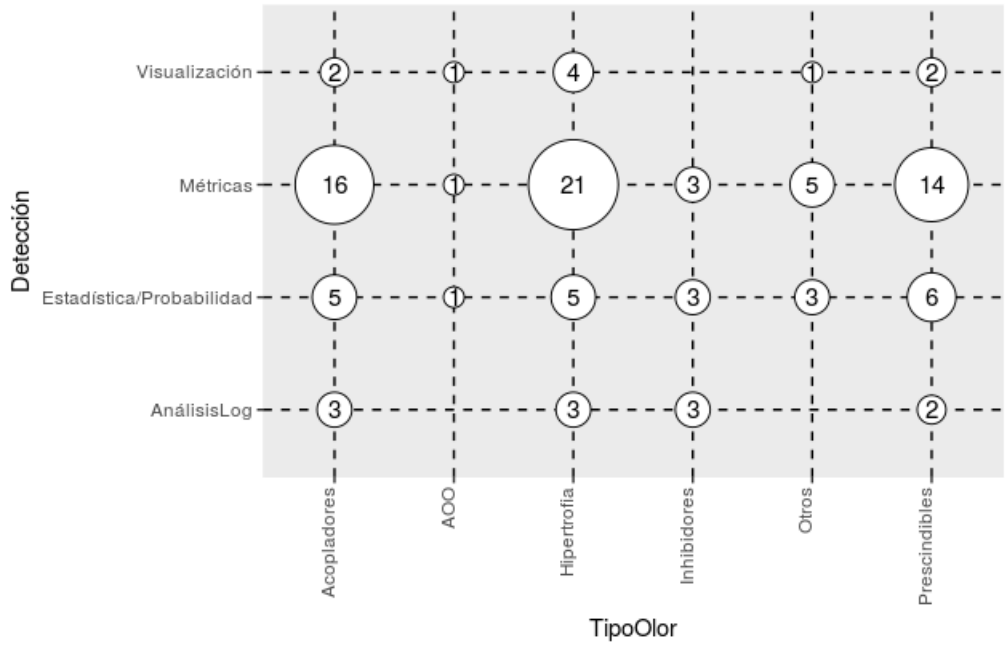

Figura 5. Mapa Tipo de olor v/s Métodos de detección

focaron en la categoría de *aporte* y en *detección*, principalmente sobre la categoría de **hipertrofia**, 29 y 33 respectivamente. Las categoría que menos estudio registró fue **abuso de orientación a objetos**, con escasos 2 artículos para la categoría de *lenguaje* y *aporte*.

Revista Cubana de Ciencias Informáticas
Vol. 12, No. 4, Octubre-Diciembre, 2018
ISSN: 2227-1899 | RNPS: 2301
Pág. 156-176
http://rcci.uci.cu

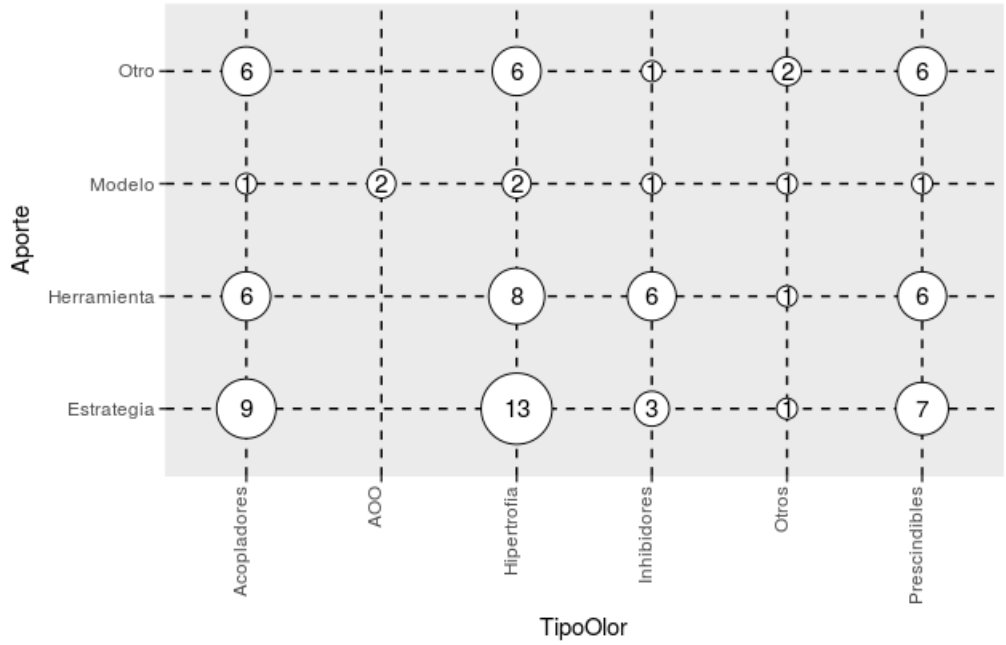

Figura 6. Mapa Tipo de olor v/s Tipo de aporte de los estudios

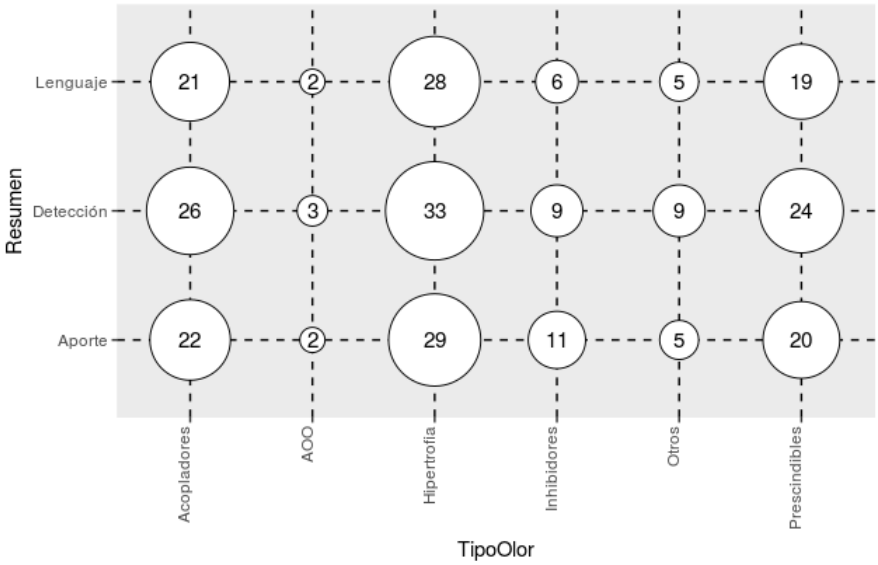

Figura 7. Mapa Resumen de Estudios

Revista Cubana de Ciencias Informáticas
Vol. 12, No. 4, Octubre-Diciembre, 2018
ISSN: 2227-1899 | RNPS: 2301
Pág. 156-176
http://rcci.uci.cu

Resultados y Análisis

El siguiente apartado busca dar respuesta a las preguntas de investigación planteadas inicialmente mediante los resultados arrojados por los diversos mapas.

PI1: ¿Cuál es el método de detección de "olores" más utilizado en los trabajos seleccionados?

Luego de cuantificar los resultados, es posible establecer que existe un gran número de trabajos que han abordado el análisis de "olores" pertenecientes a la categoría de **hipertrofias**. De manera complementaria, los métodos de detección más utilizados son: basado en *métricas* y *análisis de logs*. Esto puede deberse principalmente a la existencia de un sinnúmero de herramientas que realizan dicha tarea de manera automática (Tomas et al., 2013) y el interés por parte de investigadores por normar ciertos elementos que resultan subjetivos a la hora de establecer los umbrales de acción de las métricas (Mäntylä and Lassenius, 2006).

Cabe destacar que muchos de los estudios han abordado el mismo grupo de "olores" relacionados con código duplicado, lista extensas de parámetros, clases y métodos extensos o con demasiado lógica. Por otro lado, los métodos de detección basado en *visualización* para las categoría **inhibidores de cambios** son inexistentes. Similar situación ocurre con la categoría **abusadores de la orientación a objetos**. Una de las potenciales razones de lo anterior es que las métricas para este tipo de "olores" aún no están del todo maduras y resultan inexactas (Conceição et al., 2014).

PI2: ¿Cuál es el tipo de aporte de los estudios seleccionados, en términos de modelo, estrategia o herramienta?

Existen 13 artículos seleccionados que han propuesto una *estrategia* o parte de ésta sólo para estudiar la categoría **hipertrofias**. Lo anterior plantea que existe predominancia por estudiar "olores" relacionados con duplicación de código, métodos y clases extensas. Una posible justificación de lo anterior radica en que un gran número de estrategia de detección se basan en métricas asociadas a una herramientas por lo que dicha categoría resulta simple de abordar con alguna herramienta con métricas convencionales como CYCLO y LoC. En la mayoría de los casos se utilizó también la misma fuente de software proveniente de *Qualitas Corpus* (Tempero et al., 2010). El resto de los artículos de distribuye de manera uniforme a excepción de los *modelos* elaborados para detección que sólo son aplicados a grupos pequeños de "olores".

PI3: ¿Qué lenguajes de programación son predominantes como foco de estudio de "olores" en el código?

Antes de responder esta pregunta, es preciso indicar que algunos estudios abordaban con un lenguaje de programación varias categorías de "olores" por lo que un mismo estudio podría estar incluido en más de una categoría. En ese contexto, gran parte de los estudios han abordado *Java* como lenguaje para realizar las

Revista Cubana de Ciencias Informáticas
Vol. 12, No. 4, Octubre-Diciembre, 2018
ISSN: 2227-1899 | RNPS: 2301
Pág. 156-176
http://rcci.uci.cu

pruebas o bien software implementados en *Java* que, a su vez, analizan código *Java* bajo la propiedad de "reflexión". En menor medida existen estudios, preferentemente vinculados a software empotrado en aparatos electrónicos, en cuyo caso se contempla *C++* como lenguaje de programación. En este marco de predominancia de *Java*, la categoría menos estudiada fue **abusadores de la orientación a objetos**, donde sólo existe un estudio para cada lenguaje. Gran parte de los otros estudios analizan código duplicado y métodos extensos.

PI4: ¿Cuál es la categoría de "olor" menos estudiada?

Finalmente, más de la mitad de los artículos estudiaron sólo 3 categorías de "olor": **hipertrofia**, **prescindibles** y **acopladores** (Ver Fig. 6 y 5). Mientras que el resto de las categorías, como es el caso **abuso de orientación a objetos**, fue la menos analizada. Es preciso mencionar que ningún estudio revisado abordó la totalidad de "olores" de una sola categoría. Siempre existió una mezcla de elementos de las 3 principales mencionadas anteriormente.

Considerando lo anterior, la Tabla 4 presenta todas las categorías en las cuales no se detectaron artículos relacionados con la temática. Así por ejemplo en la categoría de "olor" **abusadores de la orientación a objetos** no se hallaron estudios donde el método de detección se basará en **análisis de logs**. Para este mismo tipo de "olor" pero respecto al tipo de aporte basado en **herramientas**, **otro** y **estrategia** el resultando fue similar. Otras categorías de olores que presentaron vacíos fueron **inhibidores** y **otros** pero sólo en *visualización* y *análisis de logs* respectivamente.

Tabla 4. Necesidad de investigación en olores de código

	MÉTODO DE DETECCIÓN				APORTE DEL ESTUDIO		
	Análisis Log	**Visualización**	**Métricas**	**Est./Prob.**	**Herramienta**	**Modelo**	**Estrategia**
Hipertrofia							
AOO	[x]				[x]		[x]
Inhibidores		[x]					
Prescindibles							
Acopladores							
Otros	[x]						

Conclusiones

Este estudio se ha focalizado en el problema de la mantención de sistemas de software y ha tomado como variable principal los malos "olores" en el código fuente, comúnmente llamados *code smells*, asociado comúnmente con altos costos en la mantención y evolución del producto. Para ello se revisó en extenso el marco conceptual de "olores" en el código, centrado en sus modelos, estrategias y herramientas de detección.

Esta investigación por tanto, ha reportado el actual estado del arte en la detección de "olores" y su impacto en

el proceso de reconstrucción, que se traduce en uno de los principales problemas vinculados a la mantención de software, tal como lo plantea Fowler (Fowler et al., 1999). La mayoría de los artículos analizados describe el uso de métricas para detectar "olores" de los grupos **hipertrofia** y **acopladores**, preferentemente utilizando el lenguaje de programación *Java*.

Uno de los hallazgos más significativos que surgen de este estudio, corresponde a la identificación de categorías no estudiadas como **abusadores de la orientación a objetos**, **inhibidores** y **otros** que dejan de manifiesto la necesidad de abordar dichos vacíos (Ver Tabla 4). Específicamente, la categoría **abusadores de la orientación a objetos** presentó carencia de estudio en el 50 % de las categorías expuestas, tanto para Método de detección (análisis de log) y Tipo de aporte (herramientas, estrategias y otros). Lo anterior podría entenderse bajo el supuesto que la detección de un olor depende de la experiencia de quien analiza el código, del conocimiento en el dominio del proyecto de software, y del análisis sobre proyectos con similares características. Este hallazgo es relevante porque abre la posibilidad de evaluar técnicas de reconocimiento de "olores" sustentados en parámetros semánticos y pragmáticos en lugar de sintácticos.

Un ámbito sin investigar se relaciona con los lenguajes de programación utilizados en los estudios. Por ejemplo, en el conjunto inicial y masivo de 215 estudios, sólo aparecieron dos estudios relacionados con **JavaScript**, lo que no es coherente con el hecho que es uno de los lenguajes ampliamente usado. De este modo, requiere mayor análisis, no sólo la detección de olores sobre lenguajes masivamente usados, sino también los estudios relacionados de costos de mantención asociados a estas tecnologías.

Una de las potenciales limitantes a la validez del estudio se relaciona con el sesgo propio de la selección de artículos. Lo anterior podría implicar la omisión o inclusión de algunos estudios además de las posibles inexactitudes a la hora de extraer datos para el mapa sistemático. Sin embargo, la ruta seguida en cada paso, admite la inclusión o eliminación de estudios permitiendo su actualización si fuese necesario. Aún en este escenario, se cree que las conclusiones relevantes no varían.

En términos de trabajo futuro, el equipo investigador ha trazado una ruta de investigación basada en las categorías más limitadas. Una investigación en agenda es un estudio controlado que permita indagar y comparar el comportamiento de los desarrolladores a la hora de enfrentar anomalías en el código.

Por lo tanto, y a partir de este trabajo, no sólo se han respondido las preguntas de investigación en cuanto al Estado del Arte en el tema del código con "olores", sino que se advierte una agenda de investigación amplia y abierta en término de anomalías del código, sus impactos y modelos de corrección eficientes desde la perspectiva de los costos de mantención.

Referencias

Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 181–190, March 2011. doi: 10.1109/CSMR.2011.24.

L Amorim, E Costa, N Antunes, B Fonseca, and M Ribeiro. Experience report: Evaluating the effectiveness of decision trees for detecting code smells. *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pages 261–269, 2015. doi: 10.1109/ISSRE.2015.7381819.

Francesca Arcelli Fontana, Mika V. M??ntyl??, Marco Zanoni, and Alessandro Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 2015. ISSN 15737616. doi: 10.1007/s10664-015-9378-4.

S. Ayshwaryalakshmi, S. A Sahaaya Arul Mary, and S. Shanmuga Vadivu. Agent based tool for topologically sorting badsmells and refactoring by analyzing complexities in source code. *2013 4th International Conference on Computing, Communications and Networking Technologies, ICCCNT 2013*, 2013. doi: 10.1109/ICCCNT.2013.6726851.

Rajiv D Banker, Srikant M Datar, Chris F Kemerer, and Dani Zweig. Software Complexity and Maintenance Costs. *Commun. ACM*, 36(11):81–94, 1993. ISSN 0001-0782. doi: 10.1145/163359.163375. URL http://doi.acm.org/10.1145/163359.163375.

Mohamed Boussaa, Wael Kessentini, Marouane Kessentini, Slim Bechikh, and Soukeina Ben Chikha. Competitive coevolutionary code-smells detection. In *International Symposium on Search Based Software Engineering*, pages 50–65. Springer, 2013.

Carlos Fábio Ramos Conceição, Glauco De Figueiredo Carneiro, and Fernando Brito E Abreu. Streamlining code smells: Using collective intelligence and visualization. *Proceedings - 2014 9th International Conference on the Quality of Information and Communications Technology, QUATIC 2014*, pages 306–311, 2014. doi: 10.1109/QUATIC.2014.49.

Floris P Engelbertink and Harald H Vogt. How to save on software maintenance costs. *Omnext White Paper. Accessed*, 2010.

L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, May 2000. ISSN 1520-9202. doi: 10.1109/6294.846201.

F. A. Fontana, J. Dietrich, B. Walter, A. Yamashita, and M. Zanoni. Antipattern and code smell false positives: Preliminary conceptualization and classification. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 609–613, March 2016. doi: 10.1109/SANER.2016.84.

Francesca Arcelli Fontana, Elia Mariani, Andrea Morniroli, Raul Sormani, and Alberto Tonello. An experience report on using code smells detection tools. *Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2011*, pages 450–457, 2011. doi: 10.1109/ICSTW.2011.12.

Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11(2):1–38, 2012. ISSN 16601769. doi: 10.5381/jot.2012.11.2.a5.

Francesca Arcelli Fontana, Vincenzo Ferme, Alessandro Marino, Bartosz Walter, and Pawe?? Martenka. Investigating the impact of code smells on system's quality: An empirical study on systems of different application domains. *IEEE International Conference on Software Maintenance, ICSM*, pages 260–269, 2013. ISSN 1063-6773. doi: 10.1109/ICSM.2013.37.

Francesca Arcelli Fontana, Vincenzo Ferme, Marco Zanoni, and Aiko Yamashita. Automatic metric thresholds derivation for code smell detection. *International Workshop on Emerging Trends in Software Metrics, WETSoM*, 2015-Augus:44–53, 2015. ISSN 23270969. doi: 10.1109/WETSoM.2015.14.

Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

Shizhe Fu and Beijun Shen. Code Bad Smell Detection through Evolutionary Data Mining. *International Symposium on Empirical Software Engineering and Measurement*, 2015-Novem:41–49, 2015. ISSN 19493789. doi: 10.1109/ESEM.2015.7321194.

Vahid Garousi Yusifolu, Yasaman Amannejad, and Aysu Betin Can. Software test-code engineering: A systematic mapping. *Information and Software Technology*, 58:123–147, 2015. ISSN 09505849. doi: 10.1016/j.infsof.2014.06.009.

Almas Hamid, Muhammad Ilyas, Muhammad Hummayun, and Asad Nawaz. A Comparative Study on Code Smell Detection Tools. *International Journal of Advanced Science and Technology*, 60:25–32, 2013. ISSN 20054238. doi: 10.14257/ijast.2013.60.03. URL http://www.sersc.org/journals/IJAST/vol60/3.pdf.

Mario Hozano, Henrique Ferreira, Italo Silva, Baldoino Fonseca, and Evandro Costa. Using developers feedback to improve code smell detection. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1661–1663. ACM, 2015.

International Organization for Standardization. ISO/IEC 25000:2014 Software Engineering — Software Product Quality Requirements and Evaluation (SQuaRE) — Guide to SQuaRE, 2005. Technical report, International Organization for Standardization,ISO, 2014.

Kamaldeep Kaur and Shilpi Jain. Evaluation of machine learning approaches for change-proneness prediction using code smells. In *Proceedings of the 5th International Conference on Frontiers in Intelligent Computing: Theory and Applications*, pages 561–572. Springer, 2017.

Foutse Khomh, Massimiliano Di Penta, Yann Ga??l Gu??h??neuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, 2012. ISSN 13823256. doi: 10.1007/s10664-011-9171-y.

Tae-Woong Kim, Tae-Gong Kim, and Jai-Hyun Seu. Specification and automated detection of code smells using ocl. *International Journal of Software Engineering and Its Applications*, 7(4):35–44, 2013.

B P Lientz, E B Swanson, and G E Tompkins. Characteristics of Application Software Maintenance. *Commun. ACM*, 21(6):466–471, 1978. ISSN 0001-0782. doi: 10.1145/359511.359522. URL http://doi.acm.org/10.1145/359511.359522.

Hui Liu, Zhiyi Ma, Weizhong Shao, and Zhendong Niu. Schedule of bad smell detection and resolution: A new way to save effort. *IEEE Transactions on Software Engineering*, 38(1):220–235, 2012. ISSN 00985589. doi: 10.1109/TSE.2011.9.

Hui Liu, Qiurong Liu, Zhendong Niu, and Yang Liu. Dynamic and Automatic Feedback-Based Threshold Adaptation for Code Smell Detection. *IEEE Transactions on Software Engineering*, 5589(c):1–1, 2015. ISSN 0098-5589. doi: 10.1109/TSE.2015.2503740. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7337457.

M. Mantyla, J. Vanhanen, and C. Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 381–384, Sept 2003. doi: 10.1109/ICSM.2003.1235447.

Mika V. Mäntylä and Casper Lassenius. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering*, 11(3):395–431, 2006. ISSN 13823256. doi: 10.1007/s10664-006-9002-8.

Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. *IEEE International Conference on Software Maintenance, ICSM*, pages 350–359, 2004. ISSN 1063-6773. doi: 10.1109/ICSM.2004.1357820.

Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008. ISBN 0132350882, 9780132350884.

T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, July 1976. ISSN 0098-5589. doi: 10.1109/TSE.1976.233837. URL http://dx.doi.org/10.1109/TSE.1976.233837.

Mohamed Wiem Mkaouer. Interactive code smells detection: An initial investigation. In *International Symposium on Search Based Software Engineering*, pages 281–287. Springer, 2016.

Naouel Moha, Yann Ga??l Gu??h??neuc, Anne Fran??oise Le Meur, Laurence Duchien, and Alban Tiberghien. DECOR: A Method for the Specification and Detection of Code and Design Smells. *Formal Aspects of Computing*, 22(3-4):345–361, 2010. ISSN 09345043. doi: 10.1007/s00165-009-0115-x.

Matthew James Munro. Product metrics for automatic identification of "bad smell"design problems in Java source-code. *Proceedings - International Software Metrics Symposium*, 2005(Metrics):125–133, 2005. ISSN 15301435. doi: 10.1109/METRICS.2005.38.

Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Mohamed Salah Hamdi. Improving multi-objective code-smells correction using development history. *Journal of Systems and Software*, 105:18–39, 2015. ISSN 01641212. doi: 10.1016/j.jss.2015.03.040. URL http://dx.doi.org/10.1016/j.jss.2015.03.040.

Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. Detecting bad smells in source code using change history information. *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings*, pages 268–278, 2013. ISSN 1527-1366. doi: 10.1109/ASE.2013.6693086.

Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, 41(5):462–489, 2015a.

Fabio Palomba, Andrea De Lucia, Gabriele Bavota, and Rocco Oliveto. Anti-pattern detection: Methods, challenges, and open issues. *Advances in Computers*, 95:201–238, 2015b.

Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic Mapping Studies in Software Engineering. *International Journal of Software Engineering & Knowledge Engineering*, 17(1):33–55, 2007. ISSN 02181940. doi: 10.1142/S0218194007003112.

Thomas M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. Wiley Publishing, 1st edition, 1996. ISBN 0471170011, 9780471170013.

Ghulam Rasool and Zeeshan Arshad. A review of code smell mining techniques. *Journal of Software: Evolution and Process*, 26(12):1172–1192, 2014. ISSN 20477481. doi: 10.1002/smr.

Ghulam Rasool and Zeeshan Arshad. A lightweight approach for detection of code smells. *Arabian Journal for Science and Engineering*, 42(2):483–506, 2017.

Dilan Sahin, Marouane Kessentini, Slim Bechikh, and Kalyanmoy Deb. Code-Smell Detection as a Bilevel Problem. *ACM Transactions on Software Engineering and Methodology*, 24(1):1–44, 2014. ISSN 1049331X. doi: 10.1145/2675067. URL http://dl.acm.org/citation.cfm?id=2676679.2675067.

Jan Schumacher, Nico Zazworka, Forrest Shull, Carolyn Seaman, and Michele Shaw. Building empirical support for automated code smell detection. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 8. ACM, 2010.

Satwinder Singh and K. S. Kahlon. Effectiveness of encapsulation and object-oriented metrics to refactor code and identify error prone classes using bad smells. *ACM SIGSOFT Software Engineering Notes*, 36(5): 1, 2011. ISSN 01635948. doi: 10.1145/2020976.2020994. URL http://dl.acm.org/citation.cfm?doid=2108144.2108157{%}5Cnhttp://dl.acm.org/citation.cfm?doid=2020976.2020994.

K. Sirikul and C. Soomlek. Automated detection of code smells caused by null checking conditions in java programs. In *2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 1–7, July 2016. doi: 10.1109/JCSSE.2016.7748884.

Kathryn T. Stolee and Sebastian Elbaum. Identification, impact, and refactoring of smells in pipe-like web mashups. *IEEE Transactions on Software Engineering*, 39(12):1654–1679, 2013. ISSN 00985589. doi: 10.1109/TSE.2013.42.

Gábor Szőke, Gábor Antal, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. Empirical study on refactoring large-scale industrial systems and its effects on maintainability. *Journal of Systems and Software*, 129: 107 – 126, 2017. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/j.jss.2016.08.071. URL http://www.sciencedirect.com/science/article/pii/S0164121216301558.

Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. The Qualitas Corpus: A curated collection of Java code for empirical studies. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, pages 336–345, 2010. ISSN 15301362. doi: 10.1109/APSEC.2010.46.

P. Tomas, M. J. Escalona, and M. Mejias. Open source tools for measuring the Internal Quality of Java software products. A survey. *Computer Standards and Interfaces*, 36(1):244–255, 2013. ISSN 09205489. doi: 10.1016/j.csi.2013.08.006. URL http://dx.doi.org/10.1016/j.csi.2013.08.006.

Bartosz Walter, Błażej Matuszyk, and Francesca Arcelli Fontana. Including structural factors into the metrics-based code smells detection. In *Scientific Workshop Proceedings of the XP2015*, page 11. ACM, 2015.

Aiko Yamashita and Steve Counsell. Code smells as system-level indicators of maintainability: An empirical study. *Journal of Systems and Software*, 86(10):2639–2653, 2013. ISSN 01641212. doi: 10.1016/j.jss.2013.05.007. URL http://dx.doi.org/10.1016/j.jss.2013.05.007.