Predicción de Defectos en un Lenguaje Dinámicamente Tipado usando Métricas Estáticas y de Cambio

Tesista: Lic. GULLINO, Mauro

Directora: Dra. ROBIOLO, Gabriela

Codirector: Dr. ROSSI, Gustavo

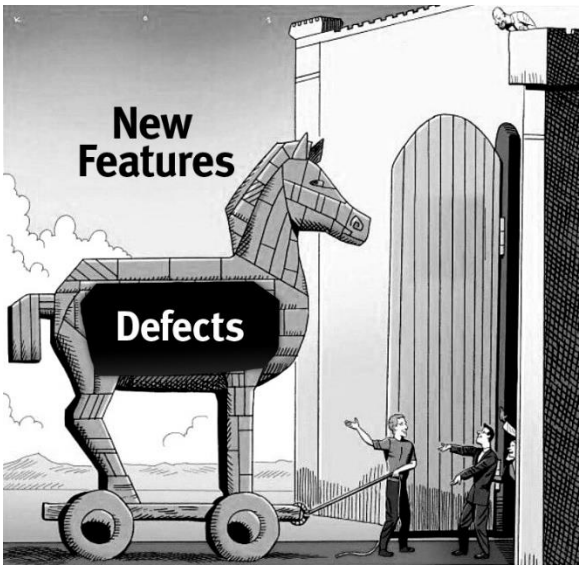
Tesis presentada para obtener el grado de Magíster en Ingeniería de Software

Facultad de Informática

Universidad Nacional de La Plata

Febrero de 2020

A C. S. O.
que vió el inicio de este sueño


New
Features
Defects

Índice

Resumen

La predicción de defectos es un tema importante en la Ingeniería de Software. Puede utilizarse para evaluar la calidad del producto final, estimando si se cumplen los estándares de calidad contractuales o los impuestos para la aceptación del producto por parte del cliente. También puede facilitar la previsión de los recursos para pruebas o verificación formal.

El presente trabajo selecciona un conjunto de **métricas estáticas** (o de producto) y **de cambio** (o de proceso) con el objetivo de predecir los defectos de versiones sucesivas. Como método de estimación se utiliza la **regresión logística**. Se presenta un caso de estudio sobre el proyecto de código abierto MediaWiki, que soporta a Wikipedia. Este producto cuenta con 1000 clases y 365 KLOC.

Se han obtenido las métricas estáticas y de cambio durante el período de un año con distintos cortes temporales. Se construyeron herramientas ad hoc para calcular las métricas definidas por autores como Chidamber & Kemerer [1], Robiolo [2] y Moser [3]. Se busca estudiar la viabilidad de aplicar estas métricas en un lenguaje de programación orientado a objetos diferente, ya que las investigaciones de referencia han trabajado históricamente con lenguajes de tipado estático (principalmente C++ y Java) mientras que MediaWiki se encuentra desarrollado en PHP, que es un lenguaje orientado a objetos interpretado y dinámicamente tipado, muy utilizado en el desarrollo web de pequeña y mediana escala. A estos efectos, también se proponen métricas nuevas, que buscan medir estos aspectos de tipado del lenguaje y sus efectos en el producto y la calidad del mismo.

Para recolectar las métricas de cambio del código fuente se han clasificado manualmente alrededor de 2800 commits de 230 desarrolladores, realizados al repositorio de versiones del proyecto durante el año 2014. Esta clasificación tiene por objeto identificar qué modificaciones al código fuente se pueden imputar a correcciones de defectos (*Fault Repairing*) y cuáles a introducción de nueva funcionalidad (*Feature Introduction*), siguiendo el trabajo de Hassan [4]. Esta clasificación permite definir métricas con las cuales construir modelos de predicción de defectos.

Se comprueba que la cantidad y tamaño de los cambios de tipo "*Feature Introduction*" realizados a una clase son los mejores predictores de los defectos futuros de la misma. Además, se comprueba que utilizando métricas de cambio es posible obtener mejores resultados en la predicción que con métricas estáticas, pero las primeras representan un mayor esfuerzo de medición que las segundas.

1 Introducción

1.1 Objetivos

1.1.1 Objetivo general

Seleccionar métricas de software *estáticas* y *de cambios* dentro del paradigma de la programación orientada a objetos, con el propósito de estimar los defectos presentes en las clases de una aplicación desarrollada con un lenguaje de tipado dinámico.

Partiendo de un framework de métricas existentes [2] se evaluará su aplicabilidad a un lenguaje de tipado dinámico, de modo de seleccionar o definir nuevas métricas que permitan estimar los defectos de software.

1.1.2 Objetivos específicos

- Relevar el estado del arte respecto de las métricas de software
- Analizar y adaptar un framework existente a un lenguaje de tipado dinámico
- Incorporar métricas de cambio definidas con posterioridad a dicho framework
- Implementar las herramientas informáticas necesarias para obtener las métricas seleccionadas sobre una aplicación construida con un lenguaje dinámicamente tipado
- Comprobar en un caso de estudio cuáles son las métricas que mejor estiman los defectos del código

1.2 Pregunta de investigación

¿Es posible predecir los defectos en un lenguaje dinámicamente tipado utilizando métricas estáticas y de cambio?

1.3 Metodología de trabajo

La metodología aplicada es similar a la utilizada por autores de referencia en trabajos publicados en publicaciones de prestigio, con diferencias en el juego de métricas estudiadas, lenguajes y aplicaciones de software seleccionados y tipos de modelos obtenidos. Entre estos autores se pueden mencionar a Basili [5] (1996), Gyimothy [6] (2005), Nagappan [7] (2006), Pan [8] (2006), Moser [3] (2008), Hassan [4] (2009), Caglayan [9] (2009), D'Ambros [10] (2010), Kamei [11] (2010), Jureczko [12] (2011), Giger [13] (2012), Madeyski [14] (2015), Zhang [15] (2014), Kamei [16] (2016) y Nam [17] (2018).

La metodología de trabajo puede describirse a través de estas etapas:

1. Estudio del estado del arte (cfr. Capítulo 2)
2. Selección de métricas de software (cfr. Capítulo 4)
3. Adecuación de las métricas al lenguaje bajo estudio (cfr. 4.2 y 4.3)
4. Desarrollo de los programas para la obtención de los valores de las métricas (cfr. 4.5)
5. Ejecución de estos programas para la obtención de las mediciones (obteniendo planillas Excel utilizadas como datos primarios) (cfr. Anexo A)
6. Obtención del conteo de defectos (cfr. 4.5.2)
7. Construcción de modelos predictivos en R[1] (cfr. Capítulo 6)
8. Análisis de los resultados obtenidos y selección de los mejores modelos (cfr. 6.3.3)
9. Comparación con los trabajos de referencia de otros autores (cfr. 6.5)
10. Respuesta a la pregunta de investigación (cfr. Capítulo 7)

En la Figura 1.1 se ilustra este proceso.

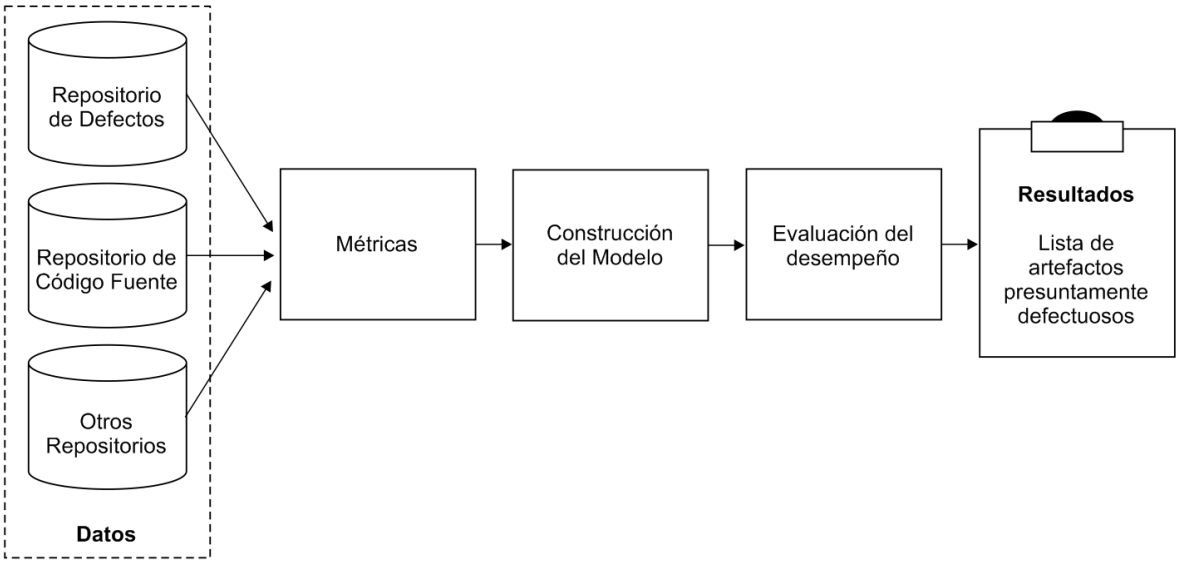


Figura 1.1: Visión general del proceso de predicción de defectos de software. (Adaptado y traducido de [16])

[1] Programa de análisis estadístico. Disponible en www.r-project.org (accedido en 1/2020).

1.4 Organización del documento

En el Capítulo 2 se realiza una presentación general de las métricas en el contexto de la Ingeniería de Software. Se discuten las investigaciones más importantes del área y los hallazgos de otros autores en relación a su uso como estimadores de defectos. En este capítulo también se presenta la definición de defecto utilizada en este trabajo.

En el Capítulo 3 se presenta el sistema de tipos del lenguaje PHP y se discuten las implicancias que tiene para el paradigma de orientación a objetos y las definiciones de métricas de este trabajo.

En el Capítulo 4 se definen detalladamente las métricas seleccionadas para esta investigación. En el Apartado 4.5 se describen las herramientas que han sido desarrolladas para la obtención de estas métricas.

En el Capítulo 5 se describe el caso de estudio, comenzando por un análisis del producto MediaWiki, su organización y su metodología de desarrollo.

En el Capítulo 6 se realiza el análisis y se comparan los resultados de los distintos modelos de regresión que se han construido con las métricas obtenidas para el caso de estudio. Se presentan además los trabajos relacionados de otros autores.

En el Capítulo 7 se presentan la conclusión general y el trabajo futuro.

1.5 Convenciones tipográficas

Para facilitar la lectura de este documento, los nombres de métricas aparecerán con un recuadro dentro del texto. Por ejemplo `LOC`, `CK_WMC` ó `Gul_3`.

Las porciones de código fuente se componen con fondo gris y tipografía monoespaciada, de esta forma:

```
class A {
    public function metodo1() {
        return “hola mundo”;
    }
}
```

2 Métricas de software

En este capítulo se caracterizan los tipos de métricas de software existentes y las razones que fundamentan su estudio. También se presentan las métricas históricamente más utilizadas y las más exitosas dentro de cada grupo. Además se presentan las distintas formas en que las métricas aportan al tópico de predicción de defectos de software y se define qué es un *defecto* a los efectos de este trabajo.

Hace décadas se busca a través del estudio de las métricas de software indicios de propiedades internas de éste que resulten interesantes durante el ciclo de vida.

Una de las primeras métricas utilizadas en la historia de la Ingeniería de Software es el LOC (*Lines of Code*), que se remonta hasta el tiempo en que la introducción de programas para su cómputo se realizaba a través de tarjetas perforadas, en lenguajes como COBOL o Fortran. Varias líneas de investigación ya clásicas [18] [19] han buscado la correlación entre la cantidad de líneas de código y la cantidad de defectos que son esperables en la etapa de mantenimiento de un artefacto con ese tamaño. De este modo, existe desde los orígenes de la Ingeniería de Software un interés por encontrar predictores confiables de defectos, ya que ello se relaciona directamente con la estimación del costo de mantenimiento del producto.

No obstante estos esfuerzos de investigación desde hace cuatro décadas, y del rol central que las métricas tienen en las Ingenierías en general, se ha dicho que en la práctica de la Ingeniería de Software éstas son generalmente consideradas “un lujo” [20]. Esta baja adopción de las métricas en la práctica profesional puede llevar a objetivos poco claros, a la imposibilidad de estimar y controlar los costos y a la dificultad para cuantificar y controlar la calidad del producto.

Al igual que los defectos, los cambios en el software son inevitables [21], tanto por la necesidad de corregir estos mismos defectos como por la necesidad de introducir nuevas características en el producto. Hoy más que nunca el software es modificado a velocidades vertiginosas y por esto son necesarias las métricas y la predicción: para dirigir los esfuerzos de testing en pos de eficientizar el proceso de mejora de la calidad del software [1].

2.1 Métricas tradicionales

LOC es la métrica más simple de extraer. Aún hoy se discute su utilidad y se la utiliza como punto de comparación para otras métricas [22].

En investigaciones recientes [15] se ha encontrado que LOC es la métrica más importante entre las medidas estáticas sobre el código fuente, comparable en su poder de estimación con métricas de más reciente utilización, que se encuentran en estudio –entre ellos en este

trabajo– como ser *métricas de proceso*[2] o *métricas de factores contextuales*[3]. Estas métricas serán presentadas y evaluadas con mayor profundidad en siguientes apartados.

Otra métrica clásica muy conocida es la *métrica de complejidad ciclomática de MacCabe*, presentada en la década de 1970 [20]. Mide la complejidad lógica de un programa a través de un grafo que representa los caminos posibles de ejecución. Esta métrica sería un indicador de cuán difícil es comprender y mantener un artefacto. Ha sido una métrica muy utilizada en el pasado, pero su utilidad dentro del paradigma orientado a objetos es cuestionada por autores como Basili [5]. Su uso ha decaido en el tiempo conforme este paradigma se constituye en predominante.

Las *métricas de complejidad de Halstead* [23], también desarrolladas en la década de 1970, consideran el "vocabulario" utilizado en un programa como una medida de su complejidad. Para calcularlas se mide la cantidad de operadores y operandos presentes en el programa y se obtienen métricas como *Tamaño*, *Volumen*, *Nivel de dificultad*, *Esfuerzo de implementación*, entre otras.

Tanto la métrica de MacCabe como las de Halstead se concentran en atributos internos del software, medidos a una escala reducida (por ejemplo, función por función). Se ha criticado [20] la validez de que estos atributos internos sean una medida fiel de los atributos de calidad externa, es decir, a nivel de producto.

A continuación se presentan los fundamentos de las distintas métricas orientadas a objetos utilizadas en el presente trabajo. La lista concreta de métricas y sus definiciones completas se encuentran en el Capítulo 4.

2.2 Métricas orientadas a objetos

A medida que el paradigma de orientación a objetos se volvió el paradigma predominante han aparecido propuestas de métricas específicas para estos desarrollos. Estas métricas representan un avance respecto de las métricas de complejidad mencionadas anteriormente, desarrolladas en su momento desde el punto de vista del paradigma estructurado. A continuación, se describen los trabajos que dirigen la presente investigación.

2.2.1 Métricas estáticas

Las *métricas estáticas*, también denominadas *métricas de producto*, son aquellas que pueden obtenerse del análisis directo del código fuente. Estas métricas son una medida de los atributos internos [20] de los componentes de un sistema. Tienen como ventaja que pueden

[2] Por ejemplo, la cantidad de cambios (revisiones) que se la ha hecho a un archivo o clase.
[3] Por ejemplo, la cantidad de archivos que componen el sistema.

obtenerse sin necesidad de ejecutar el código[4], lo que permite incorporarlas en etapas tempranas del proyecto.

Chidamber & Kemerer [1] (en adelante C&K), en un trabajo ya clásico, proponen métricas que miden la complejidad en términos del diseño de clases de un sistema. Consideran que éste es un aspecto central en el paradigma orientado a objetos. La propuesta se acompaña de una formalización matemática más rigurosa que sus antecesores.

Ya en 1994, C&K vislumbran como trabajo futuro el cómputo de las métricas en distintos momentos del ciclo de vida para entender cómo evoluciona la complejidad de la aplicación.

C&K establecen algunos conceptos centrales siguiendo los conceptos ontológicos desarrollados por el filósofo argentino Mario Bunge y por Wand, que forman la base de los conceptos de objetos, a saber:

- Acoplamiento (*Coupling*): Ocurre cuando un método de una clase utiliza métodos de otra. En este caso se dice que las clases presentan un grado de "acoplamiento".
- Cohesión (*Cohesion*): Ocurre cuando distintos métodos de una clase realizan operaciones sobre el mismo conjunto de variables de instancia. Si no existen conjuntos distintos de variables operados por distintos métodos se dice que la clase es "cohesiva".
- Complejidad (*Complexity*): Equivale según C&K a la cantidad de elementos que conforman una clase, esto es, métodos y variables de instancia.
- Alcance de las propiedades (*Scope of properties*): Equivale a la influencia que un método o variable de instancia tiene dentro de la jerarquía de clases. Esto se mide con la profundidad del árbol de herencia y con la cantidad de subclases.
- Combinación de clases (*Combination of Object Classes*): Ocurre cuando varias clases se utilizan para crear una nueva, esto es, herencia múltiple. La mayoría de los lenguajes utilizados en la actualidad no permiten esta operación.

En su trabajo, C&K proponen las siguientes seis métricas:

- `WMC` (*Weighted Methods Per Class*): Es la sumatoria de la medida de complejidad de cada método de la clase. A los efectos prácticos, C&K proponen tomar como valor 1 dicha complejidad, por lo que esta métrica se reduce al conteo de la cantidad de métodos. Basili [5] propone el mismo tratamiento. Esta métrica se relaciona directamente con el concepto de Complejidad.
- `DIT` (*Depth of Inheritance Tree*): Es la distancia hasta la raíz en el árbol de herencia de la clase, es decir, la cantidad de superclases que posee. Se realciona con el concepto de Alcance de propiedades. Esta métrica dimensiona cuán influenciada está la clase respecto de otras.
- `NOC` (*Number of Children*): Es la cantidad de subclases que se definen a partir de una clase. También se corresponde con el concepto de Alcance de propiedades. Esta métrica dimensiona la influencia que la clase tiene en el diseño.

[4] Esto también implica que no es necesario compilar, enlazar, instalar... ni realizar ninguna acción propia del proceso de creación del programa ejecutable que cada plataforma requiera, lo que puede significar un esfuerzo considerable en sí mismo.

- CBO (*Coupling between Object Classes*): Es la cantidad de clases a las cuales una clase está acoplada, esto es, utilizando sus métodos. Se corresponde con el concepto de Acoplamiento. Esta métrica dimensiona qué tan modular es un diseño, qué nivel de encapsulamiento se ha logrado y qué tan difícil será testear algunas partes del sistema debido a la interdependencia de las clases.
- RFC (*Response for a Class*): Es la cantidad de métodos de una clase más los métodos a su vez llamados por cada uno de ellos. Se corresponde con el concepto de Complejidad.
- LCOM (*Lack of Cohesion in Methods*): Es una medida de cuántos conjuntos de métodos existen en una clase accediendo a distintas variables de instancia. Se corresponde con el concepto de Cohesión. Esta métrica dimensiona el nivel de encapsulamiento y la corrección del diseño, puesto que una clase poco cohesiva probablemente deba rediseñarse planteando dos o más clases en lugar de una.

C&K señalan que estas métricas presentan todos los beneficios habituales de las métricas de software (en suma, el mejoramiento de los procesos) pero que además, al referirse a aspectos específicos del paradigma orientado a objetos, se constituyen como una forma de evaluar el nivel de adopción de los principios de diseño de este paradigma. Como un conjunto, estas métricas pueden servir a gerentes y líderes de proyecto a dimensionar la adopción de los principios de diseño orientado a objetos sin entrar en los detalles de la implementación. Debido a esto, C&K proponen que este conjunto de métricas puede ser utilizado para identificar áreas del sistema que requieran de un testing más riguroso o aquellas que sean candidatas para un rediseño.

En un trabajo ya clásico, Basili realizó una validación [5] de las métricas C&K a través del estudio de ocho proyectos desarrollados en C++. El objetivo fue analizar de manera experimental si estas métricas pueden ser buenos predictores de clases defectuosas, utilizando regresión logística. Excepto por LCOM, los autores concluyen que las métricas C&K son mejores predictores que otras métricas estáticas no orientadas a objetos[5]. Además, estas últimas sólo pueden ser computadas en etapas posteriores del desarrollo del producto.

Es interesante notar que Basili menciona que algunos aspectos específicos del lenguaje C++ no son considerados en las definiciones de las métricas C&K, por ejemplo las clases amigas y los *templates* de clases. Esto lleva a plantear que se necesita trabajo adicional para extender el conjunto de métricas orientadas a objetos con métricas específicamente diseñadas para cada lenguaje bajo estudio.

En conclusión, las métricas C&K pueden ser utilizadas como buenos predictores de defectos aun durante etapas tempranas del desarrollo. La validación realizada por Basili presenta evidencia empírica de que estas métricas pueden ser indicadores útiles del grado de calidad del producto.

[5] Como métricas estáticas mencionan LOC, cantidad de funciones declaradas, cantidad de llamados a función, complejidad ciclomática, cantidad de bucles, entre otras.

Robiolo [2] elabora un framework de métricas orientadas a objeto[6] desarrollado con la finalidad de evaluar la calidad del diseño de un sistema. Las métricas descriptas se basan en las buenas prácticas de diseño establecidas por autoridades de referencia en el paradigma, tales como Jacobson, Wirfs-Brock, Erich Gamma et al, Page-Jones y Liskov.

Este framework está compuesto por los siguientes grupos de métricas, considerando los aspectos que esta midiendo:

- Tamaño: cantidad de clases, cantidad de interfaces
- Reutilización – Herencia: cantidad de clases externas especializadas, cantidad de interfaces externas extendidas, cantidad de clases que implementan interfaces externas
- Herencia: cantidad de jerarquías de clase, cantidad de jerarquías de interface
- Herencia – Especialización: cantidad de niveles de especialización por jerarquía de clases, cantidad de niveles de especialización por jerarquía de interfaces
- Herencia – Generalización: cantidad de clases raíz no abstractas, cantidad de clases raíz no abstractas que implementan interfaces
- Herencia – Sobreescritura: porcentaje de métodos reemplazados en una jerarquía de clases
- Polimorfismo: cantidad de mensajes polimórficos enviados en una clase
- Granularidad de métodos: promedio de *statements* por método en una clase
- Responsabilidades públicas: cantidad de métodos de interface por clase
- Colaboración – Agregación: cantidad de colaboradores internos por clase

Puede observarse que algunas métricas son del *sistema* en su conjunto (por ejemplo, *cantidad de clases*) y otras se computan *clase a clase* (por ejemplo, *responsabilidades públicas*).

Haciendo uso de estas métricas, la autora ha analizado tres productos desarrollados en lenguaje Java y ha podido sacar conclusiones fundamentadas sobre la calidad del diseño desde el punto de vista del paradigma orientado a objetos. De esta forma, cada producto puede ser cuantificado en términos del reuso, la flexibilidad, las colaboraciones entre clases y el grado de polimorfismo.

Estas métricas permiten no sólo la caracterización de un sistema como un todo sino también señalar alarmas para valores de métricas que estén fuera de un rango esperado. Se propone deducir estos valores esperados desde un marco de referencia conocido. La referencia propuesta por la autora es la biblioteca de clases estándar del lenguaje Java[7].

Las métricas de clase del framework Robiolo son un buen complemento a las métricas C&K. Por ejemplo, las métricas `DIT` y `NOC` se relacionan con *Herencia-Sobreescritura* en cuanto miden aspectos de la herencia. La métrica `WMC` mide un aspecto similar a la de *Responsabilidades públicas*, aunque sin considerar los métodos heredados como sí lo hace esta última. Las métricas Robiolo de *Granularidad* y *Polimorfismo* aportan información no prevista en C&K.

[6] Se trata de la tesis de Magíster en Ingeniería de Software de la autora, presentada ante la UNLP en 2004. La Dra. Gabriela Robiolo es la directora del presente trabajo.
[7] Librerías *java* y *javax* de la plataforma J2SE.

Las métricas de sistema propuestas por Robiolo han sido utilizadas en este trabajo para caracterizar primeramente el producto bajo estudio (cfr. 6.1). Las métricas de clase se han utilizado como predictores de defectos dentro del conjunto de métricas estáticas (cfr. 6.3.1). La definición precisa de cada una de estas métricas puede encontrarse, junto con las otras métricas utilizadas, en el Capítulo 4.

2.2.2 Métricas de cambio

Varias líneas de investigación proponen predecir los defectos del software midiendo la complejidad del *proceso* mediante el cual el código fuente es modificado en lugar de medir la complejidad del *código fuente* mismo [3] [4]. Las *métricas de proceso* miden aspectos como la cantidad de líneas modificadas en una clase, la cantidad de autores que trabajaron en ella, la cantidad de clases que son modificadas con un mismo cambio, la cantidad de veces que una clase fue modificada o corregida, etc. Este tipo de métricas es llamado en este trabajo *"métricas de cambio".*

Se ha reportado en diversas investigaciones que estas métricas son iguales o mejores predictores que las demás métricas aunque no debe subestimarse el esfuerzo de obtenerlas al momento de efectuar una comparación.

Dentro de estas *métricas de cambio* podemos mencionar:

- las *métricas de modificación*, en donde la cantidad de modificaciones que tuvo con anterioridad un archivo fuente se utiliza como predictor de los defectos futuros, esto es, cuanto más se modifica un archivo más probable es que contenga defectos. El "grado de modificación" se puede medir de diferentes formas (cfr. 4.3).
- las *métricas de defectos anteriores*, esto es, cuántas más fallas ha tenido una porción de código aumenta la probabilidad de fallas futuras.

Radjenović [22] propone que las métricas de cambio se pueden dividir en dos grupos: aquellas que miden la *variación o "delta"* de métricas estáticas entre distintas versiones del producto y las de *codechurn*, que miden cuánto se modifica el código fuente entre versiones (veces que una clase fue modificada, cantidad de líneas agregadas, borradas, etc.). En el presente trabajo se denomina *"métricas de cambio"* a las segundas, esto es, a la cuantificación de los cambios que se realizan al código fuente entre dos puntos temporales dentro del desarrollo del producto.

Las métricas de cambio utilizadas en esta investigación surgen del trabajo seminal de Moser [3]. Este trabajo es uno de los 28 seleccionados en la *Systematic Literature Review* que Hall realizó en 2012 [24], con trabajos entre enero de 2000 y diciembre 2010. En esta selección se tuvo en cuenta una serie de criterios de calidad en la presentación de los datos, elaboración del modelo y cómo se reportan los resultados. De 208 reportes sobre el área de estudio en 10 años, sólo 28 cumplieron los requisitos de calidad establecidos.

En el trabajo de Moser se compara un conjunto de *métricas de cambio* con uno de *métricas estáticas*. Para realizar el trabajo experimental utilizan el proyecto Eclipse (producto desarrollado en lenguaje Java) y varios métodos de estimación entre los que incluyen la regresión

logística. El objetivo es determinar cuál conjunto de métricas tiene mejor poder predictivo a la hora de clasificar archivos defectuosos.

Moser incorpora un análisis *cost-sensitive*, en donde se tiene en cuenta que los *falsos negativos* (archivos clasificados como libres de defectos, que realmente contienen defectos) representan mayores costos de desarrollo que los *falsos positivos* (archivos clasificados como defectuosos que en realidad no lo son). Los falsos positivos representan un esfuerzo inútil ya que sugieren la inspección de archivos que no contendrán defectos, pero los falsos negativos son mucho "peores" porque representan artefactos que no serán inspeccionados cuando sí necesitan ser corregidos. Esta corrección será más costosa cuanto más tarde se realice en el ciclo de vida. Por esta razón, los falsos negativos son muy importantes a la hora de evaluar los modelos y las predicciones que generan. Este aspecto es fundamental en el desarrollo OSS[8], donde el esfuerzo debe administrarse muy bien puesto que las horas hombre suelen ser recursos escasos y fluctuantes. Los autores señalan que esta consideración se ha tenido pocas veces en cuenta en la literatura.

En el conjunto de métricas de cambio de Moser se encuentran las siguientes:

- *Revisiones*: cantidad de cambios a un archivo, cantidad de refactorizaciones de un archivo, cantidad de veces que un archivo es modificado para corregir un defecto
- *Autores*: cantidad de autores distintos que trabajaron sobre un archivo
- *Lineas agregadas*: cantidad total de líneas agregadas a un archivo, promedio de líneas que se agregan en todas las revisiones, máxima cantidad de líneas agregadas en una revisión
- *Líneas borradas*: cantidad total de líneas borradas de un archivo, promedio de líneas que se borran en todas las revisiones, máxima cantidad de líneas borradas en una revisión
- *Codechurn*: cantidad de líneas agregadas menos cantidad de líneas borradas, promedio de *codechurn* en todas las revisiones, máximo *codechurn* producido por una revisión
- *Changeset*: cantidad de archivos que cambian juntos en una misma revisión
- *Edad*: semanas transcurridas desde el agregado de un archivo al sistema

Se espera que los artefactos con más tendencia a presentar defectos sean aquellos con números altos de revisiones, *codechurn* elevado, muchos autores, participación en correcciones de defectos anteriores y baja ocurrencia de refactorizaciones.

Las métricas seleccionadas por Moser se han reportado [22] como las más convenientes en la etapa del ciclo de vida *post-release*. Las métricas de cambio `codechurn`, `revisions`, `age` y el tamaño del `changeset` se han reportado como los mejores estimadores, mientras que las métricas estáticas de código fuente parecen no ser útiles en este sentido ya que se ha observado que pierden precisión en cada iteración.

Las métricas de Moser también son utilizadas en un estudio comparativo muy importante realizado por D'Ambros en 2010 [10]. Allí la métrica `bugfixes` es el segundo mejor predictor

[8] *Open Source Software* (Software de Código Abierto)

de defectos luego de las métricas de entropía de Hassan [4] y el primero si se considera la gran dificultad de calcular estas últimas.

En el Apartado 4.3 se encuentran las definiciones de métricas de cambio utilizadas en este trabajo, en el que fundamentalmente se ha seguido a Moser debido a su importancia.

En investigaciones posteriores se han propuesto maneras de medir la entropía de un sistema software [4], entendida como la evolución en el tiempo de las probabilidades de modificación que tienen los archivos fuente que lo componen. Los proyectos que muestran una entropía ascendente son aquellos que se encaminan a tener más defectos, es decir, cuando los cambios en el código introducen modificaciones en demasiados archivos a la vez. Se hipotetiza que esto contribuye a la aparición de defectos debido a la carga cognitiva que genera en los desarrolladores. Este tipo de métricas, sin embargo, resultan muy costosas de calcular desde un repositorio de versiones como SVN o Git y no han sido consideradas en este trabajo.

Algunos investigadores [22] han propuesto métricas sobre las personas concretas que llevan adelante el desarrollo. Estas métricas intentaron encontrar correlación entre los defectos introducidos y la identidad de quienes los introdujeron, de manera de preveer si los cambios efectuados por un desarrollador pueden ser más tendientes al defecto que los de otros. No se ha avanzado mucho en esta línea de investigación y las cuestiones permanecen abiertas.

2.3 Métricas como predictores de defectos

Se estima que el costo de corrección de defectos abarca de un 50% a un 75% del presupuesto total de desarrollo de sistemas [15]. En países desarrollados, se estima que los defectos afectan negativamente a las economías en el orden de decenas de billones de dólares anuales.

La importancia y el interés por la estimación de defectos se aprecia por el largo tiempo en que los investigadores vienen trabajando sobre ellas. Las primeras investigaciones de correlación se remontan a 1971 [16]. En ese entonces Akiyama fue el primero en intentar construir modelos para la predicción de defectos usando métricas basadas en el tamaño y técnicas de regresión.

Podemos encontrar en las siguientes cuestiones los aspectos fundamentales que se buscan contestar al utilizar métricas como predictores de defectos [3]:

- ¿Qué métricas, fáciles de recolectar al inicio del desarrollo, son buenos predictores de defectos futuros?
- ¿Qué modelos (cuantitativos, cualitativos, híbridos) deben ser usados para la predicción de defectos?
- ¿Qué tan exactos (*accurate*) son esos modelos?
- ¿Cuánto le cuesta a la organización utilizar esos modelos o cuáles son los beneficios en términos económicos?

Hay consenso en que no existe un juego de métricas que sea buen predictor para absolutamente todos los proyectos, sino que debe encontrarse el conjunto adecuado a cada uno. Es

decir que un conjunto de métricas puede ser un predictor de defectos válido en un proyecto y sin embargo no poder aplicarse con los mismos resultados favorables en otro. [7]

Se reporta que la inspección manual de código identifica un 60% de defectos [25]. Por lo tanto, un modelo útil tiene que presentar una mejora sobre este porcentaje y además ser de bajo costo para lograr su aplicación efectiva.

Se han reportado como superiores para la predicción de defectos a las métricas basadas en C&K y las métricas de cambio [22].

2.3.1 Definición de defecto

Debido a que el objetivo del presente trabajo abarca la predicción de defectos, es de suma importancia la definición estricta de qué es específicamente un “defecto”. Esta definición determinará qué clases fueron modificadas por ser defectuosas (y no por otra situación), lo que afectará a todo el modelo de predicción construido a posteriori. Por esto, el resultado del modelo predictivo depende de esta definición clave.

Dado el sistema que se analiza en este trabajo (cfr. Capítulo 5), los defectos que tienen mayor interés por su impacto son los relativos a la generación del contenido dinámico de las páginas web que se envían a los navegadores de los usuarios.

Definimos "defecto" siguiendo el estándar IEEE 1044 [26], a saber:

- Imperfección o deficiencia en un artefacto software que hace que el mismo no cumpla los requerimientos o especificaciones, y que por ello necesite ser reparado o reemplazado.
- Un defecto es evidenciado por una o más fallas en tiempo de ejecución.
- No es un defecto si es detectado mediante inspección, análisis de código o técnicas de *testing* pre-release o pre-commit.

Los defectos más habituales[9] encontrados en aplicaciones web pueden ser:

- La inclusión de cierta información o funcionalidad en una página web que no debía ser incluida para ese usuario en ese momento.
- La omisión de cierta información o funcionalidad en una página web que sí debía incluirse para ese usuario en ese momento.
- Todo cálculo numérico que produce un resultado erróneo, por ejemplo, mostrar una recuento de cantidad de páginas vistas que es incorrecta o un cálculo de intervalos de tiempo entre fechas incorrecto.
- Permitir a un usuario realizar acciones no permitidas para su rol en el sistema y viceversa.
- Fallas en tiempo de ejecución producto de la introducción de modificaciones en una parte del código del sistema sin la correcta adecuación del código de otras partes

[9] Con base en la experiencia profesional del autor

dependientes. Nótese que esto no incluye los fallos que se produzcan por componentes externos, por ejemplo fallas que deban corregirse por cambios en el lenguaje de programación subyacente o bibliotecas de terceros, ya que esto no es introducido por el trabajo de los desarrolladores sino por causas externas y no será contabilizado como defecto en este trabajo.

- Fallas en tiempo de ejecución por combinaciones específicas de configuraciones del sistema hechas por el usuario y que los desarrolladores no habían previsto.
- Todo procesamiento de datos que por acción u omisión resulta en información presentada de una manera diferente a la requerida, por ejemplo, errores de *encoding* o *charset*.
- Toda generación dinámica de artefactos accesorios a las páginas web que no cumple requerimientos o estándares externos previamente establecidos al desarrollo de la funcionalidad (por ejemplo: emails, imágenes, documentos PDF, etc).
- Todo manejo de los protocolos y estándares subyacentes que produce fallos en tiempo de ejecución para el usuario, por ejemplo manipulación incorrecta de Cookies, cabeceras HTTP o etiquetas HTML.

2.3.2 Predicción de defectos

Se pueden clasificar los modelos de predicción de defectos en dos tipos [3]: los que predicen la cantidad de defectos en un artefacto (ej. regresión lineal) y los que realizan clasificación de módulos en defectuosos/no-defectuosos (ej. regresión logística).

La regresión logística permite predecir el resultado de una variable categórica (clase defectuosa / no-defectuosa) en función de las variables predictoras (métricas). Esto permite correlacionar la probabilidad de una variable dependiente binaria con variables independientes escalares.

Los modelos de regresión también se pueden dividir en técnicas estadísticas y técnicas de *machine learning*. Las técnicas estadísticas aplican modelos como regresión lineal y logística, univariadas o multivariadas. La categoría de *machine learning* incluye árboles de decisión, Näive Bayes y algoritmos genéticos. [22]

También pueden caracterizarse los modelos según la granularidad de la variable dependiente: método, clase, archivo, módulo u otros. [22]

2.3.2.1 Métricas estáticas como predictores

Estudios recientes muestran que LOC sigue siendo útil. Se reporta que el 20% de los módulos más grandes son responsables de entre un 50 y 60% de los defectos. Tal vez esto sea indicación de que a mayor tamaño naturalmente se tendrá mayor cantidad de defectos, pero persiste la incógnita de cómo utilizar el tamaño como predictor concreto a medida que crece la granularidad [22].

De las métricas de complejidad se ha reportado que, para el paradigma orientado a objetos y proyectos grandes, la complejidad ciclomática de McCabe puede resultar un estimador de una efectividad moderada, no así para proyectos pequeños o realizados en lenguajes procedurales. Las métricas de Halstead fueron encontradas inapropiadas para estimar defectos. [22]

De las métricas diseñadas según los conceptos de orientación a objetos, la suite de C&K es la más utilizada. Es aplicada prácticamente el doble (49%) de veces que las métricas tradicionales de código fuente (27%) o métricas de proceso (24%). Entre estas métricas se reportan como mejores predictores a `CBO`, `WMC` y `RFC` [22]. Otros estudios agregan que `NOC` y `DIT` no resultan útiles [6]. Posteriores validaciones han confirmado estos hallazgos y agregado que `LCOM` tampoco es útil para la predicción de defectos [5]. Otro autor indica que `RFC` y `CBO` son muy buenos predictores que presentan alta correlación con los defectos. [12]

2.3.2.2 Métricas de cambio como predictores

La investigación sugiere [3] que las métricas de cambio, es decir las que aportan información sobre el proceso de desarrollo de un producto software, contienen información más significativa y útil sobre la distribución de los defectos que el código fuente en sí mismo. Parece ser que la chance de que un artefacto sea defectuoso puede predecirse mejor a partir del estudio de los cambios que se le han hecho más que por las características propias del código en determinado momento de su historia.

Jureczko [12] reporta que las métricas de proceso `NDC`[10] y `NDPV`[11] tienen una alta correlación con los defectos al estudiar tanto proyectos propietarios como desarrollos de código abierto. Madeyski [14] encuentra que agregar la métrica de proceso `NDC` mejora mucho la capacidad predictiva a un modelo con sólo métricas de producto.

Se ha reportado que la inclusión de las métricas de cambio en los modelos predictivos lleva a un 28% menos de necesidad de inspección de código sin afectar la precisión. [9]

Desde 2005 se ve un incremento en el uso de métricas de proceso a comparación de las métricas estáticas del código, en donde el conjunto más importante sigue siendo C&K. [22]

La literatura sugiere que las métricas estáticas y las métricas basadas en cambios no son predictores ortogonales, sino que más bien las métricas estáticas parecen ser un conjunto más débil que las métricas de cambio en lo que respecta a la predicción de defectos. [3]

[10] *Number of Distinct Commiters*: cantidad de autores distintos que han modificado una clase
[11] *Number of Defects in Previous Version*: cantidad de defectos corregidos en una clase en una versión previa del producto

2.3.3 Defectos y ciclo de vida

El principal problema encontrado en las primeras etapas del ciclo de vida es la ausencia de información propia. La predicción de defectos en proyectos en sus fases iniciales es un problema importante de la Ingeniería de Software que precisa de más investigación.

Investigaciones recientes [27] han podido entrenar modelos predictivos con información proveniente de otros proyectos de una manera automatizada, logrando resultados comparables a aquellos logrados con un entrenamiento supervisado por un experto. Se necesita más investigación para saber si estos métodos son aplicables en la industria.

En cuanto a la predicción *post-release*, las métricas de proceso `codechurn`, `revisions`, `age` del módulo y el tamaño del `changesize` se han descripto como los mejores estimadores, mientras que las métricas de código fuente parecen no ser útiles en este sentido [22]. Esto puede deberse a que los defectos que se introducen en el ciclo de mantenimiento tienen más que ver con el procedimiento a través del cual se modifica el producto que con el producto mismo. En *pre-release* el mayor impacto parece estar dado por las decisiones de diseño, y éstas quizá sean mejor cuantificadas con métricas estáticas del código fuente. Es decir que la naturaleza del origen de los defectos sería distinta cuando se analizan en *pre-release* o en *post-release*.

Algunas líneas de investigación [28] han propuesto estudiar los cambios introducidos durante la fase *post-release* con el objetivo de clasificar estos cambios en inductores (o no) de defectos. Este concepto es llamado por los autores "*Just-in-time Defect Prediction*" y tiene como ventaja permitir mostrar una advertencia al desarrollador cuando éste aún tiene en mente las cuestiones de diseño que lo llevan a introducir estos cambios. Los autores plantean utilizar métricas como cantidad de archivos modificados, cantidad de líneas de código agregadas o borradas, cantidad de desarrolladores intervinientes, tiempo entre el cambio previo y el evaluado, experiencia del desarrollador, entre otros, para realizar esta clasificación sobre si el cambio que está siendo introducido al repositorio es o no un inductor de defectos. El problema es que tales modelos necesitan muchos datos de entrenamiento y esto los vuelve inviables para proyectos en sus fases iniciales, cuando no se cuenta con tanta información. Se ha intentado entrenar los modelos con información *cross-project* pero aún se necesita más investigación para conocer la factibilidad de estas ideas [17].

Es uno de los objetivos últimos del área poder generar herramientas y técnicas que puedan advertir a los desarrolladores, incluso antes de proceder a enviar el código al repositorio de versiones, que están introduciendo cambios peligrosos dado el historial del archivo o el estado de evolución de las métricas. [16]

En suma, se cuenta con varios tipos de métricas para realizar la caracterización de un sistema, de las clases que lo componen, y de los cambios que se le han introducido en el tiempo. La construcción de modelos a los efectos de la predicción de defectos futuros depende de poder obtener estas métricas desde el código fuente y del repositorio de versiones.

Dado que los autores de referencia trabajaron con lenguajes de programación distintos al estudiado en esta investigación, se presentan en el siguiente capítulo las características propias del diseño del lenguaje bajo estudio y su impacto en la orientación a objetos.

3 Lenguaje PHP

En este capítulo se presenta el sistema de tipos del lenguaje PHP y se caracterizan particularidades de su aproximación a la orientación a objetos. Los detalles del sistema de tipos son muy importantes porque impactan directamente en las métricas a utilizar y en la adaptación que se pueda hacer de éstas para ser utilizadas en sistemas desarrollados con este lenguaje.

El lenguaje PHP[12] se originó en 1995 como un conjunto de programas en lenguaje C para facilitar las tareas de creación de páginas web en un servidor [29]. El desarrollo actual de este lenguaje se realiza por medio de RFC (*Request for Comments*) y un proceso de votación en una comunidad de desarrolladores. Actualmente se encuentra en su versión 7.4 (enero/2020).

El aspecto más relevante del lenguaje es su tipado dinámico, lo que lo ubica en la vereda contraria a los lenguajes más estudiados en predicción de defectos (cfr. 6.6). A continuación se profundiza en estas características.

3.1 Sistema de tipos

PHP es un lenguaje interpretado y dinámicamente tipado. Se denomina *tipado dinámico* (*Dynamic Type Binding*) al hecho de que las variables no tienen un tipo declarado sino que la variable adquiere un tipo cuando ocurre la asignación de un valor a ella [30]. El tipo del valor del lado derecho del *operador de asignación* será el tipo que adquiere la variable. Además, el tipo puede cambiar muchas veces durante el transcurso de la ejecución a medida que se realizan distintas asignaciones. Esto aporta flexibilidad al programador y hace más rápido el desarrollo, aunque supone mayor cómputo en tiempo de ejecución y traslada el chequeo de tipos desde el tiempo de compilación al de ejecución [31]. Otros ejemplos de lenguajes dinámicamente tipados extensamente utilizados en la industria son Python y JavaScript.

En un lenguaje de tipado dinámico (PHP en este ejemplo), las siguientes dos sentencias conforman un programa válido. Nótese que no se declara un tipo para la variable. Se produce un cambio de tipo en la variable a medida que el intérprete procede a la ejecución.

```
$variable = 34;  // int

$variable = “hola mundo”;  // string
```

Por oposición, en un lenguaje estáticamente tipado los identificadores tienen un tipo declarado por el programador, y la asignación de un valor de distinto tipo puede resultar en una

[12] php.net (accedido 1/2020)

conversión de tipo implícita[13] o generar un error de tipo (*Type Error*) si no existen reglas implícitas para la conversión. Ejemplos de lenguajes estáticamente tipados son C, C++ y Java.

Sin embargo, en los últimos años ha habido un giro hacia la introducción de herramientas sintácticas que permiten acercar las características de PHP hacia el comportamiento clásico conocido de lenguajes orientados a objetos de tipado estático como Java o C#. Estos cambios surgen debido a su utilización en sistemas críticos y de alta escalabilidad, donde fue modificado específicamente para esos fines. El ejemplo más claro es el lenguaje Hack[14], desarrollado por Facebook a partir de PHP. La comunidad de desarrollo de PHP ha estado trasladando hacia él algunas ideas implementadas en Hack.

La ventaja de introducir estas posibilidades sintácticas radica en poder aumentar el chequeo de tipos (*Type Checking*), es decir, la capacidad de que el intérprete detecte si los tipos de los operandos se corresponden con aquellos compatibles para el operador utilizado [30]. Si se piensan los métodos como operadores y sus parámetros como los operandos, el chequeo de tipos significa la posibilidad de que el intérprete detecte en tiempo de ejecución las llamadas con tipos no compatibles y pueda emitir un error que advierta al desarrollador antes de la llamada, evitando que la operación falle dentro del método por recibir valores de tipos incompatibles o se produzcan cómputos con resultados sin sentido. Cuanto más posibilidades tenga el lenguaje de detectar errores de tipos se dice que es más *fuertemente tipado* (*Strongly typed*) [31].

PHP cuenta con la posibilidad sintáctica de declarar el tipo de los parámetros de un método desde su versión 5.0 (julio de 2004). En febrero de 2015 se aprobó la RFC "*Scalar Type Declarations*"[15], completando la evolución del lenguaje hacia la posibilidad de que el intérprete realice chequeos más estrictos de tipos. Este conjunto de cambios permite que en la declaración de parámetros de métodos se puedan específicar los tipos esperados, característica que los diseñadores del lenguaje llaman "*Type Hint*".

La RFC "*Return Type Declarations*"[16], implementada a partir de PHP 7.0 (diciembre de 2015), introduce la posibilidad de declarar un tipo para el retorno de un método, aunque sigue siendo opcional hacerlo al igual que en toda variable, según el espíritu original del lenguaje.

En la Tabla 3.1 se pueden observar las versiones y fechas en las que se introdujeron los cambios más importantes en relación al sistema de tipos. Puede verse como, con el tiempo, el desarrollo del lenguaje tiende a incorporar características para incrementar el chequeo de tipos, lo que eleva la fortaleza del tipado del lenguaje.

[13] Por ejemplo, en el lenguaje C al asignar un valor `int` dentro de una variable `float` se realiza una conversión implícita que "promueve" el `int` a `float`. Esto también es denominado "coerción" por los autores de referencia [31] [30].
[14] hacklang.org (accedido 1/2020)
[15] wiki.php.net/rfc/scalar_type_hints_v5 (accedido 1/2020)
[16] wiki.php.net/rfc/return_types (accedido 1/2020)

Versión	Lanzamiento	Novedades
1.0	6/1995	Primera versión del lenguaje por Rasmus Lerdorf
3.0	6/1998	Migración a un desarrollo comunitario (RFC)
5.0	7/2004	Introducción del intérprete Zend Engine 2 con un nuevo modelo de objetos que permite la orientación a objetos completa. *Type Hinting*.
5.3	6/2009	Espacios de nombres, closures
7.0	12/2015	Declaraciones de tipos de retorno y posibilidad de declarar parámetros con tipos escalares. Clases anónimas.
7.1	12/2016	Retornos tipo `void`
7.2	11/2017	Retornos y parámetros tipo `object`
7.4	11/2019	Declaración de tipos para variables miembro de una clase

Tabla 3.1: Evolución histórica en el sistema de tipos del lenguaje PHP

En suma, PHP es un lenguaje dinámicamente tipado, al que se le han agregado progresivamente características sintácticas que permiten mayor chequeo de tipos. Estos cambios están motivados por la utilización del lenguaje en contextos de escala creciente.

PHP soporta 10 tipos primitivos de datos [32]. De estos, cuatro son escalares:

- `boolean`
- `integer` (enteros signados de 64 bits)
- `float` (punto flotante de 64 bits)
- `string` (cadenas de caracteres de largo variable y dinámico)

Cuatro son compuestos:

- `array` (colección implementada como un Ordered Map[17])
- `object`
- `callable` (representa entidades que pueden ser “llamadas”, por ejemplo funciones anónimas)
- `iterable` (representa cualquier objeto que implementa la interfaz `Traversable`, es decir que será iterado con una estructura de control del tipo `foreach`)

Y dos se denominan “especiales”:

- `resource` (referencia a un recurso externo no manipulable directamente, por ejemplo un archivo)
- `NULL` (no-valor)

[17] Pueden encontrarse todos los detalles de implementación online en: www.php.net/manual/en/language.types.array.php (accedido 1/2020)

Existen diversas reglas de conversión implícita (coerción) entre los tipos. En la Figura 3.1 se ilustran, con ejemplos, las conversiones entre strings y números.

```
$prueba = 1 + "10.5";           // $prueba será float (11.5)
$prueba = 1 + "-1.3e3";         // $prueba será float (-1299)
$prueba = 1 + "abc-1.3e3";      // $prueba será integer (1)
$prueba = 1 + "abc3";           // $prueba será integer (1)
$prueba = 1 + "10 A B C";       // $prueba será integer (11)
$prueba = 4 + "10.2 A B C";     // $prueba será float (14.2)
```

Figura 3.1: Ejemplos de conversión entre strings y números. Adaptado de [32].

La gran cantidad de conversiones implícitas existentes entre estos y otros tipos atenta contra la posibilidad de detectar errores de tipos, y por ende, debilita la fortaleza del tipado[18].

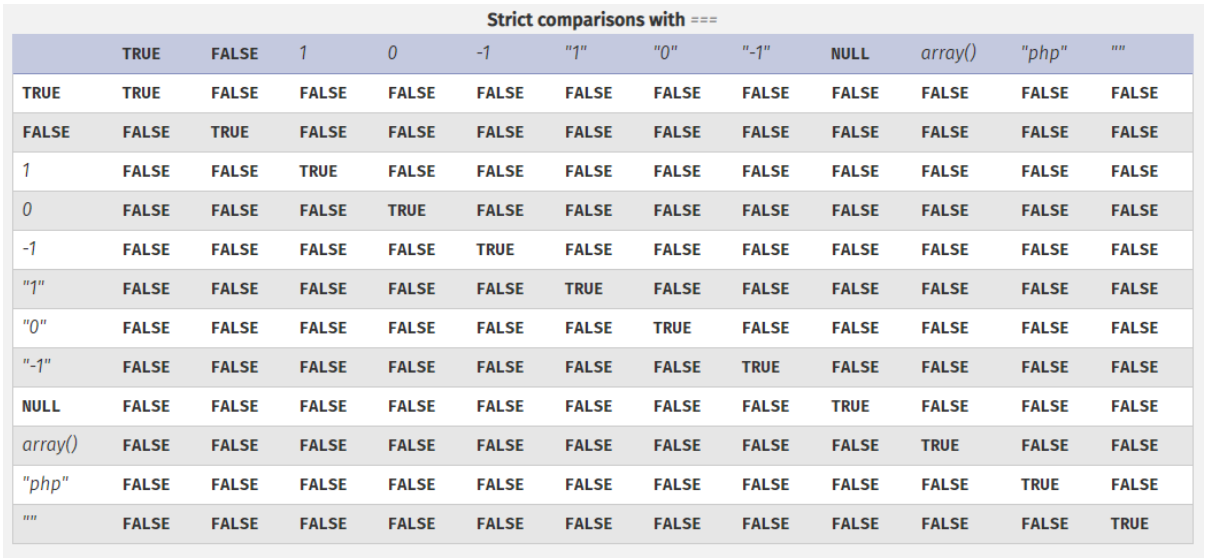
A los efectos de mostrar la gran cantidad de conversiones implícitas se presentan en la Figura 3.2 las reglas de comparación con el operador débil (*loose*) "==". Esto quiere decir que se aplican las reglas de conversión del lenguaje para proceder a la evaluación. En la Figura 3.3 se encuentran los resultados de la misma operación con el operador estricto (*strict*) "===" que, en cambio, realiza la comparación considerando los *tipos* de los valores.

Loose comparisons with ==

	TRUE	FALSE	1	0	-1	"1"	"0"	"-1"	NULL	array()	"php"	""
TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE
FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE
1	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
0	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE
-1	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
"1"	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"0"	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
"-1"	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
NULL	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE
array()	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE
"php"	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
""	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE

Figura 3.2: Reglas de comparación entre valores de distinto tipo para el operador débil (loose) "==". (Fuente: PHP Language Reference [32], Apartado "Type Comparison Tables")

[18] Sebesta [30] señala que *"los lenguajes con mayor nivel de coerción (como C y C++) son menos confiables que aquellos sin coerción (como ML y F#)"*. Apartado 6.13 en la referencia.



Strict comparisons with ===

	TRUE	FALSE	1	0	-1	"1"	"0"	"-1"	NULL	array()	"php"	""
TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
1	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
0	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-1	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"1"	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"0"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
"-1"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
NULL	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE
array()	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE
"php"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
""	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE

Figura 3.3: Reglas para el operador estricto (strict) "===". Nótese la diagonal principal. (Fuente: PHP Language Reference [32], Apartado "Type Comparison Tables")

Desde PHP 7 (diciembre de 2015) se puede solicitar al intérprete que no realice conversiones implícitas de tipo al realizar llamadas a función, lo que aumenta enormemente la fortaleza del tipado debido a la mayor cantidad de posibilidades de chequeo de tipos que ofrece. Esto se indica mediante una declaración específica en la primera línea del archivo fuente. En la Figura 3.4 se ilustra un ejemplo de esta característica.

```
// Tipado débil ("Weak Typing") original del lenguaje

function suma(int $a, int $b) {
    return $a + $b;
}

$prueba = suma(1, 2);       // int (3)
$prueba = suma(1.5, 2.5);   // int (3)   los parámetros se coercionan a int (!)



// Desde PHP 7, tipado estricto ("Strict Typing") genera una excepción

declare(strict_types=1);    // <-

function suma(int $a, int $b) {
    return $a + $b;
}

$prueba = suma(1, 2);       // int (3)
$prueba = suma(1.5, 2.5);   // TypeError: Argument 1 passed to suma() must be of
                            // the type integer, float given
```

Figura 3.4: Modos "débil" y "estricto" en el chequeo de tipos al llamar funciones. (Adaptado de PHP Language Reference [32], Apartado "Function Arguments")

Por el período de tiempo analizado en este trabajo no se han encontrado porciones de código que utilicen esta característica “estricta”, lo que representa una oportunidad de trabajo futuro.

3.2 Impacto del diseño del lenguaje en la orientación a objetos

A continuación se exponen algunas características particulares del lenguaje que tienen un impacto en el modo en que se encara la orientación a objetos. Es importante considerar las diferencias con lenguajes estáticamente tipados como Java o C++ puesto que las definiciones de métricas orientadas a objetos han sido tradicionalmente determinadas en función de éstos.

Debido a que PHP es un lenguaje interpretado posee características que complejizan o tornan imposible el análisis estático. Por ejemplo, es posible llamar a un método de esta manera:

```
$metodo = “getCurrentUserId”;  //esta variable contiene un string
$objeto->$metodo();   //se intentará invocar a un método con nombre getCurrentUserId
```

Esta posibilidad, que se asemeja a un puntero a función de C, complica el análisis estático porque el método efectivamente llamado en tiempo de ejecución depende del valor de una variable. La misma posibilidad sintáctica puede aplicarse al acceso a variables miembro[19]. Si bien esta característica permite mayor flexibilidad desde el punto de vista del programador, también complejiza la obtención de métricas de acoplamiento.

Por razones históricas no se pueden declarar dos métodos con el mismo nombre, tengan o no distintos parámetros en su firma. Esto se debe a que las llamadas a función se resuelven en tiempo de ejecución teniendo en cuenta solamente el nombre de la función. El siguiente ejemplo muestra dos llamadas que son igualmente válidas:

```
function prueba($param1) {
       /*  $param1 es un int que vale 1, para la primera llamada.
            Los otros dos valores son descartados  */

       /*  $param1 es un float que vale 4.5, para la segunda llamada  */
}

prueba(1, true, 2);
prueba(4.5);
```

Desde el punto de vista de la orientación a objetos, esto implica que no se puede hacer sobrecarga de métodos de la manera tradicional. Sin embargo, la sobrecarga de métodos puede implementarse a través de un método especial denominado `__call`, de la siguiente forma:

[19] Más detalles en www.php.net/manual/en/language.variables.variable.php (accedido 1/2020)

```
class A
{
    public function __call($nombre, $parametros)
    {
       // la variable $nombre es un string con el nombre del método llamado
       // la variable $parametros es un array conteniendo los parámetros enviados
    }
}

$a = new A();
$a->metodoInexistente1(5, “hola”);  //se ejecuta el método __call
$a->metodoInexistente2(true, 55.90, “abc”, 123);  //se ejecuta el método __call
```

Esta forma de implementar la sobrecarga complica el conteo de las métricas de acoplamiento y de complejidad, en el caso de aparecer en el código fuente de un producto.

Por otro lado, como consecuencia de la indicación opcional de tipos en los parámetros de los métodos, el polimorfismo es más dificil de detectar mediante análisis estático de código. A continuación se presentan dos ejemplos de métodos en donde la detección es posible e imposible, respectivamente:

```
class A {

       public function metodo1($objeto) {
              $objeto->count();  // (1)
       }

       public function metodo2(Countable $objeto) {
              return $objeto->count();  // (2)
       }

}
```

En `metodo1` no se declara el tipo del parámetro `$objeto`, por lo que no es posible determinar estáticamente su tipo. Esto impide conocer si (1) se trata de una llamada polimórfica. En `metodo2` sí se encuentra la declaración (o “*Type Hint*” según el vocabulario específico del lenguaje), y ésta corresponde a una interfaz llamada `Countable`[20]. En este caso sí es factible determinar que la llamada al método `count` (2) es una llamada polimórfica, debido a que el tipo declarado del objeto es una interfaz.

Estas consideraciones ponen de manifiesto que es necesario prestar atención a las características particulares del lenguaje de programación al momento de definir las métricas, algo ya señalado por Basili [5]. Características sintácticas que se utilizan en otros lenguajes para derivar algunas métricas pueden no estar presentes o no estarlo de manera consistente, lo que puede ser una amenaza a la validez. El objetivo central del presente trabajo es validar la

[20] Countable es una interfaz presente en la SPL (Standard PHP Library): un conjunto de clases, interfaces y funciones presentes de manera inmediata en el intérprete.

aplicabilidad de frameworks de métricas definidos para otros lenguajes al lenguaje del producto bajo estudio teniendo en cuenta sus diferencias estructurales.

En resumen, es necesario considerar las características propias del diseño del lenguaje al momento de analizar las amenazas a la validez (cfr. 6.4). Las métricas que caracterizan el acoplamiento y el polimorfismo podrían estar subestimadas por la imposibilidad de detectar estas características. Por otro lado, las características propias del tipado del lenguaje permiten proponer nuevas métricas, algo no posible para otros lenguajes (cfr. 4.2)

En el siguiente capítulo se presentan exhaustivamente las métricas seleccionadas en este trabajo, a los efectos de construir posteriormente modelos de regresión utilizándolas como variables independientes y predecir la probabilidad de que una clase sea defectuosa. Las métricas han sido adaptadas tomando en consideración las características del lenguaje descriptas en este capítulo.

4 Métricas seleccionadas

A continuación se describen, una por una, las métricas seleccionadas en este trabajo. Las mismas pueden dividirse en *métricas de sistema* y *métricas de clase*, siendo estas últimas las *métricas estáticas* (o de producto) y las *métricas de cambio* (o de proceso).

Las siguientes son las motivaciones que guían la selección de estas métricas:

- Las métricas estáticas C&K [1] son las más utilizadas y probadas del área (en lenguajes de tipado estático), como se explica en 2.2.1 y 2.3.2.1
- Las métricas estáticas Robiolo [2] representan una continuidad de la línea de investigación dentro de la misma Institución y son un buen complemento a C&K como se explica en 2.2.1
- Las métricas estáticas propias (Gullino) vienen a capturar la característica distintiva de este lenguaje de tipado dinámico respecto de los otros lenguajes mayormente estudiados hasta el momento: la posible ausencia de declaración de tipos en los parámetros y retornos de los métodos. En 6.5 se discuten los hallazgos[21]. Estas métricas no son posibles en lenguajes de tipado estático y son novedosas, delineando una rama de trabajos futuros.
- Las métricas de cambio de Moser [3] son las más importantes del área y se han validado en varios estudios, aunque en lenguajes de tipado estático, como se explica en 2.2.2.

Se ha elegido un nombre corto para cada una de ellas, como es habitual en la literatura, de manera de simplificar las tablas y la presentación de los datos en los apartados siguientes. Este capítulo funciona como referencia sobre la definición específica de cada una de las métricas.

4.1 Métricas de sistema

LOC: Lines of Code

Es la cantidad de líneas de código no vacías y no comentadas totales para el sistema, es decir, sumando todas las clases.

Del framework de métricas propuesto por Robiolo [2], se utilizaron las siguientes once métricas de sistema. Se han numerado consecutivamente según su aparición en el trabajo original de la autora.

[21] En síntesis, una de estas métricas participa del mejor modelo encontrado para métricas estáticas. No obstante, mayores conclusiones necesitarán de trabajo futuro con más datos.

Rob_1: Tamaño, clases

Es la cantidad de clases, concretas y abstractas, en todos los niveles de la jerarquía de herencia.

Rob_2: Tamaño, interfaces

Es la cantidad de interfaces.

Rob_3: Reutilización, clases externas especializadas

Es la cantidad de clases externas[22] que son extendidas, es decir, heredadas.

Rob_4: Reutilización, interfaces externas extendidas

Es la cantidad de interfaces externas que son extendidas, es decir, heredadas (no implementadas, cfr. Rob_5).

Rob_5: Reutilización, clases que implementan interfaces externas

Es la cantidad de clases que implementan interfaces externas.

Rob_6: Herencia, jerarquías de clase

Es la cantidad de clases raíz que poseen al menos una subclase.

Rob_7: Herencia, jerarquías de interfaz

Es la cantidad de interfaces raíz que son extendidas al menos por alguna otra interfaz

Rob_8: Especialización, niveles por jerarquía de clases

Es el nivel máximo de profundidad de herencia alcanzado desde las clases raíz

Rob_9: Especialización, niveles por jerarquía de interfaces

Se cuentan los niveles de las jerarquías de interfaces, considerando que es igual a la cantidad de interfaces, de la rama más profunda de la jerarquía, menos una interfaz.

[22] Las clases e interfaces se consideran “externas” si se encuentran en el directorio “*includes/libs*” o “*vendors*” (cfr. 5.3).

Rob_10: Generalización, clases raíz concretas

Es la cantidad de clases en el primer nivel de jerarquía que no son abstractas

Rob_11: Generalización, clases raíz concretas que implementan interfaces

Es la cantidad de clases en el primer nivel de jerarquía que no son abstractas y que implementan alguna interfaz

4.2 Métricas de clase: métricas estáticas

LOC: Lines of Code

Es la cantidad de líneas de código no vacías y no comentadas de una clase.

Del conjunto de métricas clásico presentado por Chidamber & Kemerer [1] se seleccionaron las siguientes tres.

CK_WMC: Weigthed Methods per Class

Es la cantidad de métodos definidos en una clase, sin contar los heredados. Se tomará el coeficiente "*weight*" con valor 1, al igual que Basili [5] y Gyimothy [6].

CK_CBO: Coupling Between Object Classes

Es la cantidad de clases a las cuales una clase está acoplada, esto es, accede a sus métodos o propiedades, estableciendo una dependencia con ellas en mayor o menor medida. Dadas las características de tipado del lenguaje el acoplamiento se podrá contabilizar, naturalmente, sólo cuando el tipo aparezca explícitamente declarado. Esto puede suceder:

1. cuando los parámetros de los métodos tienen el tipo declarado (*Type Hint*)
2. cuando se realicen llamadas a métodos de la superclase
3. cuando una clase realice una operación de instanciación de otra
4. cuando sean llamadas estáticas, donde se puede encontrar claramente el nombre de la otra clase
5. cuando el tipo de retorno esté declarado
6. cuando se atrapen excepciones (*try-catch*)

CK_RFC: Response For a Class

Es la cantidad de métodos de la clase, incluyendo los heredados, más la cantidad de métodos a su vez llamados en cada uno de estos métodos pertenecientes a la clase.

De las métricas C&K se han omitido `NOC` (*Number of Children*), `DIT` (*Depth of Inheritance Tree*) y `LCOM` (*Lack of Cohesion of Methods*) ya que existe consenso en que no aportan nueva información aplicable en la estimación de defectos [9] [11] [18] [19].

Del framework de métricas propuesto por Robiolo [2], se utilizaron las siguientes cinco métricas de clase.

Rob_12: Sobreescritura, métodos reemplazados en una jerarquía

Es el porcentaje de métodos concretos que sobreescriben un método concreto de la superclase y no lo llaman mediante *parent*. Se excluyen los constructores. Es el cociente de métodos reemplazados sobre el total de heredados (incluyendo aquellos sobreescritos).

El *override* se detecta por la presencia de métodos con el mismo nombre que en la superclase. Dadas las características del lenguaje, no pueden existir varios métodos con el mismo nombre (es decir, la sobrecarga clásica estilo Java, cfr. 3.2).

Rob_13: Mensajes polimórficos enviados

Es la cantidad de mensajes polimórficos enviados. Son los mensajes enviados utilizando variables declaradas de tipo interfaz o clase abstracta, o llamadas a métodos que no están declarados en el tipo del objeto sino que son heredados de alguna superclase.

Dado el declaración de tipo opcional de este lenguaje, sólo se podrán contabilizarán las llamadas a método sobre argumentos con tipos explícitamente declarados en el receptor. Se considera polimórfica toda llamada que corresponde a un método definido en la raíz de la jerarquía y enviado a una subclase de la misma.

Se ofrece el siguiente ejemplo:

```
class A {

    /* método público que recibe un parámetro de tipo B */
    public function Metodo1(B $param) {

        $param->Metodo2();   /* llamada a método */

    }
}
```

La llamada a `Metodo2` será contabilizada como polimórfica si se cumple alguna de estas condiciones:

- `B` es una interfaz
- `B` es una clase abstracta
- `Metodo2` no está declarado en la clase `B` sino que es heredado de su superclase

El hecho de poder contabilizar esta llamada polimórfica depende de que el parámetro `$param` esté declarado explícitamente de tipo `B`, algo opcional en este lenguaje (cfr. 3.1).

Rob_14: Granularidad, Promedio de statements por método

Es el promedio de sentencias que tienen los métodos de una clase. Se incluyen todos los métodos (concretos o abstractos) declarados en la clase, incluso el constructor y sean cuales sean sus visibilidades. No se incluyen los métodos heredados.

Rob_15: Responsabilidades públicas, métodos públicos por clase

Cantidad de métodos públicos de una clase, incluidos el constructor, los métodos abstractos y los heredados. Es similar a `CK_WMC` pero con la adición de los métodos heredados.

Rob_16: Responsabilidades públicas, métodos por interfaz

Cantidad de métodos declarados en una interfaz, propios y heredados. Es similar a la métrica anterior pero definida para interfaces.

De Robiolo [2] se debió descartar “Cantidad de colaboradores por clase”[23] ya que es de imposible cálculo en el lenguaje bajo estudio de una manera confiable. Implicaría inspección manual de código caso por caso, lo que no resulta materialmente factible.

Obsérvese el siguiente ejemplo tomado del archivo `Import.php`:

```
class WikiImporter {
       private $reader = null;
       /* ...  más variables ... */

       function __construct( $source ) {         /* constructor */
              $this->reader = new XMLReader();
              /* ...  */
```

[23] Apartado 3.17 en el trabajo original.

Nótese que la declaración de una variable de instancia (`$reader`) no tiene un tipo asociado estáticamente. Por otro lado también se puede apreciar un parámetro sin tipo en el constructor.

A los efectos de cuantificar las posibilidades sintácticas del lenguaje, en particular la presencia o ausencia de *Type Hints* (declaraciones de tipo en los métodos), se proponen en el presente trabajo las siguientes métricas nuevas (métricas Gullino).

Gul_1: Tipo de retorno no declarado y no documentado

Es la cantidad de métodos que no declaran un tipo de retorno ni tampoco lo documentan en el bloque de documentación que puede acompañar la declaración del método. No se considera el constructor.

El siguiente ejemplo demuestra un método con estas características:

```
class A {

    public function Metodo() {
       /* ... */
       return $i;
    }
}
```

Gul_2: Tipo de retorno no declarado y sí documentado

Es la cantidad de métodos que no declaran un tipo de retorno pero sí lo documentan en el bloque de documentación que puede acompañar la declaración del método. No se considera el constructor para el conteo de esta métrica.

En el siguiente ejemplo se encuentra un método que documenta el tipo de retorno:

```
class A {

    /*
    * @return B
    */
    public function Metodo() {
       /* ... */
       return new B();
    }
}
```

Gul_3: Tipo de retorno declarado

Es la cantidad de métodos en los que el tipo de retorno está explícitamente declarado. Para el período de análisis esta métrica es siempre nula, por lo cual no será utilizado. El código fuente producido a partir de 2015, momento en que se introdujo esta posibilidad de declaración en el lenguaje, empezará a contar con información como en el siguiente ejemplo:

```
class A {

    public function Metodo() : B {   /* tipo de retorno B, declarado explícitamente */
       /* ... */
       return new B();
    }
}
```

Gul_4: Tipo de parámetro no declarado y no documentado

Es la cantidad de parámetros de todos los métodos de una clase, incluido el constructor, que no especifican un tipo ni lo documentan. Esta es la forma histórica de trabajar con el lenguaje PHP. Con el correr del tiempo, la incorporación de la declaración opcional de los tipos (llamado *Type Hint*) hace que cada vez más los desarrolladores indiquen explícitamente un tipo.

A continuación se encuentra un método de ejemplo, con dos parámetros sin tipo declarado ni documentado, que incrementaría el conteo de esta métrica para esta clase en +2:

```
class A {

    public function Metodo($param1, $param2) {    /* parámetros sin tipo declarado */
       /* ... */
    }
}
```

Gul_5: Tipo de parámetro no declarado y sí documentado

Es la cantidad de parámetros de todos los métodos de una clase, incluido el constructor, que, si bien no tienen un tipo declarado, este sí puede deducirse del bloque de documentación que acompaña al método.

El siguiente ejemplo muestra un caso:

```
class A {

    /*
    * @param $param1 B
    */
    public function Metodo($param1) {
       $param1->Metodo2();   /* llamada a método sobre un objeto documentado de clase B */
    }
}
```

Gul_6: Tipo de parámetro declarado

Es la cantidad de parámetros de todos los métodos de una clase, incluido el constructor, que declaran el tipo de manera explícita.

A continuación se presenta un ejemplo para demostrar la sintaxis:

```
class A {

    public function Metodo(B $param1) {
       $param1->Metodo2();   /* llamada a un método sobre un objeto de clase B */
    }
}
```

4.3 Métricas de clase: métricas de cambio

Las métricas de cambio seleccionadas provienen de la importante investigación dentro del área realizada por Moser en 2008 [3]. En el Apartado 2.2.2 se describe dicho trabajo.

Respecto de las definiciones del paper original se ha determinado que las métricas de cambio se calculen a nivel de clase, cuando originalmente el autor trabaja simplemente con archivos.

Otra diferencia con el original ocurre con la métrica `changeset` y el concepto de "*co-changes"* (es decir, clases/archivos que cambian juntos). Dado que el proyecto MediaWiki trabaja con un repositorio Git (cfr. 5), el concepto de *co-change* viene a estar naturalmente soportado por el concepto de "commit", siendo éste justamente un agrupamiento de cambios a distintos archivos de manera simultánea. En el estudio original de Moser se usó un repositorio de versiones CVS, mucho más antiguo, por lo que debieron establecer que todo archivo modificado dentro de una ventana temporal de dos minutos pertenece al mismo "*changeset*", algo que ha resultado más sencillo en este trabajo y, si se quiere, más confiable por no ser una ventana de tiempo arbitraria.

Se ha descartado la métrica `Refactorings` del original porque no se ha encontrado un formato sistemático dentro de los mensajes de commit que pueda utilizarse para su deducción de

manera confiable. Es inviable un cálculo manual retrospectivo de esta métrica, por lo cual queda fuera de este estudio.

Debido a que las métricas de cambio analizan las modificaciones hechas al código fuente, este cálculo siempre tiene en cuenta un período de tiempo[24]. Las métricas estáticas, en cambio, representan una especie de "fotografía", a través de las métricas, del estado del código fuente en un punto temporal específico del repositorio.

A continuación se enumeran cada una de las métricas de cambio a utilizar en este trabajo.

Revisions

Cantidad de commits de tipo "*Feature Introduction*" (FI)[25] que modifican la clase dentro del período a considerar. Es un número entero positivo.

Bugfixes

Cantidad de commits de tipo "*Fault Repairing*" (FR) que modifican la clase dentro del período a considerar. Es un número entero positivo.

Authors

Cantidad de autores distintos que realizaron al menos un commit de tipo FI a la clase dentro del período. Es un número entero positivo.

LOC_Added

Cantidad total de líneas agregadas a una clase con commits de tipo FI, dentro del período. Es un número entero positivo.

Ejemplo: si una clase fuera modificada por dos commits de tipo FI y estos commits agregaran 10 y 30 líneas de código, respectivamente, la métrica resultaría 40.

Max_LOC_Added

Cantidad de líneas agregadas por el commit de tipo FI que más líneas agregó a la clase dentro del período considerado. Es un número entero positivo.

Ejemplo: si una clase fuera modificada por dos commits de tipo FI y estos commits agregaran 10 y 30 líneas de código, respectivamente, la métrica resultaría 30.

[24] En el presente trabajo se utilizan períodos trimestrales siguiendo a Hassan [4].
[25] En 4.5.2 se define la clasificación realizada a los commits.

Ave_LOC_Added

Promedio de líneas agregadas considerando todos los commits de tipo FI que modifican la clase, en el período. Es igual al cociente `LOC_Added` / `Revisions`. Es un número real positivo.

Ejemplo: si una clase fuera modificada por dos commits de tipo FI y estos commits agregaran 10 y 30 líneas de código, respectivamente, la métrica resultaría 20.

LOC_Deleted

Cantidad total de líneas eliminadas de una clase con commits de tipo FI, dentro del período. Es un número entero positivo.

Ejemplo: si una clase fuera modificada por dos commits de tipo FI y estos commits eliminaran 40 y 5 líneas de código, respectivamente, la métrica resultaría 45.

Max_LOC_Deleted

Cantidad de líneas eliminadas por el commit de tipo FI que más líneas eliminó de la clase dentro del período considerado. Es un número entero positivo.

Ejemplo: si una clase fuera modificada por dos commits de tipo FI y estos commits eliminaran 40 y 5 líneas de código, respectivamente, la métrica resultaría 40.

Ave_LOC_Deleted

Promedio de líneas eliminadas considerando todos los commit de tipo FI que modifican la clase, en el período. Es igual al cociente `LOC_Deleted` / `Revisions`. Es un número real positivo.

Ejemplo: si una clase fuera modificada por dos commits de tipo FI y estos commits eliminaran 40 y 5 líneas de código, respectivamente, la métrica resultaría 22,5.

CodeChurn

Es el resultado de restar `LOC_Added` - `LOC_Deleted` para una clase en el período. Es un número entero, que puede ser positivo o negativo.

Ejemplo: Una clase es modificada por dos commits de tipo FI. El primer commit agrega 10 líneas y elimina 40. El segundo commit agrega 30 y elimina 5. La métrica `CodeChurn` resultaría -5.

Max_CodeChurn

Valor absoluto mayor que tuvo la métrica `CodeChurn` para todos los commits de tipo FI realizados a la clase dentro del período. Es un número entero, que puede ser positivo o negativo.

Ejemplo: Una clase es modificada por dos commits de tipo FI. El primer commit agrega 10 líneas y elimina 40. El segundo commit agrega 30 y elimina 5. La métrica `Max_CodeChurn` resultaría -30.

Ave_CodeChurn

Promedio de las líneas agregadas menos las borradas en los commits de tipo FI que modifican la clase, en el período. Resulta de obtener `CodeChurn` / `Revisions`. Es un número real, que puede ser positivo o negativo.

Ejemplo: Una clase es modificada por dos commits de tipo FI. El primer commit agrega 10 líneas y elimina 40. El segundo commit agrega 30 y elimina 5. La métrica `Ave_CodeChurn` resultaría -2.5.

Ave_ChangeSet

En promedio, cuántas otras clases se modifican junto a ésta en el mismo commit, dentro del período. Se consideran los commits de tipo FI. Es un número real positivo.

Ejemplo: Supóngase una clase de nombre A. En el período existen dos commits de tipo FI que la modifican. El primer commit modifica a las clases A, B, C, D y E; y el segundo commit modifica a las clases A y C. La métrica `Ave_ChangeSet` para la clase A resultaría 2,5.

Max_ChangeSet

La mayor cantidad de otras clases que se modificaron junto a cierta clase mediante el mismo commit de tipo FI, dentro del período. Es un número entero positivo.

Ejemplo: Supóngase una clase de nombre A. En el período existen dos commits de tipo FI que la modifican. El primer commit modifica a las clases A, B, C, D y E; y el segundo commit modifica a las clases A y C. La métrica `Max_ChangeSet` para la clase A resultaría 4.

Age

Edad de la clase en semanas al momento de finalizar el período, contando desde su aparición en el repositorio. Es un número entero positivo.

Weighted_Age

Se calcula según la siguiente fórmula [3]:

$$Weighted\,Age = \frac{\sum_{i=1}^{N} Age(i) \times LOC\_ADDED(i)}{\sum_{i=1}^{N} LOC\_ADDED(i)}$$

Donde Age(i) es la edad en semanas que tenía la clase cuando el iésimo commit de tipo FI agregó LOC_ADDED(i) líneas de código. Se repite esto para los N commits de tipo FI dentro del período y se calcula el cociente. Es un número real positivo.

Ejemplo: Una clase es modificada por un commit de tipo FI que agrega 20 líneas de código. Al momento de ejecutarse este primer commit la edad de la clase es 4 semanas. Un segundo commit agrega 30 líneas de código cuando la clase tiene una edad de 6 semanas. La métrica `Weighted_Age` para esta clase dentro del período resulta 5,2. Esto surge de:

$$Weighted\,Age = \frac{(4 \times 20) + (6 \times 30)}{20 + 30} = 5,2$$

4.4 Tabla resumen de métricas

Granularidad	Tipo	Autor	Nombre
De sistema	---	N/D	LOC
		Robiolo	Rob_1
			Rob_2
			Rob_3
			Rob_4
			Rob_5
			Rob_6
			Rob_7
			Rob_8
			Rob_9
			Rob_10
			Rob_11
De clase	Estáticas	N/D	LOC
		Chidamber & Kemerer	CK_WMC
			CK_CBO
			CK_RFC
		Robiolo	Rob_12
			Rob_13
			Rob_14
			Rob_15
			Rob_16
		Gullino	Gul_1
			Gul_2
			Gul_3
			Gul_4
			Gul_5
			Gul_6
	De cambio	Moser	Revisions
			Bugfixes
			Authors
			LOC_Added
			Max_LOC_Added
			Ave_LOC_Added
			LOC_Deleted
			Max_LOC_ Deleted
			Ave_LOC_ Deleted
			Codechurn
			Max_Codechurn
			Ave_Codechurn
			Ave_Changeset
			Max_Changeset
			Age
			Weighted_Age

4.5 Herramientas desarrolladas y obtención de métricas

Al no encontrarse herramientas disponibles para obtener las métricas seleccionadas se debió proceder a construirlas. A continuación se enumeran las distintas aplicaciones que sirvieron de soporte para la obtención de los datos primarios, que luego son utilizadas como variables independientes para construir modelos de regresión y predicción.

4.5.1 Obtención de métricas estáticas

Para obtener las métricas estáticas del código fuente se construyó un programa que analiza el código fuente de cada una de las clases a considerar (alrededor de mil clases). Esta herramienta se realizó en lenguaje PHP aprovechando sus interesantes capacidades de reflexión, esto es, la posibilidad de un programa de acceder a, y eventualmente modificar, la propia estructura de alto nivel de sí mismo (por ejemplo: variables, funciones, clases, métodos, etc). Esta caracterísica generalmente se asocia con lenguajes interpretados o que utilizan máquinas virtuales, como Smalltalk, JavaScript o Java.

Al abrir con esta herramienta un archivo que contiene una declaración de clase, se logra que el intérprete la cargue en memoria, lo que provee de una interfaz orientada a objetos para consultar, por ejemplo: si es o no una clase concreta, cuántos métodos tiene y cómo se llaman, si existe herencia, cuáles son los parámetros de los métodos, entre varias otras.

El siguiente es un ejemplo de esta característica del lenguaje y la interfaz que provee:

```
$reflector = new ReflectionClass('ClaseDePrueba');

//se obtienen objetos que representan a los métodos de la clase
$methods = $reflector->getMethods();
```

Las operaciones más utilizadas en este programa que se ha construido son evaluaciones de expresiones regulares, debido a que la mayoría de las métricas se tratan de, por ejemplo, hacer conteos de llamadas a método para cuantificar acoplamientos (cfr. definición de métricas en Capítulo 4). Las expresiones regulares utilizadas pueden encontrarse en el Anexo D.

Haciendo uso del análisis con expresiones regulares línea por línea de cada método de cada clase y de las características de reflexión del lenguaje se han podido obtener las métricas estáticas requeridas como datos primarios. En el Anexo A se encuentran las tablas de métricas generadas con esta herramienta.

A los efectos de calcular las métricas estáticas del código fuente con esta herramienta construida, se posicionó el repositorio de versiones en dos puntos: primer cambio realizado en enero de 2014[26] y primer cambio realizado en julio de 2014[27]. Esto permite tener las métricas al inicio del primer y segundo semestre del año bajo análisis, respectivamente. Los comandos específicos del sistema de control de versión pueden encontrarse en los Anexos.

[26] Commit ee01799f47013d0d6dee73b9612e81eb82e3e460
[27] Commit cae5da1ca330021d966f5a81ec5b4760a3da0568

4.5.2 Obtención de métricas de cambio

Las métricas de cambio fueron mucho más trabajosas de obtener que las métricas estáticas del código fuente. Esto se debe, naturalmente, a que son métricas del proceso de desarrollo y, como tales, se debieron extraer del registro del trabajo de los desarrolladores que se encuentra en el repositorio de versiones, de alguna forma "desorganizado" para su procesamiento inmediato. En este repositorio se encuentran cada uno de los commits que realizó cada programador, indicando un "mensaje de commit" redactado por él y los cambios, línea por línea que se realizó en uno o varios archivos del sistema. Cada commit puede modificar varias líneas de varios archivos, simultáneamente.

En la Figura 4.1 se muestra el texto plano de un commit Git, a modo de ejemplo. En este texto de commit se encuentran en primer lugar el autor, las fechas y el mensaje de commit. Seguidamente se detallan los cambios concretos sobre el código fuente, que en este caso afectan un solo archivo (`QueryPage.php`) y lo modifican en tres lugares distintos (nótese el uso de los signos `@@` para encabezar cada cambio). Las líneas que comienzan con un signo menos (`-`) indican una línea eliminada y aquellas con un signo más (`+`) una línea agregada.

En primer lugar se obtuvo el registro total de los cambios, línea por línea, a los archivos del sistema desde el repositorio Git[28] para el período temporal bajo análisis (todo el año 2014). El archivo final conteniendo todos estos commits posee aproximadamente 500 mil líneas.

En segundo lugar se construyó un programa para realizar el parseo de estos cambios, de manera de organizar en una base de datos relacional todos los commits y los cambios que se realizaron a cada clase con cada uno de estos commits. El esquema de esta base de datos relacional se encuentra en la Figura 4.2.

```
commit e262d63d3eb08354e9d3939e7f3582b7d72a9803
Author:     umherirrender <umherirrender_de.wp@web.de>
AuthorDate: 2014-07-20
Commit:     IAlex <codereview@emsenhuber.ch>
CommitDate: 2014-07-27

    Remove return value from QueryPage::execute

    The parent SpecialPage::execute also has no return and not all return
    statements in QueryPage::execute actual return a value

    Change-Id: If7a38ca5ed1107b6a5a740acae54295534950696

diff --git a/includes/specialpage/QueryPage.php b/includes/specialpage/QueryPage.php
index 3a83d2bad8..b8fc05e5b6 100644
--- a/includes/specialpage/QueryPage.php
+++ b/includes/specialpage/QueryPage.php
@@ -470,7 +470,6 @@ abstract class QueryPage extends SpecialPage {
         * This is the actual workhorse. It does everything needed to make a
         * real, honest-to-gosh query page.
         * @param string $par
-        * @return int
         */
        function execute( $par ) {
                global $wgQueryCacheLimit, $wgDisableQueryPageUpdate;
@@ -488,7 +487,7 @@ abstract class QueryPage extends SpecialPage {
```

[28] Estos comandos pueden encontrarse en los Anexos

```
                if ( $this->isCached() && !$this->isCacheable() ) {
                        $out->addWikiMsg( 'querypage-disabled' );
-                       return 0;
+                       return;
                }

                $out->setSyndicated( $this->isSyndicated() );
@@ -578,8 +577,6 @@ abstract class QueryPage extends SpecialPage {
                }

                $out->addHTML( Xml::closeElement( 'div' ) );
-
-               return min( $this->numRows, $this->limit ); # do not return the one extra row, if exist
        }
```

Figura 4.1: Un commit del sistema de control de versiones Git

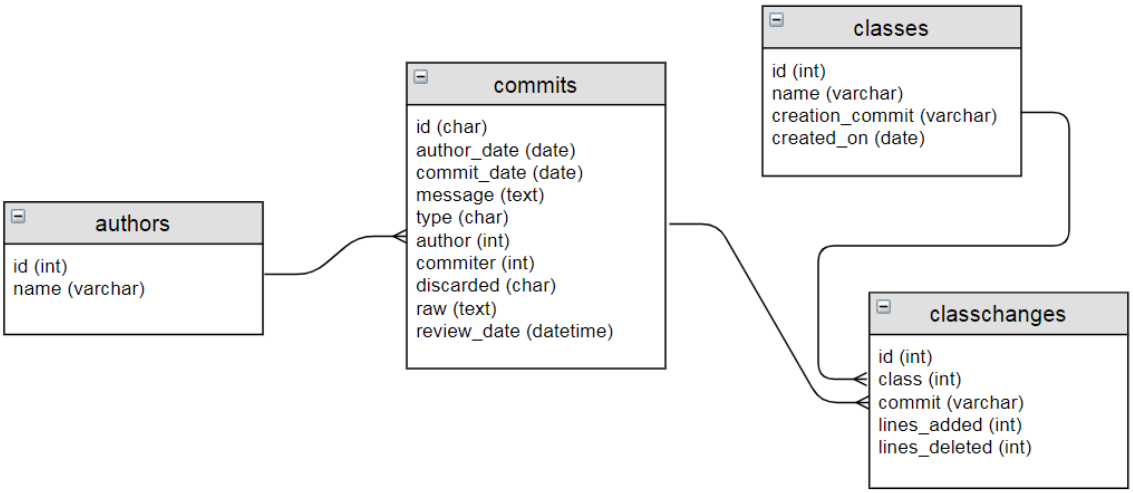


Figura 4.2: Esquema de la base de datos relacional de commits desarrollada

El programa que se utilizó para parsear los datos del repositorio creó 21700 filas en estas cuatro tablas.

En tercer lugar, se creó un programa (captura de pantalla presentada en Figura 4.3) para catalogar manualmente cada uno de los commits según las tres categorías de cambios [4] que se consideran en el presente trabajo:

- **Feature Introduction (FI):** cambio que incluye el agregado de nueva funcionalidad al sistema. Abarca todo cambio y refactorización del código que no se produce a consecuencia de un *bugfix*. En términos generales, es todo cambio en el código fuente que no pueda ser atribuito a las siguientes dos categorías.

- **Fault Repairing (FR):** toda corrección de algún defecto. Véase definición de defecto en el Apartado 2.3.1.
- **General Maintenance (GM):** actualización de números de versión, tabulación, corrección de estilo, corrección de documentación. En general, todo cambio no funcional al código fuente. Ejemplos incluidos en esta categoría: separación de sentencias confusas para mejorar la legibilidad del código, cambio de nombres de variables, unificación de if anidados, corrección de comentarios o errores gramaticales y *typos* en general.
- **Descartados:** cambios que no se realizan sobre clases sino sobre archivos auxiliares, por ejemplo archivos de configuración o de idiomas (traducciones).

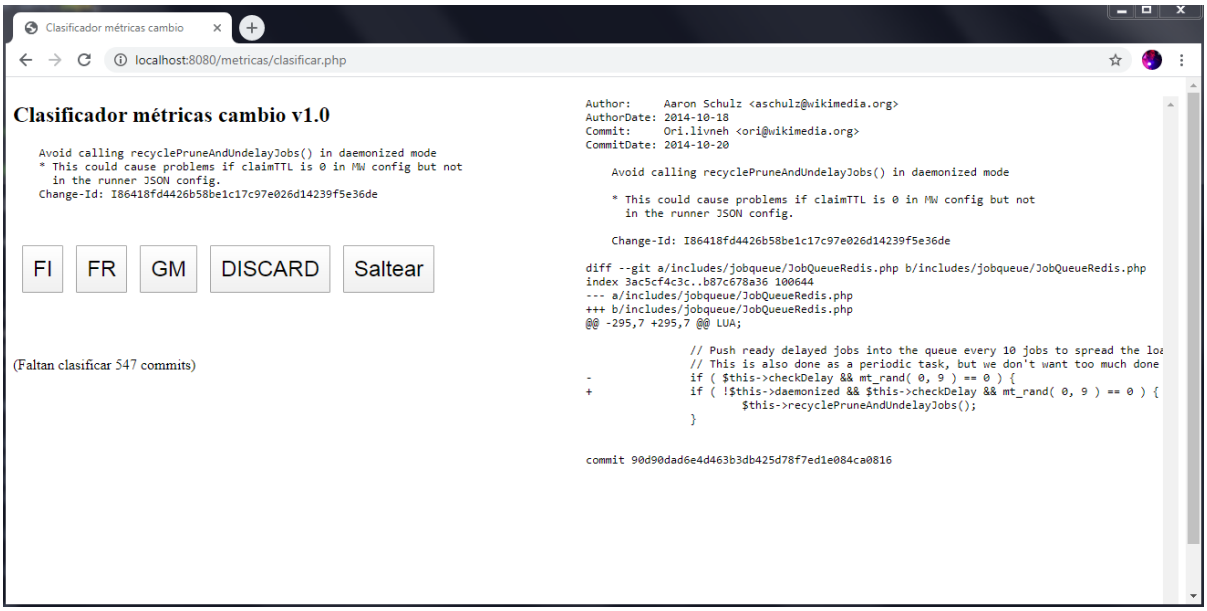


Figura 4.3: Captura de pantalla de la herramienta para clasificación de commits desarrollada

Se considera importante recalcar que debido a la ausencia de información confiable para realizar esta clasificación de manera automática, se optó por realizarla de forma manual. Para esto se construyó un programa a través del cual un operador calificado pueda examinar cada commit y decidir a qué tipo corresponde, analizando si fuera necesario los cambios en las clases involucradas línea por línea. De esta forma, un operador humano decidió a qué tipo corresponde cada commit haciendo una deducción de la intención de los cambios introducidos por ese commit en el código fuente y con la guía (a veces, ciertamente, escueta) del mensaje introducido por el autor en el propio commit. Esta tarea cognitiva sobre los 2871 commits analizados ocupó unas 60 horas reloj a lo largo de 15 días de trabajo y debe considerarse un alto costo dentro del proceso debido al conocimiento de la estructura del código fuente, la tecnología y la historia del producto que esta tarea requiere. Al respecto de la complejidad de esta catalogación se detallan algunos ejemplos en el Apartado 6.4 (Amenazas a la Validez).

Los siguientes números sumarizan el trabajo de esta etapa:

- Commits descartados: 263
- Commits no descartados: 2608, de los cuales se hallaron:
  - Feature Introduction: 1874 (72%)
  - Fault Repairing: 369 (14%)
  - General Maintenance: 365 (14%)

En cuarto lugar se realizó una depuración de los autores, debido a que varios de ellos se encontraban duplicados, por ejemplo con emails distintos, pero representando claramente al mismo desarrollador. De esta manera se corrigieron las relaciones en la base de datos para que el conteo de autores sea correcto.

En quinto lugar se realizó un procesamiento para encontrar la fecha de creación en el repositorio de cada clase. El comando Git específico puede encontrarse en los Anexos. Concluida esta tarea se detectaron inconsistencias (específicamente denotadas con una métrica age negativa), debido a que algunas clases estaban declaradas originalmente en un archivo fuente junto con otras clases relacionadas y fueron movidas a archivos exclusivos para ellas con posterioridad. Este hecho llevó a que la fecha de creación de una clase (es decir, de su archivo como declaración independiente) aparezca como posterior a la fecha en la que se le realizó algún cambio. Al detectar esta situación se debió buscar manualmente la fecha de introducción en el repositorio de 60 clases[29] para corregir esta situación, realizando una inspección manual del código fuente para cada una de ellas y evaluando su historia dentro del repositorio. Ésta también debe considerarse una operación sumamente costosa.

En sexto lugar se construyó un programa que, tomando todos los datos generados hasta el momento (es decir, la base de datos de cambios) computó las métricas de cambio según se definen en el Apartado 4.3. Este cómputo se realiza por trimestres siguiendo a Hassan [4].

En resumen, podemos describir con la Tabla 4.1 el desarrollo del trabajo para la obtención de las métricas de cambio.

[29] Las clases revisadas fueron: SpecialAllMessages, Action, ApiFormatFeedWrapper, Article, BadTitleError, BaseTemplate, CacheHelper, ChangesFeed, DoubleRedirectJob, EmailNotification, ErrorPageError, FatalError, HTMLCacheUpdateJob, ICacheHelper, ImagePage, JobQueueAggregatorRedis, JobQueueDB, JobQueueGroup, JobQueueRedis, JobSpecification, MediaWiki, MWDebug, MWException, WExceptionHandler , MWTidy, MWTimestamp, PoolCounter, PoolCounterRedis, PoolCounterWork, PoolWorkArticleView, QueryPage, RefreshLinksJob, SearchResult, SearchResultSet, SkinTemplate, SpecialAllPages, SpecialPage, SpecialPageFactory, UserMailer, UserNotLoggedIn, WikiFilePage, WikiPage, ImageHistoryList, ImageQueryPage, MWNamespace, Skin, TransactionProfiler, JsonContent, MapCacheLRU, RevDelArchivedFileItem, RevDelArchivedFileList, RevDelArchivedRevisionItem, RevDelArchiveItem, RevDelArchiveList, RevDelFileItem, RevDelFileList, RevDelList, RevDelLogItem, RevDelLogList y RevDelRevisionItem.

Etapa	Tarea	Método
1	Obtención de todos los cambios al código fuente producidos por los commits bajo estudio	Comandos Git
2	Parseo de los commits para construir una base de datos relacional	Programa ad hoc
3	Catalogación de cada commit en FI, FR ó GM	Inspección manual
4	Depuración de información de autores	Inspección manual
5	Extracción de la fecha de creación de cada clase	Programa ad hoc + inspección manual en 60 casos complejos
6	Cómputo final de las métricas de cambio	Programa ad hoc

Tabla 4.1: Pasos para la obtención de las métricas de cambio

Luego de seleccionar las métricas (de sistema, estáticas y de cambio) se han desarrollado las herramientas para obtenerlas, como se describe precedentemente en este capítulo. Utilizando estas herramientas se han obtenido las métricas para un Caso de Estudio, presentado en el capítulo siguiente.

5 Caso de Estudio: MediaWiki

En este capítulo se presenta brevemente la historia del producto bajo estudio y se describe sumariamente la metodología de desarrollo que utiliza la organización. El objeto de esta presentación es proveer de información contextual que facilite la discusión de los hallazgos posteriores.

MediaWiki fue elegido como caso de estudio para el análisis empírico del presente trabajo debido a que proporciona públicamente toda la información histórica sobre el código fuente, a través del repositorio de versiones y las distintas herramientas de comunicación de los desarrolladores. Además, se trata de un producto de indiscutible calidad y exitoso desde el punto de vista funcional.

5.1 Características generales

MediaWiki[30] es un sistema wiki de código abierto cuya principal instalación es el sitio web Wikipedia[31]. Los sistemas *wiki* (vocablo hawaiano que significa “rápido”) son sistemas de información online cuyo contenido es editado por los mismos usuarios de manera colaborativa. Es desarrollado por la Wikimedia Foundation[32], con sede en San Francisco, Estados Unidos, desde 2003. Actualmente hay decenas de miles de instalaciones de este producto en todo el mundo, según refiere el propio sitio oficial[33].

La infraestructura de Wikipedia incluye cinco *clusters* de servidores, ubicados en Estados Unidos, Amsterdam y Singapur. Estos *clusters* atienden entre 200 y 300 millones de visitas diarias[34] sólo para el idioma inglés y el sistema se posiciona en el décimo lugar de los sitios más visitados a nivel global[35]. MediaWiki está desarrollado en lenguaje PHP y utiliza la base de datos MariaDB[36] (previamente MySQL).

El esquema de licenciamiento de MediaWiki es GPL[37] versión 3. Esto permite que sea utilizado sin costo por muchas organizaciones que, si bien no están relacionadas con la Fundación, comparten el concepto de construcción colaborativa del conocimiento.

[30] www.mediawiki.org (accedido 1/2020)
[31] es.wikipedia.org (accedido 1/2020)
[32] wikimediafoundation.org (accedido 1/2020)
[33] www.mediawiki.org/wiki/MediaWiki (accedido 1/2020)
[34] Según Siteviews: tools.wmflabs.org/siteviews (accedido 1/2020)
[35] Según Alexa: www.alexa.com/siteinfo/wikipedia.org (accedido 1/2020).
[36] mariadb.org (accedido 1/2020)
[37] GNU General Public License, disponible online en www.gnu.org/licenses/gpl-3.0.html (accedido 1/2020)

5.2 Método de desarrollo

MediaWiki es desarrollado siguiendo un modelo de integración continua, donde los cambios en el código fuente se despliegan directamente en los sitios de Wikimedia Foundation, tales como Wikipedia, de forma regular.

La infraestructura de integración continua utiliza la herramienta Jenkins[38] para construir el producto. El uso principal es la ejecución de las pruebas automáticas cuando se incorporan cambios provenientes del sistema Gerrit[39], una herramienta de revisión de código entre pares. Es decir que el código enviado al repositorio del proyecto por los desarrolladores es revisado por un par, aprobado, y luego integrado en el producto, que a su vez se despliega en los servidores productivos rápidamente, de una manera continua.

Cada seis meses se crea una versión del producto con numeración correlativa. Este corte temporal se desprende del repositorio principal del producto (o *master*), creando una "rama" (o *branch*) en el repositorio. Esta versión se publica para quien desee instalar el sistema en su servidor de manera independiente, de manera que no participa en el proceso de integración continua. Estas versiones reciben actualizaciones de seguridad y correcciones de defectos durante un año.

De manera similar a otros proyectos, cada dos años se construye una versión denominada LTS (*Long Term Support*) que es mantenida durante tres años. Se garantiza un solapamiento de un año con la siguiente versión. Por ejemplo, la versión LTS 1.23 fue lanzada a mediados de 2014, con soporte hasta mediados de 2017. A mediados de 2016 se lanzó la versión LTS 1.27, lo que otorga a los usuarios un año para realizar la migración. En la Figura 5.1 puede observarse un gráfico con el plan de versiones de la organización.

5.3 Organización del código fuente

En la raíz del directorio principal se encuentran los archivos que contienen el código fuente PHP ejecutado a través de peticiones enviadas por los clientes HTTP. Estos programas se denominan "*access points*" ya que son los accesibles[40] desde los navegadores. En la Figura 5.2 se encuentra una muestra del árbol de directorios del proyecto.

[38] jenkins.io (accedido 1/2020)
[39] www.gerritcodereview.com (accedido 1/2020)
[40] Debido a que se trata de una aplicación web, aquí "accesibles" quiere decir que de manera indirecta, las URL presentes en los paquetes HTTP GET o POST enviados a los servidores provocarán una ejecución de estos programas, que se encargarán de procesar tal petición y devolver al cliente una página web en formato HTML. Son los puntos de entrada al sistema.

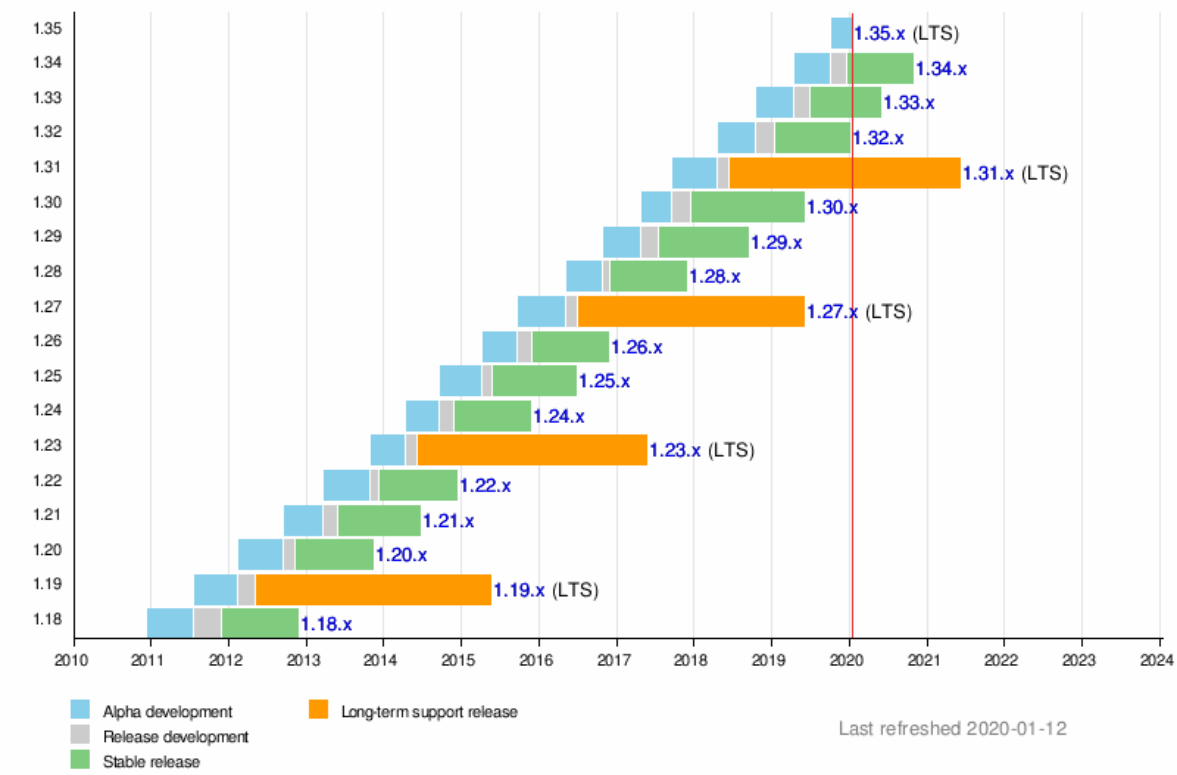


Figura 5.1: Plan de versiones del proyecto MediaWiki
(Fuente: mediawiki.org, enero de 2020)

Las declaraciones de clase se encuentran en el directorio “includes”, el más importante para el presente trabajo. Los archivos contenidos en este directorio son los analizados para obtener las métricas estáticas. Los cambios (deducidos del repositorio) a estos mismos archivos son los analizados para obtener las métricas de cambio. Dentro de includes se encuentra, organizado en múltiples carpetas, el código fuente de las aproximadamente 1000 clases que componen MediaWiki. En el Apartado 6.1 se describen las métricas de sistema, a los efectos de realizar una caracterización de alto nivel del producto.

Figura 5.2: Muestra del árbol de directorios de MediaWiki

En las versiones más modernas de MediaWiki se incluye un directorio “vendors”, donde se encuentran las bibliotecas de terceros, es decir, clases externas. Esta práctica está en concordancia con la comunidad de desarrollo PHP, que ha adoptado este nombre de directorio como convención. En versiones anteriores, las bibliotecas externas se distribuian conjuntamente con el sistema en la carpeta “includes/libs”. Las clases en estos dos directorios se consideran independientes de la base de código de MediaWiki. Pueden ser utilizados en otros proyectos sin problemas de dependencia.

A partir de la versión 1.25, creada a mediados de 2015, MediaWiki utiliza el gestor de dependencias Composer[41], una herramienta que se ha convertido en el estándar de facto en la industria cuando se desarrolla con estas tecnologías. Esta herramienta posibilita la gestión de versiones y la resolución de dependencias de bibliotecas externas de una manera automatizada a través de un manifiesto, de la misma forma que herramientas como Maven[42] (para lenguaje Java) o NPM[43] (para lenguaje JavaScript).

5.4 Sistema de control de versiones

El sistema de control de versiones utilizado actualmente para el desarrollo de MediaWiki es Git[44]. Este es un sistema distribuido, diseñado por Linus Torvalds para proyectos con gran cantidad de archivos y de colaboradores, pensado fundamentalmente para la eficiencia. Actualmente es el sistema de control de versiones más utilizado en el ámbito del software de

[41] Disponible online en getcomposer.org (accedido 1/2020)
[42] maven.apache.org (accedido 1/2020)
[43] www.npmjs.com (accedido 1/2020)
[44] git-scm.com (accedido 1/2020). Han utilizado anteriormente CVS y SVN. Toda la información histórica ha sido migrada.

código abierto. Por ejemplo, es el sistema a través del cual se gestionan los cambios en el código del kernel del sistema operativo Linux. GitHub[45], la empresa de hosting de repositorios más importante, cuenta actualmente con 100 millones de repositorios y 36 millones de usuarios.

Los distintos cortes temporales (*releases*) de MediaWiki se implementan por medio de “tags” en el repositorio Git. Además se crea un *branch* independiente para cada una de estas ramas (1.23, 1.24, etc.). Esto posibilita aplicar cambios a *releases* anteriores cuando se encuentran defectos importantes en versiones posteriores, además de mantener un esquema de versionado consistente para los usuarios que no realizan integración continua.

En el repositorio Git[46] se encuentra toda la historia de desarrollo del producto desde la primera versión de MediaWiki. Son cambios introducidos por 635 programadores de distintos países a lo largo de 15 años de trabajo.

5.5 Sistema de seguimiento de defectos

Hasta noviembre de 2014, los desarrolladores de MediaWiki utilizaron varias herramientas para el seguimiento de defectos, la administración del proyecto, la revisión del código, la gestión de tareas y la construcción de versiones. Algunas de estas herramientas eran de código abierto y otras eran propietarias. Algunas estaban alojadas en servidores propios y otras de terceros. La cantidad de distintas herramientas y canales de comunicación dificultaban el seguimiento de las tareas, y poder tener un panorama completo de lo que sucedía en el desarrollo se tornaba casi imposible. Esta multitud de herramientas atentaba contra la posibilidad de recibir nuevos desarrolladores e integrarlos rápidamente al flujo de trabajo, algo fundamental en las comunidades de código abierto.

Bugzilla, la herramienta utilizada para el seguimiento de defectos, está desarrollada en el lenguaje PERL. Gerrit, la herramienta para realizar *pair code review* está desarrollada en Java. Dado que la mayoría de los desarrolladores de MediaWiki son fluentes en PHP, se tornaba dificil poder contribuir a esos proyectos cada vez que fuera necesario realizar alguna tarea de integración o cambio para mejorar el proceso de desarrollo.

En definitiva, el conjunto de herramientas era muy grande:

- gitblit[47]: administrador de repositorios Git basado en Java
- Gerrit: herramienta de *code review,* desarrollado en Java
- Jenkins: servidor de automatización de *building* y *testing* basado en Java
- Bugzilla: herramienta de reporte de errores, desarrollado en PERL
- Trello[48]: software de administración de proyectos, sistema propietario online
- Mingle[49]: software de administración de proyectos, sistema propietario discontinuado

[45] github.com (accedido 1/2020)
[46] Disponible online en github.com/wikimedia/mediawiki (accedido 1/2020)
[47] gitblit.com (accedido 1/2020)
[48] trello.com (accedido 1/2020)
[49] mingle.thoughtworks.com (accedido 1/2020)

Como consecuencia, se decidió desarrollar una herramienta propia que integrara todos estos procesos: se la llamó Phabricator[50]. Esta herramienta está en uso hasta el día de hoy. Otros usuarios de Phabricator incluyen a Facebook (red social), AngularJS (framework para JavaScript), Blender (software de modelado 3D), FreeBSD (sistema operativo abierto), Haskell (lenguaje de programación), Uber (empresa de servicios de transporte), Quora (red social), KDE (interfaz gráfica de escritorio), entre otros.

La Figura 5.3 muestra una captura de pantalla de la herramienta siendo usada actualmente.

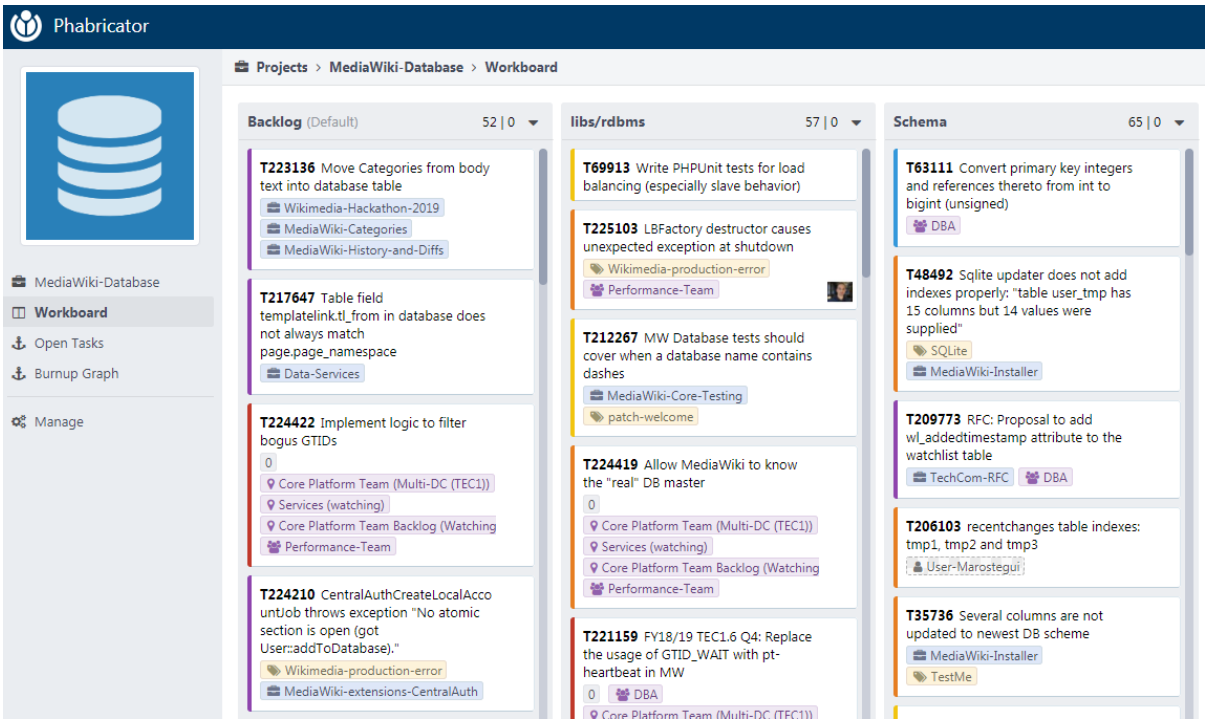


Figura 5.3: Una vista de la herramienta Phabricator mostrando tareas de MediaWiki en desarrollo (Fuente: phabricator.wikimedia.org, agosto de 2019)

En suma, Phabricator es una herramienta para el desarrollo colaborativo, fuertemente orientado al código abierto. Incluye funcionalidad para revisión de código entre pares, explorador de repositorios, seguimiento de defectos, administración de tareas, canales de chat y una wiki.

En el próximo capítulo se presentan las métricas de sistema y se procede a la construcción de los modelos de regresión logística que permiten obtener predictores de defectos tanto con métricas estáticas como con métricas de cambio.

[50] www.phacility.com (accedido 1/2020)

6 Análisis y Resultados

En este capítulo se presentan primeramente las métricas de sistema (6.1), a los efectos de dimensionar el producto y proveer de información contextual que permita posteriormente interpretar los resultados. A continuación se estudian las métricas estáticas y luego las métricas de cambio. Para cada uno de estos dos juegos de métricas se procede del mismo modo: se analiza la correlación de Pearson (cfr. 6.2) y se construyen distintos modelos con variables no correlacionadas. Para cada uno de los modelos predictivos se calculan los indicadores detallados en 6.3, lo que permite compararlos y deducir de ello cuál es el mejor predictor de defectos.

Se han obtenido las métricas estáticas y de cambio del proyecto MediaWiki tomando como marco temporal el año 2014.

Las métricas estáticas fueron calculadas sobre el código fuente correspondiente al día 1 de enero y al 1 de julio. Esto provee de dos “fotografías” del código fuente al inicio de cada semestre del año (cfr. Figura 6.1).

Las métricas de cambio fueron calculadas para los primeros tres trimestres del año. La métrica `bugfix` se calculó para los cuatro trimestres del año. De esta forma puede trabajarse con las métricas de un trimestre con el objetivo de predecir los bugfixes del siguiente. Por esta razón no se trabajó con las métricas de cambio del cuarto trimestre, ya que esto implica obtener los bugfixes del primer trimestre del año 2015, algo que queda fuera del alcance de este trabajo.

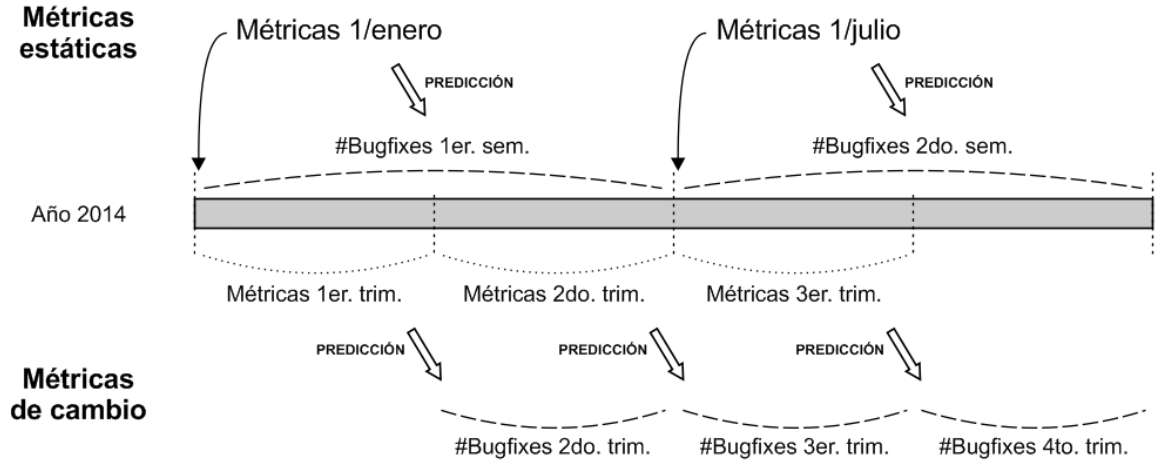


Figura 6.1: Representación de las métricas obtenidas y cómo se utilizan para predecir los defectos futuros (“bugfixes”). Los bugfixes son las modificaciones FR realizadas a una clase.

En el Anexo A se encuentran las tablas con las métricas obtenidas, tanto estáticas como de cambio, para cada clase, lo que constituye los datos primarios de esta investigación.

6.1 Métricas de sistema

La Tabla 6.1 presenta las métricas de sistema definidas en el framework Robiolo [2] y algunos conteos relevantes, a los efectos de dimensionar el producto bajo análisis.

Métrica	**Descripción**	**1/enero/2014**	**1/julio/2014**
System LOC	Líneas lógicas de código totales	364.224	365.696
Rob_1	Cantidad de clases internas	922	941
Rob_2	Cantidad de interfaces internas	24	28
---	Cantidad de clases externas	22	31
---	Cantidad de interfaces externas	0	0
---	Cantidad de clases e interfaces totales	968	1000
Rob_3	Reutilización: clases externas especializadas	1	1
Rob_4	Reutilización: interfaces externas extendidas	0	0
Rob_5	Reutilización: clases que implementan interfaces externas	0	0
Rob_6	Herencia: Jerarquías de clase	80	82
Rob_7	Herencia: Jerarquías de interfaz	0	0
Rob_8	Especialización: niveles por jerarquía de clases	5	5
Rob_9	Especialización: niveles por jerarquía de interfaces	2	2
Rob_10	Generalización: clases raíz concretas	245	245
Rob_11	Generalización: clases raíz concretas que implementan interfaces	45	46
Clases modificadas	Cantidad de clases que fueron modificadas con al menos un commit de tipo FI en el semestre	459	566
Clases no modificadas	Cantidad de clases que no fueron modificadas con ningún commit de tipo FI en el semestre	509	434
Clases sin defectos	Cantidad de clases que no fueron modificadas por ningún commit de tipo FR en el semestre	823	866
Clases con defectos	Cantidad de clases que fueron modificadas con al menos un commit de tipo FR en el semestre	145	134

Tabla 6.1: Métricas de sistema obtenidas de MediaWiki

En líneas generales se verifica que sólo un 15% de las clases han precisado correcciones de defectos durante el período analizado. Las clases modificadas por commits de tipo *"Feature Introduction"*, sin embargo, son aproximadamente el 50% y abarcan pocas líneas de código (cfr. con el análisis de la métrica `codechurn` a continuación).

Respecto de un semestre al siguiente, se evidencia un leve crecimiento en la cantidad de clases del sistema (3%) en seis meses, comparado con un cambio mínimo en las `LOC` (0,4 por mil). También crece en este período la cantidad de interfaces.

El lenguaje PHP no es utilizado mayoritariamente para proyectos de gran escala sino para la construcción de sistemas web de pequeña a mediana envergadura. MediaWiki se trata de un proyecto de mediana escala. Por ejemplo, sistemas de gran escala populares como Moodle o LimeSurvey presentan unos 13 millones de tokens[51]. Sistemas medianos presentan unos 3 millones de tokens, como Drupal, Joomla o MediaWiki[52].

La Tabla 6.2 presenta el resumen de las métricas por clase (968 casos, enero 2014) más descriptivas del sistema en función de analizar su diseño general.

Métrica	Descripción	Mínimo	Máximo	Media	Mediana	Desv. Est.
LOC	Líneas de código lógicas de la clase	2	3639	376	196	451,4
Rob_15	Cantidad de métodos públicos, incluidos los heredados	0	208	42	25	41,2
CK_CBO	Cantidad de clases a las cuales una clase está "acoplada"	0	36	3	2	4,5

Tabla 6.2: Métricas de clase más descriptivas del diseño general (datos primer semestre)

La Figura 6.2 muestra el histograma de la métrica `LOC`, donde puede observarse que la mayoría de las clases tiene menos de 500 líneas, lo que indica un diseño, en principio, deseable.

Sólo una clase consta de más de 3000 líneas (Parser, con el máximo `LOC` de 3639), siendo las siguientes Title y User, con 2644 y 2638 respectivamente.

Respecto de la cantidad de métodos públicos por clase (métrica denominada en este trabajo como `Rob_15`) puede observarse en el histograma de la Figura 6.3 que, de manera análoga,

[51] *Tokens*: elementos mínimos del léxico del lenguaje, que son parseados antes de comenzar la ejecución del programa por el intérprete. Son los componentes del árbol sintáctico. Contarlos es una forma de medir el tamaño de un proyecto sin importar el paradigma de programación utilizado por el desarrollador. Por ejemplo, el siguiente programa: `$b = $a + 10;` consta de 5 tokens.
[52] Disponible online en www.exakat.io/largest-php-applications-2018 (recuperado 8/2019)

la mayoría de las clases poseen pocas responsabilidades públicas (menos de 50), lo que también resulta adecuado. La media es de 42 con una desviación estándar de 41.

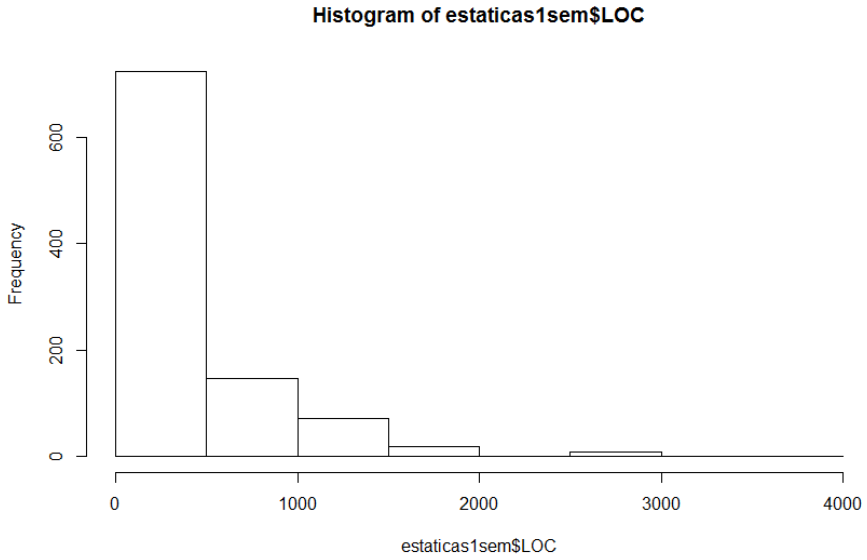


Figura 6.2: Histograma de métrica LOC por clase (datos primer semestre)

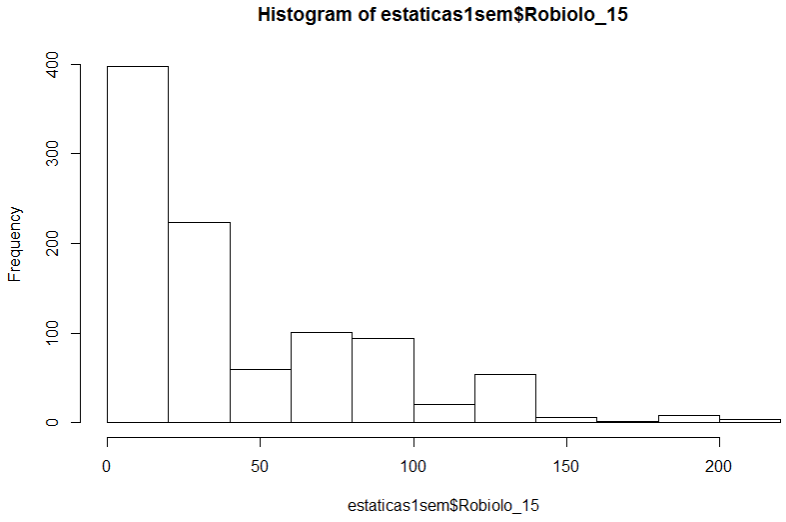


Figura 6.3: Histograma de métrica Rob_15 (datos primer semestre)

Respecto del acoplamiento entre clases (la métrica `CK_CBO` del conjunto definido por C&K), la mayor parte está acoplada con menos de 5 clases (Figura 6.4), siendo la media de 3, con una desviación estándar de 4,5. Lo descripto resulta también satisfactorio dada la cantidad de clases e interfaces totales del sistema (968 en el primer semestre).

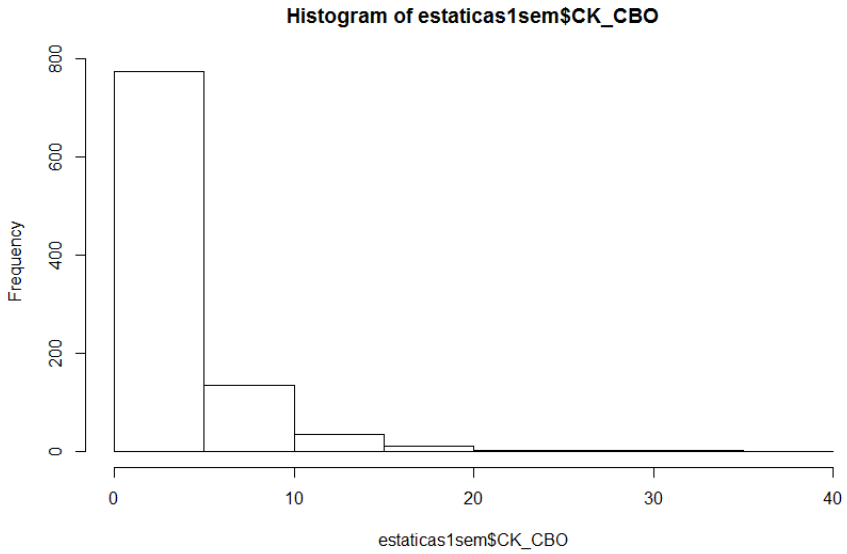


Figura 6.4: Histograma de métrica CK_CBO (datos primer semestre)

Se puede concluir que, dadas las métricas de sistema y tomando en consideración estas tres métricas descriptivas, el sistema en su conjunto y a nivel macro parece estar diseñado a priori de manera adecuada, y las responsabilidades distribuidas de manera satisfactoria.

En cuanto a los cambios introducidos en el código fuente durante el año 2014, se puede visualizar a través de la distribución de la métrica `codechurn` (líneas agregadas menos líneas borradas, para cada clase) que la mayoría de los cambios realizados abarcan pocas líneas de código (Figura 6.5 y Tabla 6.3).

Dado el período bajo análisis (código fuente y modificaciones a éste durante el año 2014) se trata de un producto con diez años de desarrollo, debido a lo cual se encuentra en su etapa de madurez. Se observa estabilidad en el código fuente y en su diseño general.

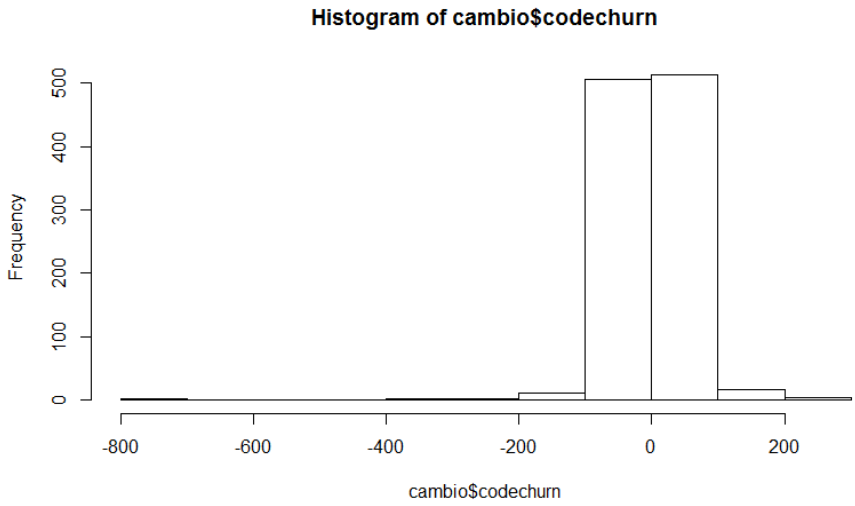


Figura 6.5: Histograma de métrica Codechurn (datos de todo el año 2014)

Métrica	Mínimo	1er cuartil	Mediana	Media	3er cuartil	Máximo
Codechurn	-795	-3	1	1,242	10	276

Tabla 6.3: Distribución de la métrica Codechurn (datos de todo el año 2014)

6.2 Análisis de Correlación

6.2.1 Correlación en métricas estáticas

En la Tabla 6.4 se muestra la matriz de correlación (Pearson) para las métricas estáticas.

	LOC	CK_WMC	CK_RFC	CK_CBO	Rob_12	Rob_13	Rob_14	Rob_15	Gul_1	Gul_2	Gul_4	Gul_5	Gul_6
LOC	1,00	0,43	0,21	0,34	-0,05	0,09	0,12	0,00	0,30	0,38	0,40	0,23	0,13
CK_WMC		1,00	0,59	0,66	0,00	0,14	0,05	0,43	0,68	0,89	0,84	0,45	0,40
CK_RFC			1,00	0,66	-0,02	0,11	0,15	0,90	0,50	0,46	0,46	0,24	0,22
CK_CBO				1,00	0,01	0,25	0,32	0,38	0,40	0,61	0,52	0,31	0,41
Rob_12					1,00	0,00	-0,08	0,00	0,09	-0,06	-0,03	-0,03	0,05
Rob_13						1,00	0,07	0,01	0,04	0,16	0,12	0,15	0,29
Rob_14							1,00	0,03	0,01	0,06	0,07	0,06	0,03
Rob_15								1,00	0,42	0,30	0,32	0,13	0,13
Gul_1									1,00	0,27	0,62	0,19	0,17
Gul_2										1,00	0,72	0,47	0,42
Gul_4											1,00	0,23	0,24
Gul_5												1,00	0,12

Tabla 6.4: Matriz de correlación de métricas estáticas

Se puede observar que `Rob_12` (porcentaje de métodos reemplazados) y `Rob_13` (mensajes polimórficos enviados) son las métricas que menos correlación presentan con las demás variables.

Es notable que `Rob_15` (cantidad de métodos públicos, incluyendo los abstractos y heredados) esté mucho más correlacionada (0,9) con `CK_RFC` (cantidad de métodos más la cantidad de métodos que son llamados por éstos) que con `CK_WMC` (cantidad de métodos por clase sin contar los heredados) dado que esta última parece una métrica más similar a `Rob_15` que la anterior. Esto se deba probablemente a qué los métodos heredados (contemplados en `Rob_15`) son llamados en las subclases, lo que incrementa el `CK_RFC`.

La métrica `LOC` no está correlacionada muy fuertemente con ninguna de las otras métricas, lo que resulta de algún modo sorprendente. La correlación más fuerte (0,43) viene dada con `CK_WMC`, lo que tiene sentido ya que a mayor cantidad de métodos naturalmente incrementará `LOC`.

Las métricas C&K presentan una correlación significativa entre sí, lo que indica que miden aspectos similares del sistema.

La métrica Gul_4 (cantidad de parámetros sin tipo declarado ni documentado) está fuertemente correlacionada con Gul_1 (cantidad de métodos que no declaran ni documentan el tipo de retorno) y Gul_2 (cantidad de métodos que no declaran pero sí documentan el tipo de retorno).

Observando la tabla puede determinarse que las métricas poco correlacionadas, factibles de ser usadas en un modelo de regresión son LOC, CK_RFC, Rob_12 (porcentaje de métodos reemplazados), Rob_13 (mensajes polimórficos enviados), Rob_14 (granularidad), Gul_5 (parámetros con tipo no declarado y sí documentado) y Gul_6 (parámetros con tipo declarado).

6.2.2 Correlación en métricas de cambio

En la Tabla 6.5 se encuentra la matriz de correlación (Pearson) para las métricas de cambio.

	revisions	loc_added	max_loc_added	ave_loc_added	loc_deleted	max_loc_deleted	ave_loc_deleted	codechurn	max_codechurn	ave_codechurn	authors	ave_changeset	max_changeset	age	weighted_age	bugfixes	bugfixes_prox_periodo
revisions	1,00	0,56	0,38	0,15	0,50	0,34	0,06	-0,01	0,31	0,01	0,85	-0,11	0,06	0,24	0,29	0,50	0,39
loc_added		1,00	0,95	0,66	0,72	0,62	0,17	0,24	0,63	0,16	0,36	-0,10	0,01	0,07	0,10	0,34	0,18
max_loc_added			1,00	0,73	0,66	0,63	0,18	0,25	0,64	0,18	0,24	-0,08	0,03	0,03	0,06	0,26	0,13
ave_loc_added				1,00	0,30	0,30	0,13	0,39	0,81	0,37	0,09	-0,15	-0,09	-0,03	0,03	0,13	0,09
loc_deleted					1,00	0,95	0,60	-0,50	0,21	-0,41	0,31	0,02	0,13	0,11	0,10	0,25	0,12
max_loc_deleted						1,00	0,68	-0,56	0,17	-0,49	0,21	0,04	0,14	0,09	0,06	0,18	0,08
ave_loc_deleted							1,00	-0,62	0,06	-0,88	0,03	0,10	0,12	0,03	-0,06	0,02	0,00
codechurn								1,00	0,48	0,77	0,02	-0,14	-0,16	-0,07	-0,01	0,07	0,06
max_codechurn									1,00	0,34	0,27	-0,12	-0,05	0,00	0,05	0,19	0,15
ave_codechurn										1,00	0,02	-0,17	-0,16	-0,05	0,07	0,05	0,04
authors											1,00	-0,13	0,03	0,29	0,34	0,50	0,39
ave_changeset												1,00	0,90	0,09	-0,09	-0,11	-0,09
max_changeset													1,00	0,16	0,04	-0,04	-0,04
age														1,00	0,89	0,22	0,21
weighted_age															1,00	0,24	0,23
bugfixes																1,00	0,34

Tabla 6.5: Matriz de correlación de métricas de cambio

La métrica `bugfixes` es el conteo, para una clase, de cuántos *bugfixes* tuvo en un trimestre. La métrica `bugfixes_prox_periodo` es el conteo de cuántos *bugfixes* se hicieron a una clase en el trimestre siguiente. Se considera *bugfix* a la modificación de la clase por medio de un commit de tipo FR (cfr. 4.5.2). Uno de los objetivos centrales es encontrar un predictor para esta métrica `bugfixes_prox_periodo` (variable dependiente) utilizando las demás.

Se puede observar que las métricas `changeset` (que hablan de la cantidad de clases que cambian juntas en un mismo commit) no tienen correlación significativa con otras métricas. Lo mismo sucede con `age` (antigüedad de la clase en el repositorio). Esto indica que `changeset` y `age` son potenciales variables para construir un modelo predictivo de `bugfixes_prox_semestre`.

Los *bugfixes* del trimestre (métrica `bugfixes`) también presentan baja correlación con la mayoría de las otras métricas, excepto por `authors` y `revisions`. Esto la convierte en otro potencial candidato para formar parte de un modelo siempre que no se utilice junto a estas dos últimas. En la literatura se ha mencionado a los *bugfixes* de un período como candidato a predictor de los *bugfixes* del período siguiente [3].

Por lo tanto, las métricas de cambio factibles de ser utilizadas juntas en un modelo de regresión son `revisions`, `ave_loc_added`, `ave_loc_deleted`, `ave_changeset`, `max_changeset`, `age` y `weighted_age`.

6.3 Modelos de Regresión logística

Para generar estos modelos se agregó una nueva columna booleana `Fixed` al conjunto de datos, que indica si una clase tuvo o no *bugfixes*[53].

El objetivo de estos modelos de regresión logística es predecir si una clase tendrá o no *bugfixes*. Es decir que se trata de estimar la probabilidad de que una clase deba ser corregida en el futuro (variable cualitativa binaria) en función de las variables cuantitativas (métricas). En otras palabras, se trata de un clasificador binario [33].

Es importante tener en cuenta que, aunque la regresión logística permite, en última instancia, clasificar (resultado binario), se trata de un modelo de regresión que modela la probabilidad (entre 0 y 1) de cada clase de pertenecer a uno de los grupos (defectuosa / no-defectuosa). La asignación final se hace en función de las probabilidades predichas.

Por ejemplo, si una clase tiene un 20% de probabilidades de ser defectuosa, ¿cómo se clasifica? ¿como defectuosa o como no-defectuosa? Luego de crear el modelo, se debe establecer un “umbral” de probabilidad a partir del cual se considerará que una clase es defectuosa. A continuación se exploran distintos valores y luego se discuten las implicancias para los resultados y la utilidad de éstos.

[53] En el caso de las métricas estáticas esto es (`Fixed` = `Bugfixes_semestre` > 0) y en el caso de las métricas de cambio es (`Fixed` = `bugfixes_prox_periodo` > 0).

6.3.1 Regresión logística con métricas estáticas

Se utilizarán las métricas del primer semestre para crear el modelo (datos de entrenamiento, 968 casos) y posteriormente se aplicará este modelo para predecir las observaciones del segundo semestre.

En primer lugar se genera un modelo de regresión logística con algunas métricas estáticas no correlacionadas, para determinar si su p-value es significativo.

La salida obtenida en R se muesta en la Figura 6.6.

```
Call:
glm(formula = Fixed ~ LOC + CK_RFC + Rob_12 + Rob_13 + Rob_14 +
    Gul_5 + Gul_6, family = "binomial",
    data = estaticas1sem)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.0832  -0.5146  -0.4120  -0.3290   2.5088

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept) -3.3223423  0.2204069 -15.074  < 2e-16 ***
LOC          0.0008176  0.0002123   3.851 0.000117 ***
CK_RFC       0.0075702  0.0017899   4.230 2.34e-05 ***
Rob_12       0.0021381  0.0075472   0.283 0.776949
Rob_13       0.0975559  0.0658731   1.481 0.138616
Rob_14       0.0752992  0.0152137   4.949 7.44e-07 ***
Gul_5        0.0383111  0.0116674   3.284 0.001025 **
Gul_6        0.0343036  0.0177307   1.935 0.053027 .
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Figura 6.6: Primer modelo de regresión logística con métricas estáticas (datos primer semestre)

Por lo tanto, se construirá un modelo solamente con las métricas `LOC`, `CK_RFC`, `Rob_14` y `Gul_5` ya que éstas presentan un p-value aceptable (menor a 0,05). La salida en R se muestra en la Figura 6.7.

```
Call:
glm(formula = Fixed ~ LOC + CK_RFC + Rob_14 + Gul_5,
    family = "binomial", data = estaticas1sem)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.0889  -0.5270  -0.4191  -0.3338   2.4872

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept) -3.2764055  0.2058406 -15.917  < 2e-16 ***
LOC          0.0008368  0.0002092   4.000 6.33e-05 ***
CK_RFC       0.0082525  0.0017516   4.711 2.46e-06 ***
Rob_14       0.0747189  0.0150297   4.971 6.65e-07 ***
Gul_5        0.0400077  0.0113918   3.512 0.000445 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Figura 6.7: Modelo obtenido a partir de métricas con p-value menor a 0,05 (datos primer semestre)

Si se utiliza un umbral de 0,5 para la clasificación (es decir, determinando una probabilidad mayor a 50% como una clase con defectos), se consigue la siguiente matriz de confusión entre predicción del modelo y observación real:

```
              predicciones
observaciones   0   1
        FALSE 811  12
        TRUE  118  27
```

En términos generales, siguiendo a Moser, se pueden definir los siguientes indicadores [3] a utilizar en el análisis (Tabla 6.6)

		predicción	
		no-defectuosa	defectuosa
observación	no-defectuosa	811 ($n_{1,1}$)	12 ($n_{1,2}$)
	defectuosa	118 ($n_{2,1}$)	27 ($n_{2,2}$)

True Positive Rate (TP), tambien llamado *Recall* o *Sensitivity*	$n_{2,2} / (n_{2,2} + n_{2,1}) * 100\%$
False Positive Rate (FP)	$n_{1,2} / (n_{1,2} + n_{1,1}) * 100\%$
True Negative Rate (TN), también llamado *Specificity*	$n_{1,1} / (n_{1,2} + n_{1,1}) * 100\%$
False Negative Rate (FN)	$n_{2,1} / (n_{2,2} + n_{2,1}) * 100\%$
Porcentaje de Predicción Correcta (PC), también llamado *Accuracy*	$(n_{1,1} + n_{2,2}) / (n_{1,1} + n_{1,2} + n_{2,1} + n_{2,2}) * 100\%$

Tabla 6.6: Indicadores a utilizar para el análisis de los modelos de regresión

Debido a que los datos se encuentran desbalanceados, es decir, se tienen muchas más clases sin defectos que defectuosas, se calculará además un indicador conocido como *Balanced Accuracy* (Precisión Balanceada) [34].

Balanced Accuracy (BA)	(TP + TN) / 2 *ó lo que es lo mismo:* (Sensitivity + Specificity) / 2

La matriz de confusión generada con este umbral de 0,5 puede graficarse como se muestra en la Figura 6.8.

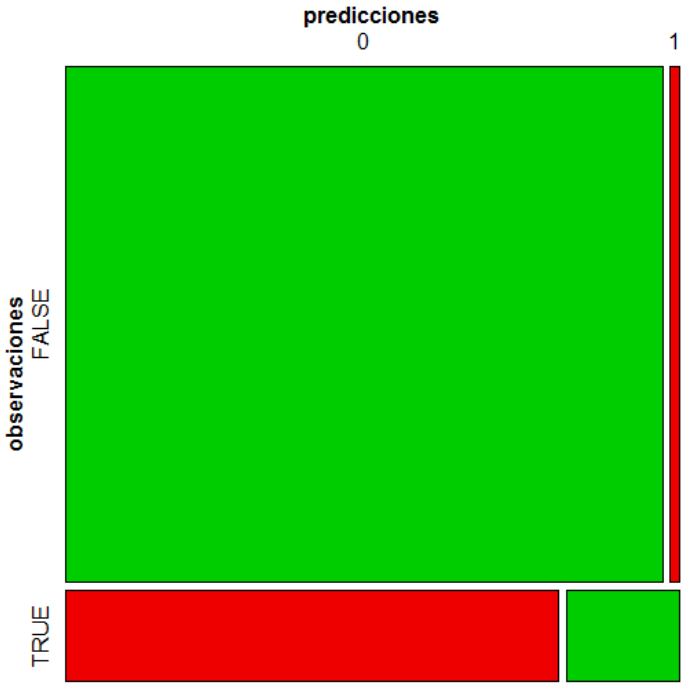


Figura 6.8: Gráfica de la matriz de confusión obtenida a partir del primer modelo

En esta gráfica de la matriz de confusión se observan los verdaderos (negativo y positivo) en la diagonal principal ($n_{1,1}$ y $n_{2,2}$).

Se puede decir que este modelo generado es bueno para los casos no defectuosos, clasificando correctamente el 99% (TN) de las clases que no tendrán *bugfixes* (811 de 823). Pero en el caso de las clases que sí serán corregidas sólo detecta a 27 de 145 (TP de 19%). Siguiendo a Moser [3] se puede considerar que es más costoso corregir posteriormente en el ciclo de vida un defecto no detectado (*false negative*) que inspeccionar una clase clasificada como defectuosa cuando realmente no lo es (*false positive*)[54]. En otras palabras, es sabido en Ingeniería de Software que los falsos negativos son mucho más costosos que los falsos positivos. Para los datos precedentes, el FN es de 118/145 = 81%.

El porcentaje de predicciones correctas (PC) es en este caso 87%, lo que surge de calcular la razón entre ambos positivos y la totalidad de casos, esto es: (811+27) / 968.

La precisión balanceada (BA) señala una realidad menos esperanzadora para el modelo que el PC, ya que es el promedio entre el TP y el TN, lo que resulta en 59%. Este indicador baja respecto del PC porque si bien el modelo es bueno para las clases no-defectuosas, no lo es para las otras, hecho claramente visible en su muy bajo TP (19%).

A continuación se modifica el umbral de probabilidad a partir del cual se hace el corte de la predicción, por tanteo, y se arriba a distintos resultados correspondientes a las matrices de confusión de la Figura 6.9.

[54] Cfr. 2.2.2

```
             predicciones   # umbral 0.4
observaciones   0   1
        FALSE 804  19
        TRUE  107  38

             predicciones   # umbral 0.3
observaciones   0   1
        FALSE 777  46
        TRUE   94  51

             predicciones   # umbral 0.2
observaciones   0   1
        FALSE 719 104
        TRUE   75  70

             predicciones   # umbral 0.1
observaciones   0   1
        FALSE 451 372
        TRUE   32 113
```

Figura 6.9: Matrices de confusión obtenidas con distintos umbrales (datos primer semestre)

Se resumen estos hallazgos por medio de los indicadores descriptos en la Tabla 6.7.

Métricas estáticas: Datos de entrenamiento (primer semestre) **Fixed ~ LOC + CK_RFC + Rob_14 + Gul_5**						
Umbral	TP	FP	TN	FN	PC	BA
0,5	19	1	99	81	87	59
0,4	26	2	98	74	87	62
0,3	35	6	94	65	86	64,5
0,2	48	13	87	52	82	67,5
0,15	57	23	77	43	74	67
0,1	78	45	55	22	58	66,5
Valores en porcentajes (%)						

Tabla 6.7: Indicadores obtenidos para el modelo de regresión logística con distintos umbrales

Como ya lo observa Moser [3], a medida que se modifica el modelo para reducir los falsos negativos (que resultan muy costosos) también se incrementa el esfuerzo de inspección de clases que realmente no tienen defectos, es decir, se obtendrán más falsos positivos. Esto es porque la disminución en el umbral traslada más casos “hacia la derecha” de la matriz de

confusión (porque naturalmente clasifica más elementos como defectuosos), donde se encuentran el falso positivo y el verdadero positivo. Con otras palabras, al enviar casos hacia la columna derecha de la matriz se está reduciendo el falso negativo (en la columna izquierda de la matriz); hecho que ya se ha descripto como deseable.

Se aplica ahora el modelo a los datos del segundo semestre (1000 casos), para evaluar su potencia predictiva con datos nuevos (Figura 6.10).

```
predicciones.newdata = predict(prueba, newdata = estaticas2sem, type="response")
predicciones.newdata.pred = rep(0, dim(estaticas2sem)[1])
predicciones.newdata.pred[predicciones.newdata > .5] = 1  #aqui elegimos el umbral
matriz_confusion <- table(estaticas2sem[["Fixed"]], predicciones.newdata.pred)

             predicciones   # umbral 0.3
observaciones   0   1
        FALSE 828  38
        TRUE  105  29

             predicciones   # umbral 0.2
observaciones   0   1
        FALSE 782  84
        TRUE   81  53

             predicciones   # umbral 0.15
observaciones   0   1
        FALSE 710 156
        TRUE   60  74

             predicciones   # umbral 0.1
observaciones   0   1
        FALSE 518 348
        TRUE   39  95
```

Figura 6.10: Matrices de confusión en diferentes umbrales, obtenidas para datos del segundo semestre con el modelo entrenado previamente

Se puede resumir el comportamiento del modelo sobre datos nuevos con la Tabla 6.8.

Métricas estáticas: Datos nuevos (segundo semestre) **Fixed ~ LOC + CK_RFC + Rob_14 + Gul_5**						
Umbral	TP	FP	TN	FN	PC	BA
0,3	22	4	96	78	86	59
0,2	40	10	90	60	84	65
0,15	55	18	82	45	78	68,5
0,1	71	40	60	29	61	65,5
Valores en porcentajes (%)						

Tabla 6.8: Indicadores obtenidos aplicando el modelo a datos del segundo semestre

El desempeño del modelo con datos nuevos (segundo semestre) es prácticamente igual a los datos de entrenamiento (primer semestre). Podemos concluir que el modelo entrenado con métricas estáticas puede aplicarse a nuevos datos manteniendo su consistencia siendo el umbral de 0,15 el más óptimo con una precisión balanceada de 68,5%, TP de 55% y FN de 45%.

6.3.2 Regresión logística con métricas de cambio

Se construirá un modelo de regresión logística con las métricas de cambio y se procederá a evaluar su desempeño de manera similar a como se hizo con las métricas estáticas. Se tomarán los datos del primer trimestre como entrenamiento, y se realizará la predicción para los trimestres segundo y tercero, en función de evaluar su poder predictivo.

Un primer modelo con métricas no correlacionadas se presenta en la Figura 6.11.

```
Call:
glm(formula = fixed ~ revisions + ave_loc_added + ave_loc_deleted +
    ave_changeset + max_changeset + age + weighted_age, family = "binomial",
    data = cambio1trim)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.2230  -0.6076  -0.4673  -0.2967   2.4312

Coefficients:
                Estimate Std. Error z value Pr(>|z|)
(Intercept)     -2.694111   0.492646  -5.469 4.53e-08 ***
revisions        0.252419   0.078622   3.211  0.00132 **
ave_loc_added    0.018861   0.008699   2.168  0.03015 *
ave_loc_deleted -0.008317   0.012710  -0.654  0.51284
ave_changeset   -0.070550   0.084923  -0.831  0.40612
max_changeset    0.050827   0.054232   0.937  0.34864
age             -0.031361   0.036841  -0.851  0.39462
weighted_age     0.033906   0.036844   0.920  0.35745
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Figura 6.11: Primer modelo de regresión logística con métricas de cambio (datos primer trimestre)

Las variables con p-value aceptable son: `revisions` y `ave_loc_added`, por lo que se utilizarán sólo estas dos para la construcción de un primer modelo.

A continuación se obtienen las matrices de confusión para los distintos umbrales de probabilidad, de la misma forma que se ha procedido para las métricas estáticas. Los resultados de aplicar el modelo sobre los datos del primer trimestre (entrenamiento, 307 casos) se resumen en la Tabla 6.9.

Métricas de cambio: Datos de entrenamiento (primer trimestre) **fixed ~ revisions + ave_loc_added**						
Umbral	TP	FP	TN	FN	PC	BA
0,3	36	6	94	64	83	65
0,2	55	16	84	45	79	69,5
0,15	69	29	71	31	71	70
0,1	100	93	7	0	24	53,5
Valores en porcentajes (%)						

Tabla 6.9: Indicadores obtenidos para el modelo de regresión logística con distintos umbrales

Utilizando este modelo construido a partir de datos de entrenamiento del primer trimestre, se pueden hallar las predicciones para el segundo y tercer trimestre (Tabla 6.10).

Métricas de cambio: Datos nuevos **fixed ~ revisions + ave_loc_added**							
Modelo	**Umbral**	**TP**	**FP**	**TN**	**FN**	**PC**	**BA**
Predicciones segundo trimestre (335 casos)	0,3	29	5	95	71	81	62
	0,2	43	18	82	57	74	62,5
	0,15	57	31	69	43	66	63
	0,1	99	95	5	1	24	52
Predicciones tercer trimestre (418 casos)	0,3	42	8	92	58	86	67
	0,2	55	18	82	45	79	68,5
	0,15	75	31	69	25	70	72
	0,1	94	89	11	6	22	52,5
Valores en porcentajes (%)							

Tabla 6.10: Indicadores obtenidos aplicando el modelo a datos del segundo y tercer semestre

Al igual que las métricas estáticas, con el modelo generado con métricas de cambio se verifica un comportamiento consistente respecto de los datos de entrenamiento cuando se utiliza el modelo con datos nuevos.

Probando distintas combinaciones de métricas no correlacionadas se pueden conformar distintos modelos con diversas variables predictoras de p-value menor a 0,05.

Estos otros modelos posibles se encuentran resumidos en la Tabla 6.11.

Métricas de cambio: Datos de entrenamiento							
Modelo	**Umbral**	**TP**	**FP**	**TN**	**FN**	**PC**	**BA**
A) fixed ~ revisions + ave_loc_added	0,2	55	16	84	45	79	69,5
	0,175	66	25	75	34	73	70,5
	0,15	69	29	71	31	71	70
B) fixed ~ codechurn + bugfixes + age	0,2	59	22	78	41	75	68,5
	0,175	67	31	69	33	69	68
	0,15	74	36	64	26	66	69
C) fixed ~ authors + codechurn	0,2	52	15	85	48	79	68,5
	0,175	60	23	77	40	74	68,5
	0,15	62	30	70	38	68	66
D) fixed ~ bugfixes + weighted_age	0,2	55	22	78	45	74	66,5
	0,175	59	27	73	41	70	66
	0,15	76	41	59	24	63	67,5
Valores en porcentajes (%)							

Tabla 6.11: Indicadores obtenidos con modelos utilizando distintas combinaciones de métricas como variables predictoras (datos primer trimestre)

Se puede decir que, de los modelos obtenidos para esta comparación entre modelos con métricas de cambio como variables independientes (Tabla 6.11), la mejor opción es la A (`revisions` + `ave_loc_added`) utilizando un umbral de 0,175. Este modelo presenta una precisión balanceada (BA) de 70,5%. Manteniendo una predicción correcta (PC) del 73% encontramos que los falsos negativos son relativamente bajos (34%) con un TP (*recall*) de 66%.

Sin embargo, también puede observarse que los modelos tienen un desempeño muy similar, lo que verifica que las métricas de cambio están describiendo un mismo fenómeno: el trabajo de los desarrolladores sobre el código fuente del producto. En este sentido, deberá analizarse el costo de obtener cada métrica, ya que si bien se trata de métricas de proceso naturalmente más difíciles de obtener que las métricas estáticas del código fuente, algunas son, sin embargo, más laboriosas que otras. Por ejemplo, `revisions` es de muy sencilla obtención ya que surge del mismo sistema de control de versiones, pero sólo será útil si cada commit ha sido catalogado correctamente por su autor, hecho que enfatiza la necesidad de estandarizar los mensajes de commit y los procedimientos de la organización.

En la Tabla 6.12 se caracterizan algunos aspectos del costo de estas métricas.

Métrica	Descripción	Cómo se obtiene	Costo
revisions	Cantidad de veces que una clase fue modificada con un commit FI	Disponible directamente en el sistema de control de versión del proyecto	Muy alto si no está discriminado el tipo de commit en el workflow del desarrollador; bajo en caso contrario
ave_loc_added	Cantidad de líneas que se agregan, en promedio, con cada cambio	Del historial de revisiones, con conteos una por una	Alto
codechurn	Cantidad de líneas agregadas menos las borradas, en un período de tiempo	Del historial de revisiones, con conteos una por una	Alto
bugfixes	Cantidad de veces que una clase fue modificada para corregir un defecto, es decir, con un commit FR	Similar a revisions, pero con la dificultad de que se debe discriminar qué cambio se realiza por un bugfix de los demás cambios posibles	Muy alto si no está discriminado el tipo de commit en el workflow del desarrollador; bajo en caso contrario
age	Edad en semanas de la clase	Disponible directamente en el sistema de control de versión del proyecto	Medio a bajo, dependiendo de si los archivos tienen cambios de nombre o de directorio frecuentes
authors	Cantidad de desarrolladores que modificaron una clase en un período de tiempo	Disponible directamente en el sistema de control de versión del proyecto	Bajo
weighted_age	Cociente que involucra las líneas agregadas en un período de tiempo y la edad de la clase al momento de ese agregado (cfr. 4.6.1.3)	Debe hacerse el cálculo tomando cada revisión a la clase y calculando el age respectivo.	Muy alto

Tabla 6.12: Costo de obtención de las métricas de cambio utilizadas en los modelos de regresión

Se puede decir entonces que el mejor modelo con métricas de cambio es `revisions + ave_loc_added` debido a su buena performance como clasificador y a su menor costo comparado con los demás, siempre que la política de commit al repositorio garantice contar con los datos necesarios para la clasificación propuesta (es decir, poder discriminar trivialmente los cambios de tipo FI y FR). Esta política puede ser, simplemente, indicar en el mensaje de commit el tipo de cambio que se está realizando (es decir FI, FR o GM) de manera que sea fácilmente distinguible con una expresión regular.

6.3.3 Comparación entre modelos estáticos vs. de cambio

Considerando los modelos de regresión logística construidos anteriormente con las métricas estáticas se pueden comparar con aquellos con métricas de cambio, a los efectos de contrastarlos.

En la Tabla 6.13 se comparan los mejores modelos de ambos juegos de métricas.

Modelo	Datos de entrenamiento	Datos a predecir	Umbral	TP	FP	TN	FN	PC	BA
Estáticas **Fixed ~ LOC + CK_RFC + Rob_14 + Gul_5**	Métricas 1er. semestre	Bugfixes durante el 2do. semestre con métricas del 2do. semestre	0,15	55	18	82	45	78	68,5
De cambio **fixed ~ revisions + ave_loc_added**	Métricas 1er. trimestre	Bugfixes durante el 4to. trimestre con datos del 3er. trimestre	0,15	75	31	69	25	70	72
Valores en porcentajes (%)									

Tabla 6.13: Indicadores de los mejores modelos encontrados (métricas estáticas vs. de cambio)

Se puede concluir que el modelo de regresión logística con métricas de cambio es mejor que el construido con métricas estáticas, debido a un considerable mayor *recall* (75 vs. 55) y un porcentaje de FN mucho menor (25 vs. 45). Esta mejor performance afecta en la precisión (BA; 72 vs. 68,5) otorgando una ventaja al modelo con métricas de cambio.

Debe recalcarse nuevamente que los falsos negativos representan mayores costos a largo plazo dado que son errores no detectados. Los falsos positivos, por otra parte, representan costos de sobreinspección por tratarse de defectos falsamente detectados, pero que son preferibles a los anteriores. Se propone que este aspecto debe tenerse siempre en cuenta a la hora de contrastar dos modelos, a los efectos de elegir cuál costo se prefiere asumir: el defecto detectado tarde o la sobreinspección temprana.

No puede soslayarse en el análisis, nuevamente, la dificultad que representa recolectar las métricas de cambio desde el sistema de control de versiones, a diferencia de la sencillez con la que se obtienen las métricas estáticas una vez que se cuenta con una herramienta que las recolecte de manera automática.

6.4 Amenazas a la validez

6.4.1 Métricas estáticas

Por la manera en que el lenguaje permite hacer llamadas a método de manera indirecta, es posible haber subestimado algún conteo de acoplamiento, esto es, `CK_CBO` y `CK_RFC`. La Figura 6.12 muestra una porción de código correspondiente al método `performAction` de la clase `AjaxDispatcher`.

```
/*  ...  */

try {
       $result = call_user_func_array( $this->func_name, $this->args );

       if ( $result === false || $result === null ) {

/*  ...  */
```

Figura 6.12: Fragmento de código de la clase AjaxDispatcher

En esta porción de código puede verse una llamada a método realizada con la funcionalidad `call_user_func_array` del lenguaje, que permite realizar una llamada a partir de un `string`. Este tipo de llamadas puede generar una subestimación del acoplamiento de esta clase con otras, al no ser posible detectar cuál es el método que, en efecto, es llamado al momento de la ejecución y perdiendo la oportunidad de incrementar el conteo de acoplamiento. Si bien esta posibilidad del lenguaje no es utilizada extensivamente en el caso de estudio bajo análisis, podría ser una amenaza importante en otros proyectos donde esto sí suceda en mayor medida.

En el siguiente ejemplo de código, perteneciente a la clase `ChangesFeed` puede observarse una instanciación a partir de una variable. En este caso también es imposible deducir cuál es el tipo del objeto realmente instanciado.

```
return new $wgFeedClasses[$this->format]( /* ... */  );
```

En términos de la sintaxis del lenguaje, `wgFeedClasses` es un vector de `strings`, al cual se accede a través de un subíndice[55] que se obtiene a su vez de una variable de instancia (llamada `format`). Con el `string` obtenido de este vector se realiza la instanciación de un objeto, utilizando el valor del `string` como el nombre de la clase a instanciar (!).

[55] En sentido estricto, dentro de los corchetes no aparece un subíndice porque la estructura de datos utilizada aquí no es un vector sino un Diccionario, por lo que el término correcto sería clave o *key*. Se

En líneas generales, la forma de calcular el acoplamiento está restringida a los lugares en donde fehacientemente se pueden deducir los tipos con análisis estático en PHP. Esto amenaza la precisión de las métricas que miden acoplamiento debido a las características mismas del sistema de tipos del lenguaje.

En las métricas `Gul_1`, `Gul_2`, `Gul_4` y `Gul_5` se trabaja con el bloque de documentación que puede acompañar cada método, con el objeto de determinar si los tipos están o no documentados. Esto genera que la métrica dependa directamente del énfasis que los desarrolladores pongan para seguir el estilo de documentación y ser consistentes.

Por último, puede existir un sesgo en la medición de `LOC` debido al uso que los desarrolladores hacen de los comentarios. Al calcular `LOC` deben omitirse las líneas en blanco y las líneas comentadas para proceder a elaborar esta métrica. Las posibilidades sintácticas y las distintas combinaciones hacen que sea difícil asegurar que todas ellas son detectadas exhaustivamente por las expresiones regulares utilizadas. Sin embargo, se estima que las desviaciones son despreciables en el código fuente analizado, aunque puede no serlo en otros proyectos.

6.4.2 Métricas de cambio

Dentro de las tareas realizadas para obtener las métricas de cambio se incluye la clasificación manual de 2900 commits (véase 4.5.2) en tres categorías: *Feature Introduction* (FI), *Fault Repairing* (FR) y *General Maintenance* (GM). Esta clasificación, debido a la carga cognitiva que implica, puede incluir naturalmente errores humanos.

La clasificación es compleja cognitivamente porque, por lo general, el mensaje (escrito por el autor) que acompaña al commit no permite saber a ciencia cierta cuál es la intención del cambio que está siendo ejecutado. Por ejemplo, muchos commits contienen la palabra “fix” en su mensaje de commit pero al evaluar pormenorizadamente los cambios que se están realizando en las clases se puede determinar que no son realmente correcciones de defectos (FR) sino agregados de nueva funcionalidad (FI). Esta es la razón por la que se ha procedido a clasificar los commits manualmente aunque resultara una tarea muy costosa.

La clasificación de algunos commits puede ser objeto de discusión, ya que la diferenciación entre FR, FI y GM puede tornarse sumamente sutil. Esto hace que la clasificación realizada por distintos operadores humanos pueda no ser consistente. Por ejemplo, un commit[56] realiza un cambio en la forma en que los resultados de búsqueda se muestran en pantalla. El objetivo es que no se vean resultados repetidos similares. ¿Es esto la corrección de un defecto (FR) o una mejora en la usabilidad (FI)? Se ha considerado este caso como una mejora, por lo tanto fue clasificado como FI. Pero debe asumirse que algunas veces[57] la mejora en la usabilidad es bastante similar en efecto final a una corrección de defecto. Tomar esta decisión

explica el sentido de la sentencia utilizando términos del lenguaje C clásico para facilitar la comprensión de la situación sin entrar en los detalles de la implementación.

[56] Se trata del commit identificado como 2ec9915816577dbd749625d083e4ee590bf49f6f

[57] Casos similares: 3b9721d0550516aaa62a4d53cac6fc96734daf75 y 5073379d35df694574cd5be8201f8627bb2329a9.

puede ser una tarea muy costosa desde el punto de vista del tiempo invertido en explorar los cambios al código fuente que se están introduciendo y deducir de éstos una intención que permita justificar la clasificación.

En el presente trabajo la clasificación fue realizada por un mismo operador, de manera que los errores de clasificación son despreciables en cuanto los criterios son aplicados de manera consistente. Sin embargo, esta situación puede ser una amenaza significativa si la clasificación se llevara a cabo por personas distintas, razón por la cual se sugiere que los mensajes de commit incluyan la clasificación explícitamente indicada por el desarrollador.

También se han encontrado commits[58] que revierten, por ejemplo, la introducción de alguna característica en la interfaz de usuario del sistema que, con posterioridad, la comunidad ha evaluado como mala idea. Estos cambios han sido clasificados como FI, ya que se consideran una evolución de la aplicación y no la corrección de un defecto (cfr. definición de defecto en Apartado 2.3.1).

Otra amenaza surge de que algunos commits presentan intenciones mixtas, de manera que realizan simultáneamente cambios en varias clases, y estos cambios resultan ser FI y FR. En estos pocos casos se ha decidido cuál era el mayor impacto en términos de volumen de cambio en el sistema y se ha clasificado el commit dentro de esa categoría, perdiendo la posibilidad de incrementar el conteo en la otra y produciendo un error en la métrica. En términos de procedimiento, y para la utilización del conjunto de métricas de este trabajo, este tipo de commit de "intención múltiple" debería ser separado en varios al momento de enviarse al repositorio de versiones.

6.5 Discusión de los resultados

A continuación, se discuten los hallazgos de este trabajo.

- El framework Robiolo [2] puede ser aplicado al lenguaje estudiado.

Teniendo en cuenta los detalles sintácticos del lenguaje PHP, es posible aplicar con buenos resultados el framework de métricas definido originalmente por la autora para el lenguaje Java (cfr. 4.2). La métrica `Rob_14` forma parte del mejor modelo de regresión encontrado con métricas estáticas (cfr. 6.3.1).

- No se han podido construir modelos significativos con regresión lineal.

En el Anexo B se encuentra el estudio de las métricas a través de regresión lineal. El intento de construir un modelo predictor de defectos no ha sido satisfactorio puesto que todos los modelos muestran un R^2 muy bajo. Se puede concluir que para los datos del presente trabajo

[58] Dos ejemplos: 5ad871239d4e28e16ee737f8467b8e305f62eccd y a94f11d625ea70f253e0d9620a44b447dd830b6b.

no es posible utilizar el método de regresión lineal. Se intuye que esto puede deberse a la madurez del producto.

- Los modelos de regresión logística definidos con métricas de cambio son los que lograron mejores resultados en la predicción.

Las métricas de cambio `revisions` y `ave_loc_added` constituyen un modelo[59] que predice una clase defectuosa con un TP (*recall*) del 75%, falso negativo del 25% y precisión de 72% (cfr. 6.3.2). Esto significa, en breve, que la cantidad de modificaciones hechas a una clase con el propósito de realizar *Feature Introduction* es el mejor predictor de defectos futuros de dicha clase.

- Los modelos de regresión logística con métricas estáticas constituyen buenos predictores, aunque por debajo de las métricas de cambio.

Con la combinación de métricas estáticas `LOC` + `CK_RFC` + `Rob_14` + `Gul_5` se puede construir un modelo[60] que predice defectos con un TP del 55%, FN 45% y precisión de 69%.

- Las métricas de cambio son más costosas de obtener.

Si bien constituyen mejores predictores, también es cierto que las métricas de cambio son más laboriosas de computar. Es más complejo trabajar con esta información de cambios en el repositorio de versiones que con métricas estáticas obtenidas con una versión puntual del código fuente a partir, simplemente, de los archivos. Para calcular las métricas de cambio se debieron clasificar manualmente 2800 commits ya que no se encontró forma automática confiable de dividir estos commits en "*Feature Introduction*" o "*Fault Repairing*". Además se debió construir una base de datos relacional de cambios para procesar la información y poder calcular las métricas de cambio (cfr. 4.5.2).

- Con el objeto de calcular las métricas de cambio se sugiere que los equipos de desarrollo indiquen de manera consistente en el mensaje de commit si el cambio que está siendo introducido se clasifica en FI, FR o GM (cfr. 4.5.2).

Esta incorporación al estilo de los mensajes de commit reduciría el costo de obtener las métricas de cambio. En este mismo sentido, los desarrolladores se verían obligados a organizar los commits para que no se produzcan juntos cambios que deben ser clasificados en tipos diferentes. Esto se puede instaurar como una buena práctica.

Esta buena práctica puede implementarse indicando de manera consistente en el mensaje de commit el tipo de cambio que se está realizando (es decir FI, FR o GM) de manera que sea trivialmente distinguible con una expresión regular en el futuro.

[59] Fixed ~ revisions + ave_loc_added (cfr. 6.3.2)
[60] Fixed ~ LOC + CK_RFC + Rob_14 + Gul_5 (cfr. 6.3.1)

- Comparando con los resultados obtenidos por Moser [3] (para modelos de regresión logística con métricas de cambio) se ha obtenido una precisión similar, pero con diferencias en TP y FN.

Autor	TP	FP	TN	FN	PC
Moser (promedio)	38	6	94	62	78
Este trabajo	75	31	69	25	70

En promedio, con el mismo método de regresión y las mismas métricas de cambio, los resultados de Moser[61] arrojan un TP de 38%, FP de 6% y una precisión de 78%. Estas diferencias con los modelos obtenidos en el presente trabajo están relacionados con la elección del umbral, como se explicó en el Apartado 6.3. Llama la atención que Moser presente estos resultados, teniendo en cuenta la importancia que le otorga a los FN (cfr. 2.2.2). Observando la matriz de confusión (cfr. 6.3.1) puede verse que el TP y el FN comparten la misma fila, por lo que debido a la fórmula de cálculo, puede deducirse que FN = 100 - TP. Esto quiere decir que el modelo presentado por Moser tiene un FN muy alto[62].

- Los modelos entrenados con datos de un trimestre se aplican consistentemente a datos nuevos (de trimestres posteriores).

En 6.3 se demuestra que los modelos entrenados sirven para realizar predicciones satisfactorias con datos nuevos.

- Existe una correlación positiva entre la métrica estática `Gul_5` y la probabilidad de que una clase sea defectuosa.

En 6.3.1 se demuestra que el coeficiente obtenido para la métrica `Gul_5` en el modelo con métricas estáticas es positivo (0,04). Esto quiere decir que existe una relación directa entre esta métrica, propuesta en este trabajo, y la probabilidad de que una clase sea defectuosa. La métrica `Gul_5` cuantifica la cantidad de parámetros sin tipo declarado pero que sí cuentan con documentación del tipo esperado, en un bloque de comentario que acompaña la declaración del método (cfr. 4.2). No se ha podido encontrar una razón para este fenómeno.

[61] Página 186 del trabajo original [3], publicado en ICSE, mayo de 2008.
[62] Por la misma razón, también es cierto que el TN es muy alto (algo deseable) ya que el FP reportado está en promedio en el 6%. Esto surge de: TN = 100 - FP. Esto quiere decir que el umbral está alto, clasificando más archivos como no-defectuosos.

- Mediante el análisis de las métricas Gullino, propuestas en este trabajo, se observa una progresiva adopción por parte de los desarrolladores de las posibilidades sintácticas de declarar los tipos para los parámetros de los métodos, lo que aumenta la posibilidad de detectar errores de tipos.

La cantidad de clases que presentan un `Gul_6` (cantidad de parámetros con tipo declarado) mayor a cero es de 298 en el primer semestre (lo que representa un 31% de un total de 968 clases) y de 332 en el segundo semestre (un 33% sobre un total de 1000 clases). Esto indica que, porcentualmente, cada vez más parámetros tienen declarado su tipo.

- Con las métricas `Gul_1` y `Gul_2` puede determinarse que los tipos de retorno de los métodos han mejorado levemente en su documentación. La métrica `Gul_3` (cantidad de métodos con tipo de retorno declarado) es en todos los casos cero, debido a que la posibilidad de declarar el tipo de retorno no se había introducido aún en el lenguaje en el año 2014.

Las siguientes consideraciones tienen en cuenta solamente las clases que tienen por lo menos un método.

La métrica `Gul_1` (cantidad de métodos que tiene una clase en donde no se han documentado ni declarado el tipo de retorno) se comporta de esta manera: en el primer semestre, la cantidad de clases con `Gul_1` igual a cero es 144 (15% de 968 clases) mientras que en el segundo semestre es 165 (17% de 1000 clases). Este incremento de la cantidad de “ceros” de la métrica indica que las clases que tienen métodos sin tipo declarado ni documentado presenta un leve descenso. Con otras palabras, es preferible que la cantidad de clases que tienen `Gul_1` igual a cero sea cada vez mayor, ya que esta métrica evalua los métodos con retorno no documentado, y esto no es deseable. Si se consideran las clases que tienen al menos un método, un cero en esta métrica indica que no existe ningún método con retorno sin documentar.

De manera consistente, la métrica `Gul_2` (cantidad de métodos con tipo no declarado pero sí documentado) muestra este comportamiento: en el primer semestre la cantidad de clases con `Gul_2` igual a cero es 256 (26% de 968 clases) mientras que en el segundo semestre es 250 (25% de 1000 clases). Este pequeño decremento de la cantidad de “ceros” también indica que los retornos han mejorado en la documentación. Si se consideran las clases que tienen al menos un método, un cero en esta métrica indica que no existe ningún método con retorno documentado, por lo que su decrecimiento es deseable.

No obstante estos hallazgos, tanto el valor porcentual de cambio como el intervalo de tiempo analizado son pequeños, por lo que su significancia deberá seguir siendo analizada en el futuro con más datos.

6.6 Trabajos relacionados

Diversos autores han trabajado en la predicción de defectos durante décadas. Los últimos trabajos del área utilizan métricas estáticas y, cada vez más frecuentemente, métricas de cambio para construir modelos con diversas técnicas de estimación. A continuación se resumen los hallazgos más relevantes de varios autores.

Gyimothy et al. [6] analizaron la suite C&K de métricas orientadas a objetos para la predicción de defectos. Como caso de estudio trabajaron con el código fuente del cliente de correo Mozilla, desarrollado en C++. Aplicando distintos métodos de estimación (regresión lineal, logística y árboles de decisión) logran una precisión menor a 70%, hallando en `CBO` y `LOC` los mejores predictores.

Basili [5] comparó la suite C&K con otras métricas estáticas del código fuente[63], encontrando que aquellas son mejores predictores (excepto `LCOM`). Utilizó proyectos universitarios desarrollados en C++ y aplicó regresión logística. Logra una precisión de 79% y TP de 83%.

Moser [3] continúa el trabajo de Zimmermann [35] sobre la predicción de los defectos del proyecto Eclipse (lenguaje Java) pero introduciendo una suite de métricas de cambio[64] y aplicando varias técnicas de estimación (regresión logística, Naive Bayes, árboles de decisión). Al igual que la presente investigación, encuentra que las métricas de cambio son mejores predictores que las métricas estáticas del código fuente, logrando modelos con TP de 80%.

Kamei [11], Madeyski [14], Caglayan [9] y Zhang [15] arriban a resultados similares a Moser, utilizando casos de estudio comparables, con varios métodos de estimación.

Giger [13] trabajó a nivel de método y también confirmó que las métricas de cambio son mejores predictores que las métricas estáticas. Logra una precisión superior al 80% con varias técnicas de *machine learning* (Random Forest, Bayesian Network, árboles de decisión). El caso de estudio se desarrolla sobre proyectos open source en lenguaje Java.

Hassan [4] introduce el concepto de entropía de cambios, es decir, el análisis de la distribución de los cambios en el código fuente con el paso del tiempo. Propone considerar los cambios en el código fuente de acuerdo a la intención del desarrollador[65]. Trabajando con proyectos en lenguaje C y C++, y por medio de regresión lineal, halla que la entropía es igual o mejor predictor que la cantidad de fallas anteriores en un artefacto.

D'Ambros [10] utilizó varios conjuntos de métricas con la intención de contrastarlos en un estudio de gran extensión sobre la temática de predicción de defectos. Como caso de estudio utiliza proyectos de código abierto desarrollados en Java. Halla que la mejor predicción se logra con la entropía de los cambios definida por Hassan. Sin embargo, esta información es muy costosa de obtener desde el repositorio de código fuente. En segundo lugar aparecen algunas métricas de cambio y luego las métricas C&K.

Nam et al. [17] ha estudiado recientemente la posibilidad de crear modelos de predicción de defectos en proyectos nuevos utilizando datos de otros proyectos (técnica llamada *Cross-*

[63] Se ha descripto este trabajo en 2.3.2.1
[64] Estas métricas se presentan extensamente en 4.3
[65] Se describe esta clasificación en 4.5.2

project defect prediction). Esto permite superar la falta de información en las fases iniciales del desarrollo. Describen una técnica para seleccionar métricas de distintos proyectos, relacionarlas y construir un modelo con este conjunto heterogéneo de métricas. Se necesita mucha más investigación en esta línea.

Recientemente Kondo [36] ha explorado el modo en que se pueden seleccionar o combinar las métricas con las que se construyen modelos (técnicas de *feature selection* y *feature reduction*). La necesidad de reducir la cantidad de métricas aparece cuando la cantidad de información (muestra) es pequeña, algo que sucede en proyectos nuevos y atenta contra los resultados que pueden proveer los modelos. Resultan particularmente prometedores los modelos de la familia de *machine learning* de tipo no-supervisado, por ejemplo las redes neuronales, ya que no precisan de datos de entrenamiento y, por lo tanto, siempre se trabaja con la información propia del proyecto aunque ésta sea escasa. A futuro se deberán probar las técnicas de *feature reduction* basadas en redes neuronales con otros repositorios, a los efectos de validar estos resultados en otros contextos.

En la rama de la predición JIT (*just-in-time*), recientemente Huang [37] comparó modelos supervisados versus no-supervisados, utilizando métricas de cambio y métricas que miden el grado de experiencia de los desarrolladores. Reportan que distintos modelos muestran mejor desempeño en distintos indicadores[66], por lo que recomiendan ampliar el criterio de comparación y evaluación, y sugieren que distintos modelos pueden ser más útiles que otros según el caso particular del equipo de desarrollo. Notablemente, dentro de los seis proyectos que utilizan para su caso de estudio dos se encuentran desarrollados en lenguajes de tipado dinámico (Perl y Ruby), aunque la participación en la muestra es de solo 11%[67]. La predicción JIT se enfoca en predecir si un cambio puede ser inductor de defectos o no, en lugar de identificar archivos (u otro tipo de entidad) existentes que ya sean defectuosos. Este tipo de aproximación al problema de la predicción de defectos fue propuesta por Kamei [38]. Las técnicas JIT más actuales (por ejemplo, el modelo EALR) pueden identificar un 35% de los cambios que inducen defectos inspeccionando solo un 20% de las líneas de código modificadas por todos los cambios anteriores. Estos trabajos representan una línea de investigación distinta a la del presente.

Radjenovic [22] realizó una importante SLR (*Systematic Literature Review*) en 2013, hallando que las métricas estáticas orientadas a objetos y las métricas de cambio son consistentemente reportadas como mejores predictores de defectos en comparación con métricas de tamaño y complejidad tradicionales (cfr. 2.1).

En dicho trabajo se destaca la necesidad de mayor investigación orientada a la industria, con el objetivo de encontrar métricas relevantes para la práctica profesional[68]. Se observa que en el trabajo académico predomina el estudio de métricas estáticas, cuando las métricas de

[66] Por ejemplo *recall*, *precision*, *F1-score*, etc

[67] Representan 25051 cambios al código fuente de un total de 227417 estudiados.

[68] *"Researchers from the industry used mostly process metrics, whereas researchers from academia preferred OO metrics. This may indicate that process metrics are more appropriate for industrial use than static code metrics and that process metrics are what the industry wants. Therefore, we would like to make an appeal to researchers to consider including process metrics in their future studies."* ([22], Conclusions)

cambio han demostrado ser más adecuadas (cfr. 2.3.3). Se reporta [22] una proporción 70/30 de estudios que utilizan métricas estáticas vs. métricas de cambio.

Radjenovic también observa que si se consideran los lenguajes estudiados, el grupo C/C++/Java abarca el 85% de las investigaciones, no existiendo prácticamente investigación[69] en lenguajes orientados a objetos muy utilizados en la industria[70] tales como C#, Python, PHP o JavaScript (estos tres últimos de tipado dinámico).

Por lo expuesto se puede afirmar que la investigación de los últimos años ha demostrado que las métricas de cambio constituyen mejores predictores que las métricas estáticas. Sin embargo, existe un gran vacío en torno a los lenguajes de tipado dinámico ya que no han sido objeto de estudio.

[69] *"In most cases, one of three programming languages was used, i.e. C++ (35%), Java (34%) and C (15%), while others were rarely employed (8%). Future studies should consider other OO languages (e.g. C#) to investigate the performance of metrics in different programming languages. The metrics' effectiveness across different programming languages could be further investigated, since it appears to differ between languages"* ([22], Conclusions)

[70] A enero de 2020 el top 8 del índice TIOBE es: Java, C, Python, C++, C#, VB.net, JavaScript, PHP. El índice TIOBE mide la importancia de los lenguajes en la industria y se basa en la cantidad de ingenieros capacitados a nivel mundial, en la cantidad de cursos de capacitación y en los desarrollos de terceras partes. Índice online en www.tiobe.com/tiobe-index.

7 Conclusión general

En la presente investigación se analizó el código fuente, y las modificaciones hechas a éste, del proyecto MediaWiki durante el período de un año. Para ello se construyeron herramientas que computan las métricas y se inspeccionaron manualmente 2871 commits al repositorio de versiones. El producto MediaWiki está desarrollado en PHP, un lenguaje interpretado que posee características de tipado dinámico y que es ampliamente utilizado en la industria para la construcción de aplicaciones web.

Utilizando las métricas producidas como datos primarios se construyeron modelos de regresión con el objetivo de obtener el mejor conjunto de métricas que predigan defectos futuros de una clase.

Los mejores predictores de defectos hallados utilizando modelos de regresión logística son las métricas de cambio `revisions` y `ave_loc_added`. Es decir que la cantidad y tamaño de los cambios introducidos en una clase son los mejores predictores de los defectos futuros que dicha clase presentará. Esto confirma, en sintonía con los trabajos de otros autores, que las métricas de cambio son mejores predictores que las métricas estáticas.

Como respuesta a la pregunta de investigación[71] se puede concluir que sí es posible predecir defectos en un lenguaje dinámicamente tipado. Con los modelos propuestos se obtiene una precisión balanceada de 72% en el caso de las métricas de cambio y de 68% con métricas estáticas. El *recall* es de 75% en el modelo con métricas de cambio y 55% con métricas estáticas. Los falsos negativos son de 25% con métricas de cambio y de 45% con métricas estáticas.

No obstante su mejor desempeño, no puede soslayarse el mayor costo que implica actualmente la obtención de las métricas de cambio. Como se detalla en 4.5.2, los cambios introducidos en las clases por medio de los commits debieron ser organizados en una base de datos relacional a los efectos de calcular las métricas de cambio. Además, cada commit debió ser clasificado manualmente, uno por uno, según se estipula en el mismo apartado. Estas dos tareas, no realizadas previamente por la organización, son las que explican el alto costo de estas métricas versus las métricas estáticas, simplemente obtenibles con una herramienta que se desarrolla una sola vez. Esto lleva a recomendar que los equipos de desarrollo estandaricen los mensajes de commit, de modo de automatizar esta clasificación y reducir los costos de estas métricas que resultan ser estadísticamente los mejores predictores (cfr. Discusión 6.5).

[71] ¿Es posible predecir los defectos en un lenguaje dinámicamente tipado utilizando métricas estáticas y de cambio?

7.1 Trabajo futuro

La evolución histórica del lenguaje PHP permite medir la incidencia de defectos a medida que se aumenta la fortaleza de tipado del lenguaje debido a la presencia de mayores chequeos de tipos por parte del intérprete. Esto puede resultar de mucho interés para el conocimiento de los sistemas de tipado en general. El mismo análisis realizado en este trabajo se puede replicar con los commits de años siguientes, de forma de trabajar con código que incluye cada vez más declaraciones de tipos y, por lo tanto, permite mayor chequeo de errores. De este modo, se podría analizar lo que sucede con la distribución de defectos en la misma aplicación, a medida que los desarrolladores aprovechan cada vez más las posibilidades que el lenguaje incorpora con el tiempo.

Otra línea de trabajo futuro se relaciona con otros lenguajes de tipado dinámico, por ejemplo JavaScript y Python, ambos de uso muy extendido en la industria. Existen muchísimas aplicaciones desarrolladas en estos lenguajes, que además tienen un futuro muy prometedor. Es de interés experimentar con modelos predictores de defectos en estos lenguajes. Como se menciona en 6.6, el grupo de tipado estático C/C++/Java abarca el 85% de las investigaciones [22].

Otra posibilidad de trabajo futuro tiene que ver con la producción de herramientas factibles de ser aplicadas en la industria, en el contexto de aplicaciones y equipos de desarrollo reales. En este sentido, un posible camino a recorrer es la construcción de aplicaciones que presenten alarmas a los desarrolladores en el momento de realizar un commit. Esto es posible[72] en función de los archivos que están siendo modificados por este commit, un modelo de regresión previamente entrenado, y los datos que pueden deducirse del repositorio.

[72] Concretamente, una funcionalidad de este tipo podría construirse haciendo uso de los "*hooks*" del sistema de versiones Git. Documentación online: git-scm.com/book/en/v2/Customizing-Git-Git-Hooks (recuperado 8/2019)

8 Referencias bibliográficas

[1] S. R. Chidamber y C. F. Kemerer, «A metrics suite for object oriented design», *IEEE Trans. Softw. Eng.*, vol. 20, n.º 6, pp. 476-493, jun. 1994, doi: 10.1109/32.295895.

[2] Robiolo, Gabriela, «Métricas de Diseño Orientado a Objetos aplicadas en Java», Tesis de Maestría, Facultad de Informática, UNLP, 2004.

[3] R. Moser, W. Pedrycz, y G. Succi, «A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction», presentado en ICSE'08, May 10–18, Leipzig, Germany, 2008, doi: 10.1145/1368088.1368114.

[4] A. E. Hassan, «Predicting faults using the complexity of code changes», presentado en ICSE'09, May 16–24, Vancouver, Canada, 2009, pp. 78-88, doi: 10.1109/ICSE.2009.5070510.

[5] V. R. Basili, L. C. Briand, y W. L. Melo, «A validation of object-oriented design metrics as quality indicators», *IEEE Trans. Softw. Eng.*, vol. 22, n.º 10, pp. 751-761, oct. 1996, doi: 10.1109/32.544352.

[6] T. Gyimothy, R. Ferenc, y I. Siket, «Empirical validation of object-oriented metrics on open source software for fault prediction», *IEEE Trans. Softw. Eng.*, vol. 31, n.º 10, pp. 897-910, oct. 2005, doi: 10.1109/TSE.2005.112.

[7] N. Nagappan, T. Ball, y A. Zeller, «Mining metrics to predict component failures», 2006, p. 452, doi: 10.1145/1134285.1134349.

[8] K. Pan, S. Kim, y E. Whitehead, Jr., «Bug Classification Using Program Slicing Metrics», 2006, pp. 31-42, doi: 10.1109/SCAM.2006.6.

[9] B. Caglayan, A. Bener, y S. Koch, «Merits of using repository metrics in defect prediction for open source projects», en *2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, Vancouver, BC, Canada, 2009, pp. 31-36, doi: 10.1109/FLOSS.2009.5071357.

[10] M. D'Ambros, M. Lanza, y R. Robbes, «An extensive comparison of bug prediction approaches», 2010, pp. 31-41, doi: 10.1109/MSR.2010.5463279.

[11] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, y A. E. Hassan, «Revisiting common bug prediction findings using effort-aware models», 2010, pp. 1-10, doi: 10.1109/ICSM.2010.5609530.

[12] Jureczko, Marian, «Significance of different software metrics in defect prediction», *Softw. Eng. Int. J.*, vol. 1, n.º 1, pp. 86-95, 2011.

[13] E. Giger, M. D'Ambros, M. Pinzger, y H. C. Gall, «Method-level bug prediction», 2012, p. 171, doi: 10.1145/2372251.2372285.

[14] L. Madeyski y M. Jureczko, «Which process metrics can significantly improve defect prediction models? An empirical study», *Softw. Qual. J.*, vol. 23, n.º 3, pp. 393-422, sep. 2015, doi: 10.1007/s11219-014-9241-7.

[15] F. Zhang, A. Mockus, I. Keivanloo, y Y. Zou, «Towards building a universal defect prediction model», 2014, pp. 182-191, doi: 10.1145/2597073.2597078.

[16] Y. Kamei y E. Shihab, «Defect Prediction: Accomplishments and Future Challenges», 2016, pp. 33-45, doi: 10.1109/SANER.2016.56.

[17] J. Nam, W. Fu, S. Kim, T. Menzies, y L. Tan, «Heterogeneous Defect Prediction», *IEEE Trans. Softw. Eng.*, vol. 44, n.º 9, pp. 874-896, sep. 2018, doi: 10.1109/TSE.2017.2720603.

[18] R. B. Grady, «Measuring and Managing Software Maintenance», *IEEE Softw.*, vol. 4, n.º 5, pp. 35-45, sep. 1987, doi: 10.1109/MS.1987.231417.

[19] T. C. Jones, «Measuring programming quality and productivity», *IBM Syst. J.*, vol. 17, n.º 1, pp. 39-63, 1978, doi: 10.1147/sj.171.0039.

[20] N. E. Fenton y J. Bieman, *Software metrics: a rigorous and practical approach*, Third edition. Boca Raton: CRC Press, Taylor & Francie Group, 2014.
[21] A. Kaur, K. Kaur, y D. Chopra, «Application of Locally Weighted Regression for Predicting Faults Using Software Entropy Metrics», en *Proceedings of the Second International Conference on Computer and Communication Technologies*, vol. 379, S. C. Satapathy, K. S. Raju, J. K. Mandal, y V. Bhateja, Eds. New Delhi: Springer India, 2016, pp. 257-266.
[22] D. Radjenović, M. Heričko, R. Torkar, y A. Živkovič, «Software fault prediction metrics: A systematic literature review», *Inf. Softw. Technol.*, vol. 55, n.º 8, pp. 1397-1418, ago. 2013, doi: 10.1016/j.infsof.2013.02.009.
[23] M. H. Halstead, *Elements of software science*. New York: Elsevier, 1977.
[24] T. Hall, S. Beecham, D. Bowes, D. Gray, y S. Counsell, «A Systematic Literature Review on Fault Prediction Performance in Software Engineering», *IEEE Trans. Softw. Eng.*, vol. 38, n.º 6, pp. 1276-1304, nov. 2012, doi: 10.1109/TSE.2011.103.
[25] F. Shull *et al.*, «What we have learned about fighting defects», en *Proceedings Eighth IEEE Symposium on Software Metrics*, Ottawa, Ont., Canada, 2002, pp. 249-258, doi: 10.1109/METRIC.2002.1011343.
[26] Institute of Electrical and Electronics Engineers, IEEE Computer Society, Software Engineering Standards Committee, y IEEE-SA Standards Board, *IEEE standard classification for software anomalies*. New York: Institute of Electrical and Electronics Engineers, 2010.
[27] J. Nam y S. Kim, «CLAMI: Defect Prediction on Unlabeled Datasets (T)», 2015, pp. 452-463, doi: 10.1109/ASE.2015.56.
[28] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, y A. E. Hassan, «Studying just-in-time defect prediction using cross-project models», *Empir. Softw. Eng.*, vol. 21, n.º 5, pp. 2072-2106, oct. 2016, doi: 10.1007/s10664-015-9400-x.
[29] K. Tatroe, R. Lerdorf, y P. MacIntyre, *Programming PHP*, 3rd ed. Sebastopol, CA: O'Reilly Media, 2013.
[30] R. W. Sebesta, *Concepts of programming languages*, Eleventh edition, Global edition. Boston: Pearson, 2016.
[31] M. L. Scott, *Programming language pragmatics*, 3rd ed. Amsterdam ; Boston: Elsevier/Morgan Kaufmann Pub, 2009.
[32] The PHP Group, «PHP Language Reference». www.php.net/manual/en/langref.php (accedido 1/2020).
[33] R. E. Walpole, *Probabilidad y estadística para ingenieros*. México: Pearson, 2012.
[34] J. D. Kelleher, B. MacNamee, y A. D'Arcy, *Fundamentals of machine learning for predictive data analytics: algorithms, worked examples, and case studies*. Cambridge, Massachusetts London, England: The MIT Press, 2015.
[35] T. Zimmermann, R. Premraj, y A. Zeller, «Predicting Defects for Eclipse», en *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*, Minneapolis, MN, USA, 2007, pp. 9-9, doi: 10.1109/PROMISE.2007.10.
[36] M. Kondo, C.-P. Bezemer, Y. Kamei, A. E. Hassan, y O. Mizuno, «The impact of feature reduction techniques on defect prediction models», *Empir. Softw. Eng.*, vol. 24, n.º 4, pp. 1925-1963, ago. 2019, doi: 10.1007/s10664-018-9679-5.
[37] Q. Huang, X. Xia, y D. Lo, «Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction», *Empir. Softw. Eng.*, vol. 24, n.º 5, pp. 2823-2862, oct. 2019, doi: 10.1007/s10664-018-9661-2.
[38] Y. Kamei *et al.*, «A large-scale empirical study of just-in-time quality assurance», *IEEE Trans. Softw. Eng.*, vol. 39, n.º 6, pp. 757-773, jun. 2013, doi: 10.1109/TSE.2012.70.

Anexos

A Métricas obtenidas (datos primarios)

A.1 Métricas estáticas

A continuación se presentan con fines ilustrativos algunas filas de la tabla completa (1970 filas totales) con las métricas estáticas obtenidas:

Semestre	Id	Clase	LOC	CK_WMC	CK_RFC	CK_CBO	Rob_12	Rob_13	Rob_14	Rob_15	Gul_1	Gul_2	Gul_4	Gul_5	Gul_6	Bugfixes_semestre
2	23	Parser	3674	132	410	37	0	0	14,79	132	16	115	170	0	20	6
1	23	Parser	3639	131	408	36	0	0	14,73	131	18	112	61	108	20	4
2	361	lessc	2789	123	131	4	0	0	0	0	120	2	174	0	0	0
2	946	PasswordError	2722	0	24	0	0	0	0	24	0	0	0	0	0	0
2	21	User	2722	189	318	26	0	0	6,15	189	41	147	160	0	3	6
2	7	Title	2650	191	377	38	0	0	5,6	191	12	178	190	0	4	2
1	7	Title	2644	184	373	35	0	0	5,82	184	12	171	126	60	3	2
1	946	PasswordError	2638	0	25	0	0	0	0	24	0	0	0	0	0	0
1	21	User	2638	183	309	26	0	0	6,13	183	39	143	134	20	3	2
2	92	EditPage	2344	86	309	28	0	0	12,26	86	38	47	57	1	9	5
1	92	EditPage	2335	87	312	28	0	0	12,07	87	39	47	34	24	8	5
2	-1	Page	1938	0	0	0	0	0	0	0	0	0	0	0	0	0
2	377	PoolWorkArticleView	1938	8	30	5	66,67	3	5,63	10	0	7	4	0	2	0
2	101	WikiPage	1938	99	290	35	0	16	8,22	99	14	84	111	0	49	2
1	-1	Page	1887	0	0	0	0	0	0	0	0	0	0	0	0	0
1	377	PoolWorkArticleView	1887	8	30	5	66,67	3	5,63	10	0	7	0	4	2	0
1	101	WikiPage	1887	100	289	35	0	16	8	100	16	83	57	54	49	6
1	33	OutputPage	1859	164	331	25	0	4	4,43	178	89	74	83	80	6	5
2	33	OutputPage	1857	168	338	25	0	4	4,35	182	90	77	164	0	7	2
2	28	DatabaseBase	1737	170	202	11	0	0	4,09	187	35	134	233	0	5	2
2	1177	DatabaseType	1737	18	18	0	0	0	1	18	1	17	18	0	0	0
2	411	IDatabase	1737	0	0	0	0	0	0	0	0	0	0	0	0	0
1	28	DatabaseBase	1712	168	200	12	0	0	4,1	185	37	130	132	98	5	4
1	1177	DatabaseType	1712	18	18	0	0	0	1	18	1	17	10	8	0	0
1	411	IDatabase	1712	0	0	0	0	0	0	0	0	0	0	0	0	0
2	26	LocalFile	1701	64	251	26	22,88	0	7,3	155	20	43	73	0	4	0
2	158	LocalFileDeleteBatch	1701	10	41	2	0	0	9,5	10	4	5	5	0	1	0
2	542	LocalFileMoveBatch	1701	9	33	3	0	1	9,56	9	3	5	3	0	2	0
2	159	LocalFileRestoreBatch	1701	9	38	7	0	0	10,22	9	4	4	7	0	1	0
1	-1	JSCompilerContext	1666	1	1	0	0	0	0	0	0	0	1	0	0	0
1	-1	JSMinPlus	1666	6	9	3	0	0	0	0	5	0	8	0	0	0
1	-1	JSNode	1666	4	5	0	0	0	0	0	3	0	6	0	0	0
1	500	JSParser	1666	12	25	4	0	0	0	0	11	0	20	0	0	0
1	-1	JSToken	1666	0	0	0	0	0	0	0	0	0	0	0	0	0
1	-1	JSTokenizer	1666	13	13	2	0	0	0	0	12	0	9	0	0	0
2	-1	JSCompilerContext	1666	1	1	0	0	0	0	0	0	0	1	0	0	0
2	-1	JSMinPlus	1666	6	9	3	0	0	0	0	5	0	8	0	0	0
2	-1	JSNode	1666	4	5	0	0	0	0	0	3	0	6	0	0	0
2	500	JSParser	1666	12	25	4	0	0	0	0	11	0	20	0	0	1
2	-1	JSToken	1666	0	0	0	0	0	0	0	0	0	0	0	0	0
2	-1	JSTokenizer	1666	13	13	2	0	0	0	0	12	0	9	0	0	0

1	26	LocalFile	1660	61	251	26	23,73	0	7,57	151	20	40	65	4	4	1
1	158	LocalFileDeleteBatch	1660	10	41	2	0	0	9,3	10	4	5	3	1	1	0
1	542	LocalFileMoveBatch	1660	9	33	3	0	1	9,22	9	3	5	3	0	2	0
1	159	LocalFileRestoreBatch	1660	9	39	7	0	0	10,22	9	4	4	7	0	1	0
2	34	ApiBase	1528	78	135	12	0	0	4,92	92	22	55	69	0	0	2
1	34	ApiBase	1505	76	133	12	0	0	4,95	90	21	54	46	20	0	0
2	245	BaseTemplate	1348	15	39	5	21,43	0	9,6	27	8	7	17	0	0	0
2	-1	DummyLinker	1348	1	1	0	0	0	1	1	0	1	2	0	0	0
2	11	Linker	1348	68	171	16	0	12	8,54	68	2	66	185	0	13	3
2	1011	MediaWiki_I18N	1348	2	3	0	0	0	6	2	2	0	3	0	0	0
2	715	QuickTemplate	1348	15	17	1	0	0	1,6	15	4	10	14	0	0	0
2	246	SkinTemplate	1348	18	209	15	2,33	0	22,89	102	6	12	15	0	2	2
1	-1	DummyLinker	1336	1	1	0	0	0	1	1	0	1	2	0	0	0
1	11	Linker	1336	68	171	16	0	12	8,47	68	2	66	98	87	13	4
1	245	BaseTemplate	1315	14	38	5	21,43	0	10	26	8	6	13	3	0	1
1	1011	MediaWiki_I18N	1315	2	3	0	0	0	6	2	2	0	3	0	0	0

A.2 Métricas de cambio

De igual forma, se presentan a continuación algunas filas de la tabla general de métricas de cambio (1060 filas totales):

trimestre	nombre	revisions	loc_added	max_loc_added	ave_loc_added	loc_deleted	max_loc_deleted	ave_loc_deleted	codechurn	max_codechurn	ave_codechurn	authors	ave_changeset	max_changeset	age	weighted_age	bugfixes	bugfixes_prox_periodo
1	SpecialWatchlist	20	721	201	36,05	651	117	32,55	70	85	3,5	4	1,15	4	302	290,87	3	1
3	Title	19	131	27	6,89	471	285	24,79	-340	24	-17,89	11	1,26	9	598	591,87	2	0
3	Installer	18	157	66	8,72	108	32	6	49	34	2,72	11	2,5	7	230	219,95	4	0
1	SpecialSearch	17	53	15	3,12	73	20	4,29	-20	5	-1,18	4	1,59	8	302	296,49	0	4
1	SpecialRecentChan-ges	16	260	123	16,25	486	127	30,38	-226	12	-14,13	4	1,69	5	302	293,07	1	0
3	SpecialSearch	15	47	20	3,13	82	22	5,47	-35	7	-2,33	7	1,4	10	328	319,98	5	1
3	User	15	207	124	13,8	152	86	10,13	55	38	3,67	11	2	16	598	590,49	5	1
1	SwiftFileBackend	14	1052	841	75,14	1032	881	73,71	20	28	1,43	1	0,71	3	86	75	4	2
2	Parser	14	286	165	20,43	104	65	7,43	182	100	13	10	2,93	16	315	308,49	4	2
3	EditPage	14	103	44	7,36	227	78	16,21	-124	37	-8,86	10	2,14	15	583	576,81	4	1
2	SpecialSearch	13	122	49	9,38	128	79	9,85	-6	49	-0,46	5	0,77	4	315	311,42	4	5
3	Skin	13	56	16	4,31	126	77	9,69	-70	5	-5,38	6	2,46	7	598	590,93	0	1
1	Title	13	256	110	19,69	226	180	17,38	30	32	2,31	7	2,77	10	572	566,11	2	0
2	LocalFile	13	169	54	13	98	35	7,54	71	50	5,46	5	0,85	4	137	127,88	1	0
1	MediaWiki	12	97	42	8,08	94	49	7,83	3	14	0,25	6	1,58	10	431	425,52	1	2
3	OutputPage	12	121	43	10,08	130	42	10,83	-9	15	-0,75	9	0,83	4	598	593,62	1	1
2	WikiPage	11	169	102	15,36	295	190	26,82	-126	37	-11,45	9	2,27	13	157	147,3	3	1
3	ApiBase	11	1186	1000	107,82	1155	957	105	31	76	2,82	5	11,09	74	421	415,88	2	0

B Regresión lineal

A continuación se enumeran los mejores modelos de regresión lineal que se han encontrado para la muestra bajo estudio. Se implementaron varios programas con la aplicación estadística R a los efectos de testear distintas combinaciones de variables.

Es interesante observar que no ha sido posible construir un modelo de regresión lineal estadísticamente significativo para los datos bajo estudio.

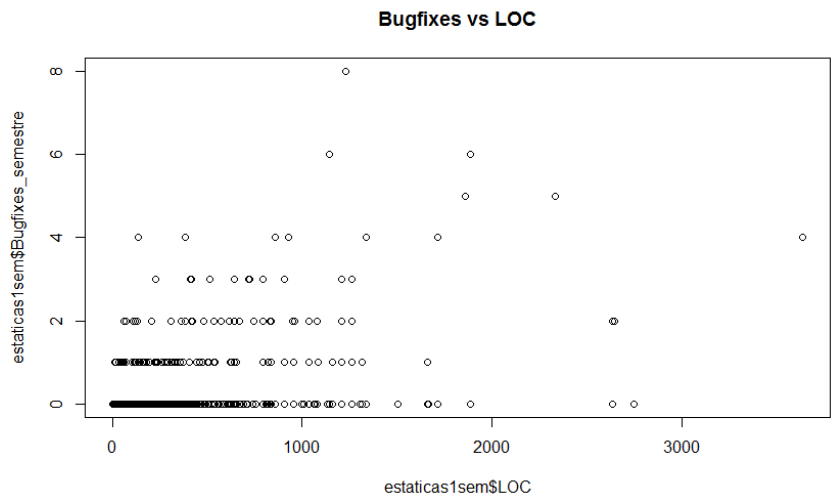
B.1 Métricas estáticas

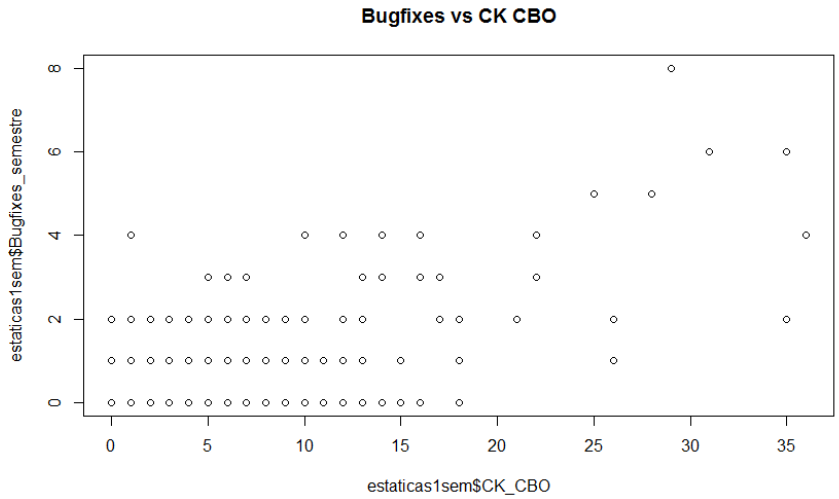
En este caso, la variable independiente a explicar es `Bugfixes_semestre`, que indica cuántas veces se ha modificado una clase con un commit de tipo *"Fault Repairing"*.

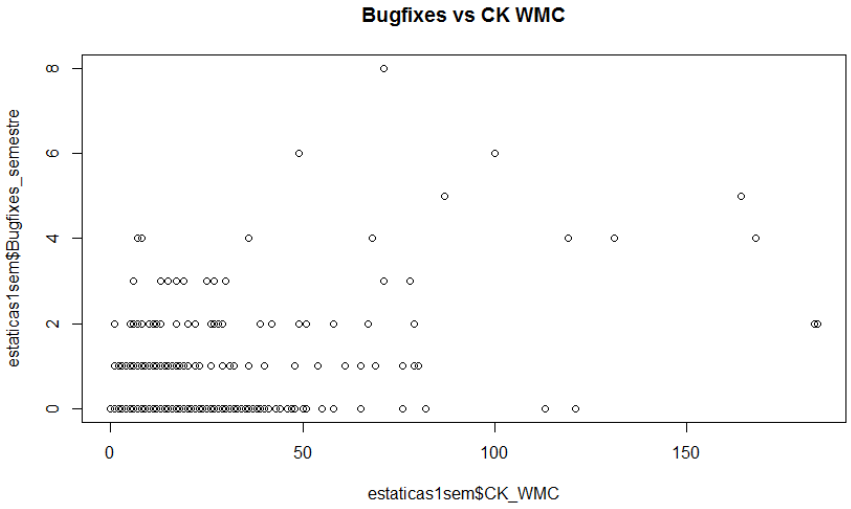
Variables predictoras	R^2	Adjusted R^2	p-value
Rob_12 + CK_CBO	0,302	0,301	< 2.2e-16
Rob_13 + CK_CBO	0,299	0,298	< 2.2e-16
Rob_14 + CK_WMC	0,265	0,264	< 2.2e-16
Rob_14 + Gul_2	0,242	0,241	< 2.2e-16
CK_CBO + Rob_12 + Rob_13	0,302	0,301	< 2.2e-16
Rob_14 + Gul_2 + Gul_1	0,268	0,267	< 2.2e-16
Rob_13 + Rob_14 + CK_WMC	0,266	0,265	< 2.2e-16

Se puede concluir que los modelos de regresión lineal, utilizando métricas estáticas, no proveen de un predictor ni combinación de éstos que sea estadísticamente significativa para el presente producto en su actual ciclo de vida (R^2 muy bajo).

De algún modo, se puede encontrar en la dispersión de las variables una explicación de por qué no se puede elaborar con éstas un modelo de regresión lineal:


Bugfixes vs LOC
estaticas1sem$Bugfixes_semestre
estaticas1sem$LOC
0
2
4
6
8
0
1000
2000
3000


Bugfixes vs CK CBO
estaticas1sem$Bugfixes_semestre
estaticas1sem$CK_CBO
0
2
4
6
8
0
5
10
15
20
25
30
35


Bugfixes vs CK WMC
estaticas1sem$Bugfixes_semestre
estaticas1sem$CK_WMC
0
2
4
6
8
0
50
100
150

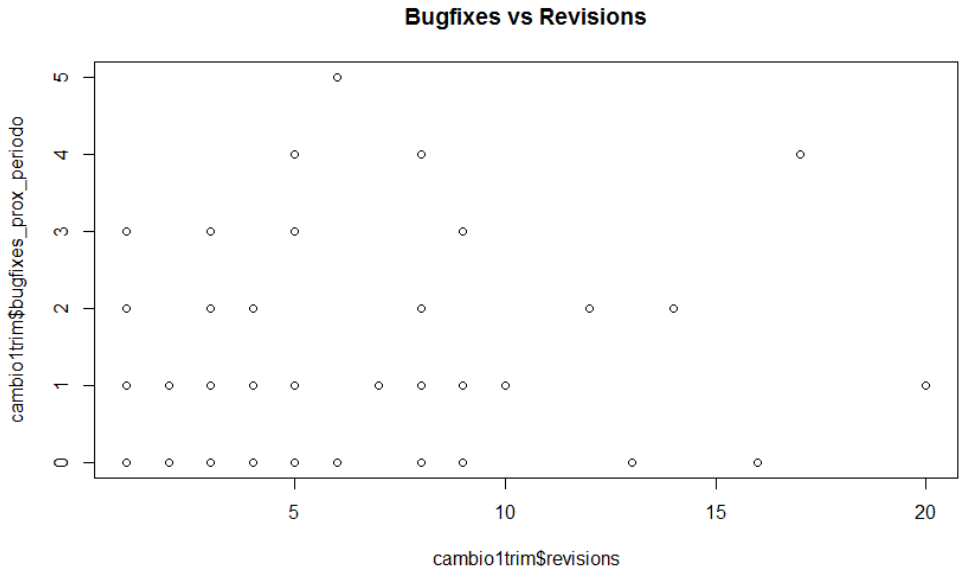
B.2 Métricas de cambio

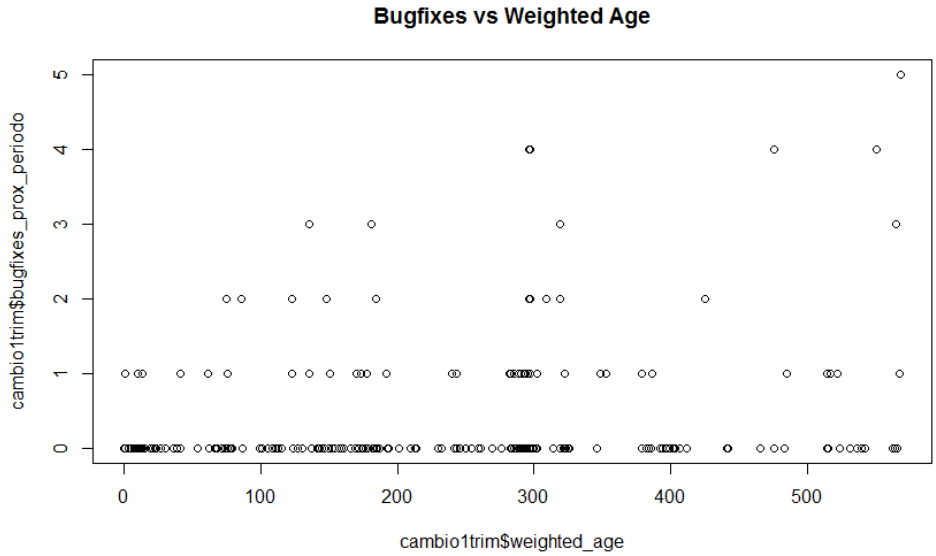
En este caso, la variable a explicar es `bugfixes_prox_periodo`, que indica cuántas veces se ha modificado una clase con un commit de tipo FR en el trimestre siguiente al considerado para calcular las métricas de cambio. El análisis de las métricas de cambio se realizó por trimestres y, además, los *bugfixes* del trimestre bajo análisis también pueden ser utilizados como variable predictora de los *bugfixes* del siguiente trimestre.

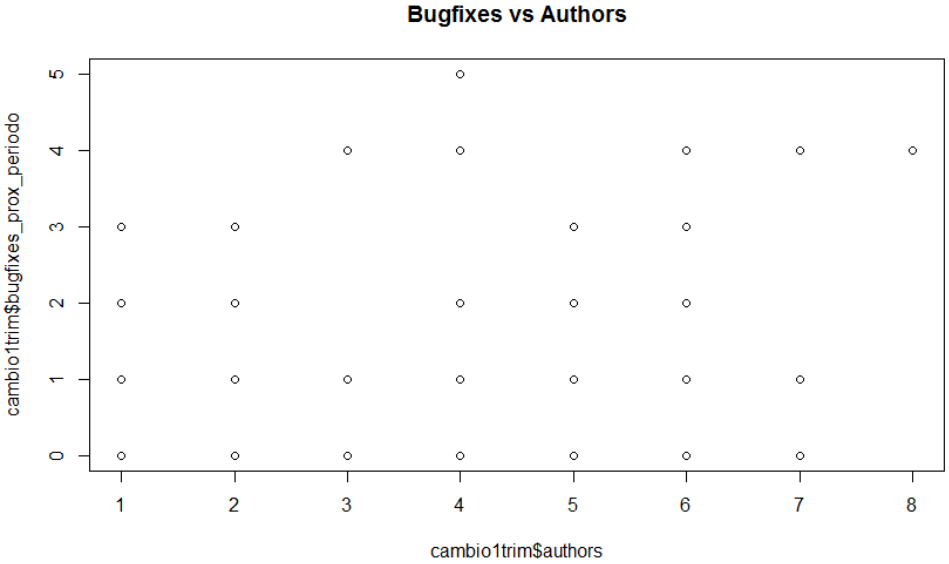
Variables predictoras	R^2	Adjusted R^2	p-value
revisions + weighted_age	0,163	0,162	< 2.2e-16
revisions + age	0,162	0,161	< 2.2e-16
age + authors	0,162	0,161	< 2.2e-16
revisions + max_changeset + age	0,169	0,167	< 2.2e-16
revisions + age + codechurn	0,168	0,166	< 2.2e-16

De igual manera, las métricas de cambio tampoco son estadísticamente significativas para crear un modelo de regresión lineal predictor de *bugfixes* en el producto bajo estudio, siendo las presentadas en la tabla las mejores combinaciones encontradas.

Se demuestra a continuación la dispersión de las observaciones, ahora considerando las métricas de cambio:







Se puede concluir que la regresión lineal no es una buena opción a los efectos de construir modelos predictores de defectos en el producto bajo análisis. Esto se debe a dos factores, entre los que se encuentran la madurez del producto y la metodología de desarrollo (integración continua). Se cree que estos dos factores contribuyen a la dispersión de las variables, lo que impide crear un modelo significativo con este método en el actual ciclo de vida.

C Comandos Git más utilizados

Posicionar el repositorio en el primer commit del primer y segundo semestre, respectivamente:

```
git checkout ee01799f47013d0d6dee73b9612e81eb82e3e460

git checkout cae5da1ca330021d966f5a81ec5b4760a3da0568
```

Obtención de todos los commits del año bajo estudio:

```
git checkout master

git log --after="2014-1-1" --before="2014-12-31"  --pretty=fuller --date=short
--no-merges -p -- includes > "commits_master_2014.txt"
```

Estos commits son los interpretados con la herramienta construida para obtener las métricas de cambio (cfr. 4.5.2). El archivo obtenido contiene 500 mil líneas de texto y pesa 26 MB e incorpora todos los cambios realizados al código fuente de MediaWiki en el transcurso de un año de trabajo.

Obtención de la fecha en la que una clase se agregó al repositorio:

```
git log --diff-filter=A --date=short -- includes/Clase.php
```

D Expresiones regulares

Las siguientes expresiones son utilizadas en el programa que calcula las métricas estáticas. Se incluyen aquí para referencia futura.

```
Llamada a método. Ejemplo: $aux->getName()
/\$([a-zA-Z0-9_]+)->([a-zA-Z0-9_]+)\s*\(/
```

```
Llamada estática. Ejemplo: AjaxService::callNext()
/([a-zA-Z0-9_\)\(]+)::([a-zA-Z0-9_]+)\(/
```

```
Acceso a miembro estático. Ejemplo: Revision::DELETED_USER
/([a-zA-Z0-9_\)\(]+)::([a-zA-Z0-9_]+)/
```

```
Instanciación de una clase. Ejemplo: new \User()
/new\s+([\\a-zA-Z0-9_]+)\s*\(/
```

```
Atrapado de excepción. Ejemplo: catch(MWException $ex) {
/catch\s+\(\s+([a-zA-Z0-9_]+)\s+\$/
```