risti
Revista Ibérica de Sistemas e Tecnologias de Informação
Revista Ibérica de Sistemas y Tecnologías de Información

Recebido/Submission: 10/12/2017
Aceitação/Acceptance: 05/02/2018

Caracterización y Detección Automática de Bad Smells MVC

Perla Velasco-Elizondo[1], Lucero Castañeda-Calvillo[2], Alejandro García-Fernández[3], y Sodel Vazquez-Reyes[4]

pvelasco@uaz.edu.mx, b160629@sagitario.cic.ipn.mx, agarciafdz@cimat.mx, vazquezs@uaz.edu.mx

[1] Universidad Autónoma de Zacatecas, Zacatecas, ZAC, 98000, México.

[2] Centro de Investigación en Computación, Ciudad de México, 07738, México.

[3] Centro de Investigación en Matemáticas, Zacatecas, ZAC, 98060, México.

[4] Universidad Autónoma de Zacatecas, Zacatecas, ZAC, 98000, México.

DOI: 10.17013/risti.26.54–67

Resumen: Los *bad smells* son causa frecuente de acumulación de deuda técnica; término que se refiere al costo en que se incurre por utilizar un enfoque de diseño o desarrollo apresurado y descuidado. Existen trabajos sobre la caracterización de *bad smells,* así como sobre enfoques para detectarlos y corregirlos automáticamente. Sin embargo, pocos de estos trabajos caracterizan, detectan y corrigen *bad smells* arquitectónicos. Este trabajo es un esfuerzo inicial para a llenar este vacío y contribuir en: (i) la caracterización de *bad smells* relevantes al estilo de arquitectura MVC, y (ii) la detección automática de estos *bad smells* utilizando técnicas de análisis estático de software. Los resultados obtenidos muestran que la mayoría de los *bad smells* definidos existen en la práctica, y que la estrategia de detección propuesta reduce en un amplio margen el tiempo requerido para detectar *bad smells* mediante una revisión de código de forma manual.

Palabras-clave: Arquitectura de Software; Bad Bad smells; Análisis Estático; MVC; Yii.

Characterization and Automatic Detection of Bad Smells MVC

Abstract: Bad smells are a frequent cause of technical debt, which denotes the cost of adopting a quick and dirty design or development approach. There are works on characterizing bad smells as well as on detecting and fixing them automatically. However, few of these works characterize, detect and fix architectural bad smells. The work presented in this article represents an initial effort to fill this by contributing to: (i) the characterization of bad smells tha are relevant to the MVC architecture style, and (ii) the automatic detection of these using static analysis of software techniques. The obtained results show that most of the defined bad smells exist in practice and that the proposed detection method reduces by a wide margin the detection time required by a code review.

***Keywords**:* Software Architecture, Bad Bad smells, Static Analysis, MVC, Yii.

1. Introducción

El éxito de un proyecto de desarrollo de sistemas no sólo depende de que el sistema creado se termine dentro plazo y presupuesto establecido, sino también que genere valor (Espejo Chavarría, Bayona Oré, & Pastor, 2016; Morales Huanca & Bayona Oré, 2017). Así, durante la actividad de diseño de sistemas de software, es muy importante definir un modelo que permita satisfacer los comportamientos y características de calidad especificadas durante la actividad de análisis de requerimientos. Para logar esto es recomendable que se haga uso de conceptos de diseño, como por ejemplo los patrones. Un patrón describe una solución general a un problema recurrente en el desarrollo de sistemas (Gamma et al. 1995). Sin embargo, el uso de patrones de manera diferente a la definida propicia la generación de *bad smells*.

En Ingeniería de Software el término *bad smell* es utilizado para denotar un síntoma como consecuencia de un diseño pobre o implementación que negativamente impacta las propiedades de un sistema de software (Fowler et al. 1999), por ejemplo, mantenibilidad, reusabilidad. Un *bad smell* es usualmente utilizado para indicar un problema en el software. Aunque no hay un consenso universal en el área, generalmente se acepta que los *bad smells* pueden ocurrir en diferentes niveles de abstracción: desde el código fuente, por ejemplo, una *long parameter list* (Source Makings, Code Smells 2017) hasta la arquitectura, por ejemplo, *connector envy* (Garcia et al. 2009). A estos últimos, esto es a los patrones de arquitectura, también se les llama estilos arquitectónicos.

Los bad smells son una causa frecuente de acumulación de deuda técnica (Brown et al. 2010). La deuda técnica, es un término que hace referencia al costo en el que se incurre por un enfoque de diseño o desarrollo apresurado y descuidado en el que se ignoran principios y conceptos establecidos, en lugar de usar un enfoque sistemático y disciplinado. Por lo tanto, detectar y corregir *bad smells* se vuelve relevante para el diseño y desarrollo de sistemas de software.

En años recientes, la comunidad de investigación ha estado activamente identificando las características de *bad smells,* por ejemplo (Ganesh, Sharma, and Suryanarayana 2013), así como en el desarrollo de enfoques para detectarlos y corregirlos por ejemplo (Fernandes et al. 2016). Sin embargo, la mayoría de estos enfoques se centran en *bad smells* que ocurren en niveles de abstracción más bajos y pocos de ellos se caracterizan, identifican y corrigen a nivel de arquitectura. Adicionalmente, en estos trabajos de presta poca atención a la realización de estas actividades en el contexto de los estilos arquitectónicos, estructuras de diseño comúnmente utilizadas para el desarrollo de software.

El trabajo presentado en este artículo, es un enfoque inicial para llenar ese vacío y contribuir en: (i) la caracterización de *bad smells* que son relevantes para el estilo de arquitectura Modelo-Vista-Controlador (Model-View-Controller o MVC, por sus siglas en inglés) (Best MVC Practices 2017), que ha sido ampliamente adoptado en frameworks para el desarrollo de sistemas Web, y (ii) la detección automática de estos *bad smells* utilizando técnicas de análisis estático de software en sistemas implementados con el Framework Yii (yiiFramework 2017). Los resultados obtenidos muestran que los *bad smells* definidos existen en la práctica y permiten tener una idea inicial sobre cuáles ocurren con mayor frecuencia. Con respecto al método de detección automática, los resultados muestran que el método de detección propuesta reduce en un amplio margen el tiempo requerido para detectar *bad smells* mediante una revisión de código de forma manual

El contenido de este documento está organizado de la siguiente manera. En la sección 2, se explica el estilo arquitectónico MVC y, en función de sus limitaciones generales, se define una caracterización de *bad smells* relacionados. En la sección 3, se explica el método usado para la detección automática de estos *bad smells*. En la Sección 4 se presentan obtenidos al realizar dos experimentos para determinar la relevancia de las contribuciones esperadas. En la Sección 5 se presenta una discusión del trabajo relacionado. Finalmente, en la Sección 6, se presentan las conclusiones de este trabajo, así como algunas líneas de trabajo futuro.

2. Estilo Arquitectónico MVC y *Bad Smells* Relacionados

La arquitectura de un sistema de software es un modelo de alto nivel de un sistema definido en términos componentes, conectores y propiedades de ambos (Bass, Clements, and Kazman 2012). Si bien es posible especificar la arquitectura de un sistema utilizando este vocabulario genérico, en dominios de aplicación específicos es mejor adoptar un vocabulario más especializado. A este vocabulario especializado da lugar a los estilos arquitectónicos (Bass, Clements, and Kazman 2012), como por ejemplo cliente servidor, tres capas o MVC.

El estilo arquitectónico MVC ha sido ampliamente adoptado para el diseño e implementación de sistemas Web. Actualmente, varios frameworks de desarrollo permiten la construcción de sistemas Web utilizando este estilo, como por ejemplo, Spring (Spring 2018), Django (Django 2018), Rails (Rails 2018), Laravel (Laravel 2018), and Yii. Este estilo separa la lógica de negocio de la lógica de presentación, lo que da como resultado un sistema más fácil de probar y mantener. La Figura 1 muestra una vista estática del estilo arquitectónico MVC y la Tabla 1 describe los elementos que esta vista incluye.

El estilo arquitectónico MVC define la siguientes restricciones generales:

- El Modelo no debe tratar directamente con el procesamiento de las solicitud del usuario final. Por ejemplo, su implementación no debe contener variables $_GET, $_POST.
- El Modelo no debe tratar directamente con la presentación de los datos para la solicitud del usuario final. Por ejemplo, su implementación no debe contener código de presentación HTML.
- La Vista no debe tratar directamente de realizar un acceso explícito a los datos del sistema. Por ejemplo, su implementación no debe contener código para consultas a Bases de Datos.
- La Vista no debe tratar directamente con la solicitud del usuario final. Por ejemplo, su implementación no debe contener variables $_GET, $_POST.
- El Controlador, no debe tratar directamente de realizar un acceso explícito a los datos del sistema. Por ejemplo, su implementación no debe contener código para consultas a Bases de Datos.

El Controlador no debe tratar directamente con la presentación de los datos para la solicitud del usuario final. Por ejemplo, su implementación no debe contener código de presentación HTML.

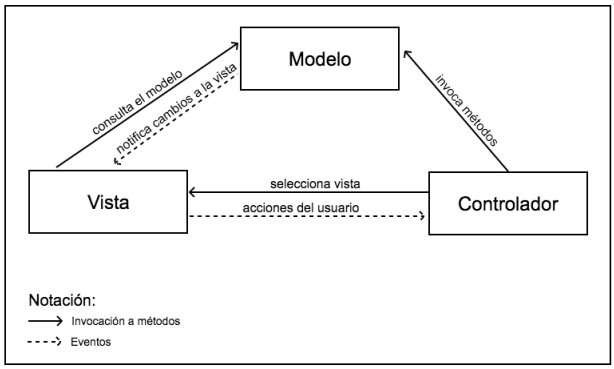

Figura 1 – Representación gráfica del estilo arquitectónico MVC.

Elemento	Descripción
Modelo	Representa los datos del sistema y las reglas de negocio que rigen el acceso a los datos. El Modelo notifica a la Vista si se ha hecho un cambio a los datos.
Vista	Es la representación del Modelo en un formato deseado por el usuario final. La Vista consulta al Modelo para cualquier cambio en los datos.
Controlador	Es un intermediario entre la Vista y el Modelo. El Controlador recibe las acciones del usuario final, las solicitudes de comandos provenientes de la Vista, invoca los métodos requeridos en el Modelo y cambia la presentación de la Vista del Modelo cuando es necesario.

Tabla 1 – Descripción de los elementos del estilo arquitectónico MVC.

2.1. Caracterización de *Bad Smells* MVC

El concepto de *bad smell* arquitectónico fue originalmente usado en (Lippert and Roock 2006) para describir la aparición de un problema de diseño o implementación que ocurre a un alto nivel de abstracción. Esto es, problema de diseño o implementación en la arquitectura. Las causas de los *bad smells* arquitectónicos incluyen, entre otros, aplicar un concepto de diseño arquitectónico en un contexto inapropiado, mezclar responsabilidades entre componentes o aplicar abstracciones de diseño en el nivel a un nivel de granularidad incorrecto (Lippert and Roock 2006).

Aunque casi todos los desarrolladores Web conocen el estilo MVC, su implementación correcta elude a muchos de estos (Best MVC Practices 2017). Frecuentemente, las restricciones generales, que se definieron en la sección anterior, no son respetadas. Esto da como resultado decisiones de diseño deficientes de diseño o implementación, que en este trabajo llamamos *bad smells* MVC. Específicamente, y considerando las restricciones antes mencionadas, la Tabla 2 describe una caracterización de *bad smells* MVC.

Es importante resaltar que esta caracterización de *bad smells* no es propuesta desde cero por nosotros; se deriva de una recopilación de información extraída de varias fuentes (por ejemplo, blogs de desarrolladores, sitios de preguntas y respuestas), que hemos definido en este trabajo de manera más completa y consolidada.

Id	Elemento	Descripción
1.	*El Modelo incluye cálculos y/o datos de la Vista*	Ocurre cuando el Modelo realiza responsabilidades relacionadas a presentación o recepción de datos en la interfaz del usuario (por ejemplo, código HTML).
2.	El Modelo incluye cálculos y/o datos del Controlador	Ocurre cuando el Modelo accede de forma directa a información sobre las solicitudes del usuario (por ejemplo, acceso directo a las variables $_GET, $_POST).
3.	La Vista incluye cálculos y/o datos del Modelo	Ocurre cuando la Vista tiene responsabilidades relacionadas a la consulta y persistencia de datos (por ejemplo, código de acceso a la base de datos).
4.	La Vista incluye cálculos y/o datos del Controlador	Ocurre cuando la Vista accede de forma directa a información sobre las solicitudes del usuario (por ejemplo, acceso directo a las variables $_GET, $_POST).
5.	El Controlador incluye cálculos y/o datos de la Vista	Ocurre cuando el Controlador Modelo realiza responsabilidades relacionadas a presentación o recepción de datos en la interfaz del usuario (por ejemplo, código HTML).
6.	El Controlador incluye cálculos y/o datos del Modelo	Ocurre cuando el Controlador tiene responsabilidades relacionadas a la consulta y persistencia de datos (por ejemplo, código de acceso a la base de datos).

Tabla 2 – Caracterización de *bad smells* relevantes en el estilo arquitectónico MVC.

Una vez presentada la caracterización de *bad smells* relevantes en el estilo arquitectónico MVC, que es la primera contribución presentada en este trabajo, en la siguiente sección describimos el método de detección automática de estos *bad smells*.

3. Detección automática de Smells MVC

La herramienta de detección automática de *smells* MVC presentada en este trabajo funciona para sistemas implementados en el framework Yii. Yii es un framework de desarrollo creado en el año 2008. Yii es orientado a objetos, usa en el lenguaje PHP (PHP 2018) y es ampliamente utilizado para el desarrollo de una amplia gama de sistemas Web. Gracias a que no demanda grandes recursos para la ejecución de los sistemas implementados, Yii es adecuado para desarrollar sistemas Web de gran tráfico como portales, foros, sistemas de administración de contenidos, sistemas de comercio electrónico, etc. Como la mayoría de los frameworks PHP, Yii es un framework MVC (modelo-vista-controlador).

En lugar de construir una herramienta desde el principio, se decidió extender las capacidades de la herramienta PHP_CodeSniffer. En las siguientes secciones se proveen los detalles correspondientes.

3.1. PHP_CodeSniffer.

PHP_CodeSniffer es una herramienta de análisis estático de software (PHP_CodeSniffer 2018). El análisis estático de software es un tipo de análisis que se realiza sin ejecutar el código fuente de un sistema –en contraste con el análisis dinámico de software que se realizado sobre un sistema en ejecución. En la mayoría de los casos, el análisis se

realiza en alguna versión del código fuente y en otros casos se realiza en el código objeto. El término se aplica generalmente a los análisis realizados automáticamente usando una herramienta –en contraste con el análisis realizado por un humano, que es denominado comprensión de programas o revisión de código.

PHP_CodeSniffer utiliza archivos denominados *sniffs* para detectar violaciones al conjunto de reglas definidas en un estándar de código (PHP_CodeSniffer 2017). Un estándar de código puede definirse como un conjunto de convenciones que deben utilizarse para escribir archivos de código fuente con el objetivo de lograr estructuras de código que tienen más calidad. Algunos ejemplos de estándares de codificación incluyen: el estándar de código PEAR (PEAR - PHP Extension and Application Repository 2018) o el de GNOME (GNOME Programming Guidelines 2018).

PHP_CodeSniffer funciona sobre la base de separar el contenido de un archivo de código en bloques de construcción, llamados tokens y luego validarlos para verificar una variedad de aspectos especificados en un estándar de codificación. Se pueden usar múltiples estándares de codificación dentro de PHP_CodeSniffer. Después del proceso de análisis estático, PHP_CodeSniffer genera una lista de las violaciones que han sido encontradas, con los correspondientes mensajes de error y números de línea.

Un estándar de código en PHP_CodeSniffer consiste en una colección de archivos de código abierto denominados archivos *sniff*. Cada archivo *sniff* verifica una convención del estándar de código y puede ser codificada en PHP, JavaScript, o CSS. Por lo tanto es posible crear nuevos estándares de codificación, reutilizando, extendiendo y construyendo nuevos archivos *sniff*.

3.2. Estándar de Código para Detectar Smells MVC.

Para detectar *bad smells* Arquitectónicos MVC, construimos una estándar de código llamado Yii MVC. Los archivos *sniff* en este estándar de código son clases PHP, que se utilizan en seis algoritmos de detección para identificar los *bad smells* MVC definidos en la Tabla 2.

La Figura 2 muestra un extracto de código de un archivo *sniff*. Este archivo *sniff*, en comparación con otros, es utilizado para detectar el *bad smell* número 6, que ocurre cuando el Controlador incluye cálculos y/o datos del Modelo. Como se explicó anteriormente, este *bad smell* está relacionado con el problema que aparece cuando un Controlador realiza las responsabilidades de un Modelo. Por lo tanto en términos de código, este *bad smell* resulta incluir un modelo de código para leer escribir o actualizar los datos e incluso almacenarlos, en una base de datos.

Como se muestra en la Figura 2, una clase de tipo Sniff debe implementar la interfaz **`PHP_CodeSniffer_Sniff`**. Esta interfaz declara dos funciones que son necesarias para el análisis estático del código: la función **`register`** y la función **`process`**. La función **`register`** permite a un *sniff* recuperar los tipos de tokens que van a analizar, en este caso el análisis es sobre cadenas de texto. Una vez que las cadenas están listas para ser analizadas, se llama a la función **`process`** con una referencia al archivo de código que se está analizando y otra referencia a una pila en donde se encuentran las cadenas de texto: la primer referencia es el parámetro denominado **`PHP_CodeSniffer_File`** y la segunda referencia es el parámetro **`$stackPtr`**, respectivamente.

Se puede encontrar información sobre una cadena de texto con el método **`getTokens`** en el archivo que contiene el código que se está analizando (línea 13). Este método regresa un arreglo de cadenas, que se indexa de acuerdo a la posición de la cadena en la pila de cadenas. Las cadenas tienen un índice denominado **`content`** para acceder a su contenido tal y como aparece en el código.

El análisis implementado detecta la declaración **`delete`** (línea 14) en el código de un archivo que pertenece al Controlador. Esta declaración permite la eliminación de registros en la base de datos. Si la declaración **`delete`** se encuentra en el código se activa el mensaje de detección del *bad smell* (líneas 17-20). Un *sniff* indica que ha ocurrido un error mediante el método **`addError`**, el cual es el encargado de generar el mensaje de error como primer argumento, y la posición en la pila donde fue detectado el error como segundo argumento, incluido el código para identificar de forma única el tipo de error dentro del *sniff* y un arreglo de datos utilizado dentro del mensaje de error.

```
    ...
5:  class Yii_Sniffs_Bad smellsArchitectureMVC_SA6_DeleteSniff implements
6:  PHP_CodeSniffer_Sniff {
7:      public function register() {
8:          return array(T_STRING);
9:      }
10:
11:     public function process(PHP_CodeSniffer_File $phpcsFile, $stackPtr){
12:         $name = $phpcsFile->getFilename();
13:         $tokens = $phpcsFile->getTokens();
14:         if ($tokens[$stackPtr]['content'] === 'delete') {
15:             ...
16:             if (strpos($name, "controllers")) {
17:                 $error = '6. Controller includes Model's computations and/or
18:                     data --> Executes a SQL statement.;
19:                     Delete found';
20:                 $data  = array(trim($tokens[$stackPtr]['content']));
21:                 $phpcsFile->addError($error, $stackPtr, 'Found', $data);
22:             }
23:         }
24:     }
    ...
```

Figura 2 – Extracto de un archivo *sniff* para el *bad smell* número 6: El Controlador incluye cálculos y/o datos del Modelo.

Cuando PHP_CodeSniffer se ejecuta en un proyecto Yii usando el estándar de código Yii MVC, existen dos tipos de reportes que se pueden obtener: detallado o resumido. La Figura 3 muestra un ejemplo de un reporte detallado que produce una lista de *bad smells* encontrados, incluyendo el mensaje de error correspondiente y el número de línea. En la Figura 3 se muestra el reporte producido en el análisis estático del archivo de código: **`ConfigurationController.php`**. La Figura 4 muestra un ejemplo de

reporte resumido, que contiene una lista de todos los *bad smells* detectados en el análisis estático los cuales están ordenados por tipo.

Una vez que se ha presentado la herramienta de detección automática de *bad smells MVC*, que es la segunda contribución presentada en este trabajo, en la siguiente sección se describe la evaluación de las contribuciones presentadas.

4. Evaluación

En esta sección se presentan los resultados obtenidos al realizar dos experimentos para determinar la relevancia de las contribuciones esperadas: (i) la caracterización de *bad smells* relevantes para el estilo MVC y (ii) la detección automática de estos *bad smells* utilizando técnicas de análisis estático.

4.1. Caracterización Propuesta y Detección Automática.

Idealmente para evaluar si la caracterización de los *bad smells* era relevante, se deberían comparar los *bad smells* detectados con los contenidos en un *gold standard* como en los Sistemas de Recuperación de Información (Manning, Raghavan and Schütze 2017). Hasta donde sabemos, no existe un *gold standard* para determinar el número de *bad smells* introducidos en un sistema MVC. Por esta razón realizamos una serie de experimentos básicos en algunos sistemas existentes. Específicamente, se analizaron cinco sistemas creados por terceros e implementados con el Framework Yii para detectar los *bad smells* MVC utilizando PHP_CodeSniffer con el estándar de código Yii MVC. Los sistemas son públicos y de código abierto, y pueden ser descargados desde GitHub (GitHub 2018).

```
FILE ... protected/Controllers/ConfigurationController.php
----------------------------------------------------------------------
FOUND 6 ERRORS AFFECTING 6 LINES
----------------------------------------------------------------------
84 | ERROR | 6. Controller includes Model's computations and/or data --> Executes a
   |       | SQL statement.;
           Delete found
154| ERROR | 6. Controller includes Model's computations and/or data --> Executes a
   |       | SQL statement.;
           Count found
222| ERROR | 6. Controller includes Model's computations and/or data --> Executes a
   |       | SQL statement.;
           Delete found
244| ERROR | 6. Controller includes Model's computations and/or data --> Executes a
   |       | SQL statement.;
           Update found
264| ERROR | 6. Controller includes Model's computations and/or data --> Executes a
   |       | SQL statement.;
           Insert found
291| ERROR | 6. Controller includes Model's computations and/or data --> Executes a
   |       | SQL statement.;
           Delete found
```

Figura 3 – Un ejemplo de un reporte de errores detallado.

La Tabla 3 muestra los resultados obtenidos al ejecutar los experimentos. Se comprobó que todos los sistemas analizados tenían *bad smells MVC*. El *bad smell* MVC detectado con mayor frecuencia fue el *bad smell* número 6: El Controlador incluye cálculo y/o datos del Modelo. El único *bad smell* MVC sin ocurrencias en los análisis realizados a todos los sistemas fue el número 4: La Vista incluye cálculos y/o datos del Controlador. Aunque los tiempos de detección automática son un tanto heterogéneos, es claro que estos aumentan entre más líneas de código tenga el sistema.

```
CODE BAD SMELLS SUMMARY
------------------------------------------------
STANDARD    BAD SMELL ID    COUNT
------------------------------------------------
Yii MCV     6               91
Yii MCV     3               50
Yii MCV     3               12
Yii MCV     2               9
Yii MCV     5               6
Yii MCV     1               5
Yii MCV     2               2
Yii MCV     2               1
------------------------------------------------
A TOTAL OF 176 BAD SMELLS FOUND
------------------------------------------------
TIME: 69 MIN, 49 SEC
```

Figura 4 – Un ejemplo de un reporte de errores resumido.

Sistema	**Líneas de código**	***Bad smells* detectados**	***Bad smells* con mayor ocurrencia**	***Bad smells* sin ocurrencias**	**Tiempo**
1. Linkbooks	66, 954	176	6	4	69 min, 49 seg.
2. yii play ground	17, 341	20	6	4	10 min, 54 seg.
3. Blog Bootstrap	6, 393	15	6	4	5 min, 5 seg.
4. yii2-shop	5, 324	14	6	4	4 min, 37 seg.
5. yii-jenkis	836	1	6	1, 2, 3, 4 y 5	1.57 seg.

Tabla 3 – Resultados obtenidos utilizando PHP_CodeSniffer con el estándar de código Yii MVC.

4.2. Caracterización Propuesta y Detección Manual.

Para evaluar el grado de disminución de tiempo y esfuerzo en la detección de *bad smells* MVC usando el PHP_CodeSniffer se realizó otro experimento. El experimento consistió

en la detección manual de los *bad smells* MVC en el sistema en el cual el cual el PHP_ CodeSniffer consumió mas tiempo: Linkbooks. La detección la realizaron a un grupo de 15 desarrolladores de nivel experimentado, con conocimientos sólidos en ingeniería de software, arquitectura de software, el estilo arquitectónico MVC y el framework Yii.

Los sistemas fueron proporcionados los participantes y se les pidió que realizaran una revisión de código durante 69 minutos y 49 segundos. Esto es durante el mismo tiempo que tomó la detección automática de *bad smells* MVC en el sistema Linkbooks. Se explicó y proporcionó la caracterización propuesta de *bad smells* MVC a los desarrolladores.

Diez desarrolladores detectaron de 2 a 5 *bad smells*, tres desarrolladores encontraron de 6 a 10 *bad smells* y dos desarrolladores encontraron de 10 a 20 *bad smells*. Así, con la revisión de código realizada por 15 desarrolladores durante 69 minutos y 49 segundos se detectaron un total de 45 *bad smells* MVC. Esto en contraste con los 176 detectados automáticamente.

De este total de 44, el bad smell MVC detectado con mayor frecuencia fue el número 6: El Controlador incluye cálculo y/o datos del Modelo, con 33 ocurrencias. Los *bad smells* MVC que no fueron detectados son el número 4: La Vista incluye cálculos y/o datos del Controlador y el número 5: El Controlador incluye cálculos y/o datos de la Vista.

Habiendo discutido los resultados de estos análisis básicos, en la siguiente sección abordaremos trabajos relacionados.

5. Trabajos Relacionados

Hay dos tipos de categorías principales para trabajos relacionados: (i) catálogos de *bad smells* y (ii) las herramientas para la detección de *bad smells*. En esta sección, relacionamos nuestro trabajo con el encontrado en la literatura sobre estas dos categorías.

5.1. Catálogos de *bad smells*.

En la literatura se distinguen varios tipos de *bad smells*, que al ser organizados ha propiciado la definición de catálogos que caracterizan los *bad smells* en los siguientes tipos: (i) de código, (ii) de diseño y (iii) de arquitectura. A continuación describimos cada uno de estos.

Los *bad smells* de código decriben problemas originados por malas prácticas de programación. Existen catálogos que definen bad smells de código; entre los más relevantes encontramos (SourceMaking: Code Smells 2018).

Los *bad smells* de diseño se producen generalmente por violaciones a principios o conceptos del diseño orientado a objetos. Existen catálogos que definen bad smells de diseño como por ejemplo (Ganesh, Sharma, and Suryanarayana 2013) y (Lippert and Roock 2006).

La arquitectura de un sistema es un término que se utiliza para referirse a un modelo de alto nivel que describe su estructuración general en términos de componentes y relaciones (también llamados conectores) entre ellos y las propiedades de ambos (Bass,

Clements, and Kazman 2012). Los *bad smells* de arquitectura se refieren a soluciones no apropiadas al contexto o uso inapropiado de abstracciones de diseño arquitectónico.

A pesar de la gran variedad de trabajos que analizan el impacto de los *bad smells* en arquitecturas de software, se han propuesto muy pocos catálogos de *bad smells* arquitectónicos. Uno de ellos se propone en (García et al. 2009) e incluye cuatro *bad smells* arquitectónicos mencionados a continuación: *connector envy, scattered parasitic functionality, ambiguous interfaces, y extraneous adjacent connector*. En (Vale et al. 2014) los autores realizaron una revisión sistemática para caracterizar los *bad smells* arquitectónicos.. Los autores reportaron un conjunto de 14 *bad smells* arquitectónicos, que además de los 4 *bad smells* arquitectónicos reportados en (García et al. 2009) y en versiones específicas de SLP (por ejemplo, *connector envy SPL*), se encontraron otros como: *component concern overload, cyclic dependency, overused interface, redundant interface, unwanted dependencies y feature concentration*. Además de los pocos catálogos que consideran los *bad smell* a nivel arquitectónico, ninguno de ellos considera los *bad smells* dentro del contexto de un estilo arquitectónico, en contraste con nuestro trabajo.

5.2. Herramientas para la detección de bad smells.

Existen herramientas que permiten la detección automática de algunos de los *bad smells* mencionados anteriormente y otras herramientas que proporcionan una vista de la estructura del sistema para encontrarlos manualmente.

En (Fernandes et al. 2016) los autores presentan los hallazgos de una revisión sistemática en la literatura con un total de 84 herramientas para la detección de *bad smells*. Entre otras observaciones los autores reportan que las herramientas existentes para analizar el código, se concentran en 3 lenguajes de programación, como Java, C, y C++. Encontraron 4 herramientas para el análisis de sistemas de código desarrollados en PHP. Con respecto a las estrategias de detección, los autores descubrieron que la mayoría de la herramientas están basadas en métricas. Finalmente, los autores encontraron 61 *bad smells* diferentes detectables por herramientas, incluyendo Fowler's [1] así como otras encontradas en (Bavota et al. 2015), (Khomh et al. 2011), (Maiga et al. 2012) y (Vidal, Marcos, and Díaz-Pace 2014). A diferencia de la mayoría de estas herramientas descritas, la herramienta presentada en este trabajo puede detectar *bad smells* en sistemas implementados en PHP. Además, los *bad smells* detectados por nuestra herramienta son arquitectónicos y relevantes para el estilo arquitectónico MVC.

6. Conclusiones y Trabajo Futuro

En este trabajo, presentamos nuestro progreso en la caracterización (i) de un conjunto de *bad smells* que son relevantes para el estilo arquitectónico MVC, así como (ii) la detección automática de estos *bad smells* utilizando técnicas de análisis estático en código de sistemas implementados con el Framework Yii.

La realización de este trabajo nos permite afirmar que las herramientas de detección de *bad smells* arquitectónicos son ciertamente útiles para evaluar qué partes del código

deben mejorarse para respetar las restricciones impuestas por el estilo arquitectónico utilizado. Es posible concluir que es se pueden desarrollar herramientas de detección de *bad smells* arquitectónicos usando análisis estático y estándares de código definidos para este propósito.

Los resultados obtenidos en nuestros experimentos en la detección automática muestran que la mayoría de los *bad smells* MVC definidos existen en la práctica. Además, aunque los tiempos de detección automática son un tanto heterogéneos, son mejores en comparación con los requeridos en una revisión de código realizada de forma manual. Para el experimento realizado con una revisión de código, involucrando a 15 desarrolladores, se logró detectar solo el 25.5 % de los *bad smells* MVC que detectaron usando la herramienta PHP_CodeSniffer. A nivel individual el desarrolladores que rindió mejor detectó 20 *bad smells* MVC, lo que representa solo el 11.3 % de los que detectaron de forma automática.

Aunque este trabajo continúa en progreso, consideramos que los resultados obtenidos hasta el momento son importantes no solo en el área de investigación sino también para el área desarrollo, donde podría ser de gran utilidad el uso de los artefactos generdos.

Como posibles líneas trabajo a futuro, tenemos planeado investigar aspectos relacionados, incluyendo la identificación y caracterización de otros *bad smells* que pueden ocurrir en arquitecturas MVC, la detección de *bad smells* en otros lenguajes de programación, y la mejora continua de la herramienta. En los siguiente párrafos abundamos sobre estas líneas.

Consideramos que es posible agregar a nuestro conjunto inicial de *bad smells* arquitectónicos MVC otros *bad smells* que no se están documentados en la literatura. Nosotros visualizamos la clasificación de *bad smells* MVC basada considerando dimensiones, como por ejemplo el comportamiento y la estructura (Ganesh, Sharma, and Suryanarayana 2013). Todos los *bad smells* detectados por usando el enfoque descrito en este trabajo consideran solo la una dimensión de comportamiento.

Como se describe en (Fernandes et al. 2016) la mayoría de las herramientas que permiten detectar *bad smells*, están restringidas para solo detectar *bad smells* en un lenguaje de programación específico, lo cual se convierte en una limitación significativa para cada una de las herramientas que permiten detectar *bad smells*. Por otro lado, existen técnicas de *machine learning* que son métodos computacionales que usan la "experiencia" para obtener una mejor precisión en la predicción. Consideramos que la aplicación de técnicas de *machine learning* para la detección de *bad smells* arquitectónicos puede favorecer el desempeño y la precisión, así como soportar más de un lenguaje de programación, siempre y cuando se tengan conjuntos de ejemplos para el entrenamiento.

Finalmente, con respecto a la mejora del soporte de herramientas, tenemos la intención de explorar la identificación de posibles refactorizaciones del código para corregir los *bad smells* identificados. Además, también sería útil que los *bad smells* detectados se pudieran mostrar de manera visual. Consideramos que esto contribuiría a que programadores novatos pudieran comprender de forma más fácil por qué no se respeta las restricciones del estilo.

Referencias

Bass, L., Clements, P., & Kazman, R. (2012). *Software Architecture in Practice*. Boston, MA, USA: Addison-Wesley Professional.

Bavota, G. (2015). An Experimental Investigation on the Innate Relationship between Quality and Refactoring. *Journal of Systems and Software,* 107, 1–14.

SourceMaking. (2018). *SourceMaking: Code Smells 2018*. Retrieved from: https://sourcemaking.com/refactoring/badsmellsincode

Yiiframework. (2017). Best MVC Practices. Retrieved from: http://www.yiiframework.com/doc/guide/1.1/en/basics.best-practices.

Brown, N. (2010). Managing Technical Debt in Software Reliant Systems. In Proceedings of *Workshop on Future of Software Engineering Research, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, (pp. 47-52). Santa Fe, NM, USA : ACM.

Django Software Foundation. (2018). *Django*. Retrieved from: https://www.djangoproject.com.

Espejo Chavarría, A., Bayona Oré, S., & Pastor, C. (2016). Aseguramiento de la calidad en el proceso de desarrollo de software utilizando CMMI, TSP y PSP. RISTI - *Revista Ibérica de Sistemas y Tecnologías de Información*, 20(12), 62–77.

Fernandes, E. (2016). A Review-Based Comparative Study of Bad Smell Detection Tools. In Proceedings of *20th International Conference on Evaluation and Assessment in Software Engineering (pp. 109-20)*. New York: ACM.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley.

Gamma, E. (1995). *Design patterns: Elements of reusable object-oriented software*. Reading, MA, USA: Addison-Wesley.

Ganesh, S. G., Sharma, T., & Suryanarayana, G. (2013). Towards a Principle-Based Classification of Structural Design Smells. *Journal of Object Technology* 12(2): 1–29.

García, J., Popescu, D., Edwards, G., & Medvidovic, N. (2009). Toward a Catalogue of Architectural Bad Smells. In Proceedings of *5th International Conference on the Quality of Software Architectures (pp.* 146–62). Berlin, Heidelberg: Springer-Verlag.

GitHub. (2018). *GitHub*. Retrieved from: https://github.com/

GNOME developer. (2018). GNOME Programming Guidelines. Retrieved from: https://developer.gnome.org/programming-guidelines/stable/

Khomh, F., Vaucher, S., Guéhéneuc, Y., & Sahraoui, H. (2011). BDTEX: A GQM-Based Bayesian Approach for the Detection of Antipatterns. *Journal of Systems and Software* 84: 559–72.

Laravel. (2018). *Laravel*. Retrieved from: https://laravel.com.

Lippert, M., & Roock, S. (2006). *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Wiley.

Maiga, A. (2012). Support Vector Machines for Anti-Pattern Detection. In Proceedings of *27th International Conference on Automated Software Engineering*, (pp. 278-81). New York: IEEE Press.

Manning, C. D., Raghavan, P., & Schütze, H. (2017). *Introduction to information retrieval*. Delhi: Cambridge University Press.

Morales Huanca, L., & Bayona Oré, S. (2017). Factores que afectan la precisión de la estimación del esfuerzo en proyectos de software usando puntos de caso de uso. *RISTI - Revista Ibérica de Sistemas y Tecnologías de Información*, *21*(03), 18–32.

PEAR. (2018). *PEAR - PHP Extension and Application Repository*. Retrieved from: http://pear.php.net/

PEAR. (2018). *PHP_CodeSniffer*. Retreived from: https://pear.php.net/package/PHP_CodeSniffer.

PHP. (2018). *PHP: Hypertext Processor*. Retrieved from: https://secure.php.net/

Rails. (2018). Rails. Retrieved from: http://rubyonrails.org

Source Makings. (2017). *Source Makings, Code Smells*. Retreived from: https://sourcemaking.com/refactoring/smells.

Spring. (2018). *Spring*. Retrieved from: https://spring.io.

Vale, G., Figueiredo, E., Abílio, R., & Costa, H. (2014). Bad Smells in Software Product Lines: A Systematic Review. In Proceedings of *Eighth Brazilian Symposium on Software Components, Architectures and Reuse (pp*. 84–94). Washington, DC, USA: IEEE Computer Society.

Vidal, S., Marcos, C., & Díaz-Pace, J. (2014). An Approach to Prioritize Code Smells for Refactoring. *Automated Software Engineering* 23: 501–32.

YiiFramework. (2017). *YiiFramework*. Retreieved from: http://www.yiiframework.com/.