Towards Detecting MVC Architectural Smells

Perla Velasco-Elizondo [1], Lucero Castañeda-Calvillo [2], Alejandro García-Fernandez [3] and Sodel Vazquez-Reyes [1]

[1] Autonomous University of Zacatecas, Zacatecas, ZAC, 98000, Mexico.
[2] Centre for Computing Research, Mexico City, 07738, Mexico.
[3] Centre for Mathematical Research, Zacatecas, ZAC, 98060, Mexico.

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

Abstract. The term "bad smell" denotes a symptom of poor design or implementation that negatively impacts a software system's properties. The research community has been actively identifying the characteristics of bad smells bad smells as well as developing approaches for detecting and fixing them. However, most of these efforts focus on smells that occur at code level: little consideration is given to smells that occur at higher levels of abstraction. This paper presents an initial effort to fill this gap by contributing to (*i*) the characterization of bad smells that are relevant to the Model-View-Controller architectural style and (*ii*) assessing the feasibility of their automatic detection using text analysis techniques in five systems, implemented with the Yii Framework. The obtained results show that the defined smells exist in practice and give some insight into which of them tend to occur more frequently. Regarding the automatic detection method, results show that it exhibits good performance and accuracy.

Keywords: Software Architecture, Bad Smells, static analysis, text analysis, MVC, Yii.

1 Introduction

In Software Engineering the term *bad smell*, hereafter referred to as "smell", is used to denote a symptom of poor design or implementation that negatively impacts a software system's properties (e.g., maintainability, testability, reusability.) [1]. A smell is usually used to indicate a potential problem with software. Although not universally agreed upon, it is generally accepted that smells can occur at different levels of abstraction going from source code (e.g., long parameter list [2]) to architecture (e.g., connector envy [3]).

Smells are a common factor in the accumulation of *technical debt* [4]. Thus, detecting and fixing them becomes relevant to software system development. In recent years, the research community has been actively characterising smells (e.g. [5]) as well as developing approaches and tools for detecting and fixing them (see [6]). However, most of these efforts focus on smells that occur at lower levels of abstraction and few of them characterize, identify and fix smells at the architectural level. Additionally, in these works little consideration is given to performing these activities within the context of

J. Mejia et al. (eds.), *Trends and Applications in Software Engineering*,
Advances in Intelligent Systems and Computing 688,
https://doi.org/10.1007/978-3-319-69341-5_23

architectural styles, which are design structures commonly used for software development.

The work presented in this paper is an initial effort to fill this gap by contributing to (*i*) the characterisation of a set of smells that are relevant to the Model-View-Controller (MVC) architectural style [7], which has been widely adopted for Web systems in major programming frameworks, and (*ii*) assessing the feasibility of the automatic detection of these smells in five systems implemented with Yii Framework [8] by using text analysis techniques. The obtained results show that the defined smells exist in practice and give some insight on which of them tend to occur more. Regarding the automatic detection method, results show that it exhibits good performance and accuracy.

The remainder of this paper is organized as follows. In section 2 the MVC architectural style is explained and, based on its general constraints, a characterization of related smells is defined. Next, in Section 3, an approach to the automatic detection of these smells is explained. A basic evaluation of the approach is presented in Section 4. In Section 5 a discussion of related work is presented. Finally, in Section 6, the conclusions of this work are stated as well as some lines of future work.

2 MVC Architectural Style and Related Smells

Software architecture provides a high-level model of a system in terms of components and, connectors, and properties of both each [9]. While it is possible to specify the architecture of a system using this generic vocabulary, it is better to adopt a more specialized one vocabulary when targeting architectures of a particular application domain. This specialized modelling vocabulary is known as an architectural style [9].

The MVC architectural style has been widely adopted as an architecture for the design and implementation of Web systems. Today, many popular development frameworks allow for the construction of Web systems using this style, e.g., Spring [10], Django [11], Rails [12], Laravel [13], and Yii [8]. Successful use of this style isolates business from presentation logic, which results in a system that is easier to test and maintain. Figure 1 shows a graphical representation the MVC architectural style and Table 1 describes the elements in this representation.

The MVC style defines the following general constraints:

- The Model should not deal directly with processing end-user requests. For example, its implementation should not contain $_GET, $_POST variables.
- The Model should not deal directly with the presentation of data for end-user requests. For example, its implementation should not contain HTML presentational code.

- The View should not deal directly with performing explicit access to system data. For example, its implementation should not contain code for DB queries.
- The View should not deal directly with end-user requests. For example, its implementation should not contain $_GET, $_POST variables.
- The Controller should not deal directly with performing explicit access to system data. For example its implementation should not contain code for DB queries.
- The Controller should not deal directly with the presentation of data for end-user requests. For example, its implementation should not contain HTML presentational code.

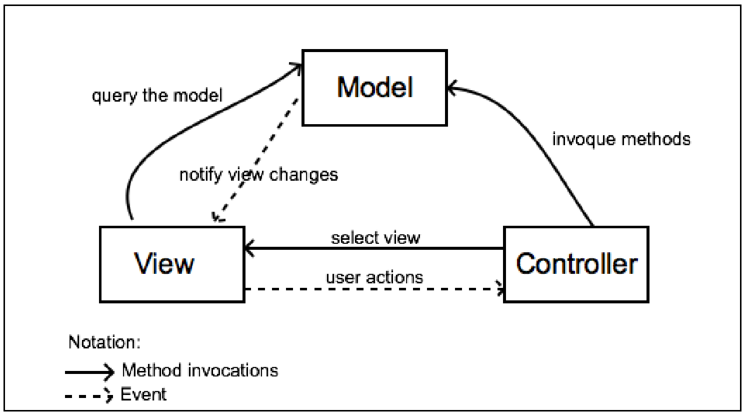

Fig. 1. Graphical representation of the MVC architectural style

2.1 MVC Smells

The concept of *architectural smell* was originally used in [14] to describe an indication of an underlying problem that occurs at a higher level of a system's abstraction than code. Causes of architectural smells include, amongst others, applying a design solution in an inappropriate context, mixing combinations of design abstractions, or applying design abstractions at the wrong level of granularity [14].

Although nearly every Web developer knows the MVC style, properly implementing it still eludes many [7]. Frequently, the general constraints, as defined in the previous section, are not respected, resulting in poor design or implementation decisions that we call *MVC Architectural Smells*. Based on these constraints, Table 2 describes a categorisation of smells relevant to the MVC architectural style.

We should note that this categorization of smells is not our own invention, but is a compilation of elements drawn from several informal sources (e.g. developer blogs, question and answer sites), which we have assembled here in a more comprehensive and consolidated manner.

Table 1. Description of the elements of the MVC architectural style

Element	Description
Model	Represents the system's underlying data and the business rules that govern data access. It notifies the View of any changes made in the data.
View	A representation of the Model in a format desired by the end users. It queries the Model for any changes made in the data.
Controller	An intermediary between the View and the Model. It receives end users' actions, commands requests coming from the View, invokes the required methods in the Model, and changes the View's presentation of the Model when necessary.

Table 2. Categorisation of smells relevant to the MVC architectural style

ID	Name	Description
1.	Model includes View's computations and/or data	Happens when the Model contains presentation of data of end-user requests (e.g. HTML code).
2.	Model includes Controller's computations and/or data	Happens when the Model has direct access to variables that represent the end-user's request (i.e. direct access to $_GET, $_POST variables).
3.	View includes Model's computations and/or data	Happens when the Controller has domain logic (e.g. code of DB queries).
4.	View includes Controller's computations and/or data	Happens when the View has direct access to variables that represent the end-user's request (i.e. direct access to $_GET, $_POST variables.).
5.	Controller includes View's computations and/or data	Happens when the Model contains presentation of data of end-user requests (e.g. HTML code).
6.	Controller includes Model's computations and/or data	Happens when the Controller has domain logic (e.g. code of DB queries).

3 MVC Code Sniffer

PHP_CodeSniffer is a static analysis tool that "sniffs" PHP code files to detect violations of a given set of rules defined in a *coding standard* [15]. It works by tokenising the contents of a code file into building blocks. These are then validated through the use of text analysis to check a variety of aspects against the coding standard in question. In this context, a coding standard can be seen as a set of conventions regulating how code must be written. These conventions often include formatting, naming, and common idioms. Multiple coding standards can be used within PHP_CodeSniffer. After the analysis process, PHP_CodeSniffer outputs a list of violations found, with corresponding error messages and line numbers.

3.1 MVC Sniff files

A coding standard in PHP_CodeSniffer consists of a collection of open-source *sniff files*. Each sniff file checks one convention of the coding standard and can be coded in PHP, JavaScript, or CSS. Thus, it is possible to create new coding standards by reusing, extending, and building new sniff files.

To detect MVC Architectural Smells, we built a *Yii MVC code standard*. The sniff files in this code standard are PHP classes, which are used in six smell detection algorithms that allow sniffing for the smells defined in Table 2.[1] Figure 2 shows an excerpt from the code of a sniff file. This sniff file, in combination with others, is utilised to detect smell number 6, which is *Controller includes Model's computations and/or data*. As previously explained, this smell is related to an issue with a Controller performing the responsibilities of a Model. Thus, in terms of code, this smell results in a Model including code to read, write, or update data or data stores, typically in a database.

As shown in Figure 2, a sniff class must implement the **`PHP_CodeSniffer_Sniff`** interface. This interface declares two functions that are needed for code analysis: the **`register`** and **`process`** functions. The **`register`** function allows a sniff to retrieve the types of token that it will process, in this case strings. Once these tokens are available, the **`process`** function is called with a representation of the code file being checked and the position in the stack where the token was found: the **`PHP_CodeSniffer_File`** and **`$stackPtr`** parameters, respectively.

Information about a token can be found through a call to the **`getTokens`** method on the code file being checked (line 13). This method then returns an array of tokens, which is indexed by the position of the token in the token stack. Tokens have a **`content`** index in the array, consisting of the content of the token as it appears in the code.

The implemented analysis detects the **`delete`** statement (line 14) in the code of a Controller file. This statement allows for the deletion of existing records in a database. If the **`delete`** statement is discovered in the code, the corresponding smell detection message is triggered (lines 17-20). A sniff indicates that an error has occurred by calling the **`addError method,`** which generates the created error message as the first argument, and the position in the stack where the occurrence was detected as the second

[1] The code standard can be downloaded from https://github.com/Lucerin/Yii.

argument, including code to uniquely identify the type of error within this sniff and an array of data used inside the error message.

```
 ...
class Yii_Sniffs_SmellsArchitectureMVC_SA6_DeleteSniff implements
PHP_CodeSniffer_Sniff {
   public function register() {
      return array(T_STRING);
   }

   public function process(PHP_CodeSniffer_File $phpcsFile, $stackPtr){
      $name = $phpcsFile->getFilename();
      $tokens = $phpcsFile->getTokens();
      if ($tokens[$stackPtr]['content'] === 'delete') {
         ...
         if (strpos($name, "controllers")) {
            $error = '6. Controller includes Model's computations and/or
                     data --> Executes a SQL statement.;
                     Delete found';
            $data  = array(trim($tokens[$stackPtr]['content']));
            $phpcsFile->addError($error, $stackPtr, 'Found', $data);
        }
     }
  }
...
```

Fig. 2. Excerpt of a sniff file for smell number 6: Controller includes Model's computations and/or data

When PHP_CodeSniffer is run on a Yii project using the Yii MVC code standard, there are two possible report types that can be obtained: detailed or summary. Figure 3 shows an example of a detailed report that produces a list of smells found, including corresponding error messages and line numbers. This figure shows the report produced from the analysis of a single code file: **`ConfigurationController.php`** Figure 4 illustrates a sample summary report, which gives the total of all detected smells arranged by type.

4 Evaluation

In order to assess whether the definition of the Architectural Smells was appropriate, we wanted to compare them to a "*gold standard*" (as in Information Retrieval Systems). As far as we know, there is not a gold standard for determining the number of smells introduced in a MVC system. For this reason we ran a set of basic experiments on existing systems. Specifically, five systems created by third parties and implemented with the Yii Framework were analysed to detect MVC Architectural Smells using the PHP_CodeSniffer with the defined Yii MVC code standard. The systems are public and open-source, and can be downloaded from github.

Table 3 shows the results obtained from running the experiments. Each of the analysed systems had smells. It is notable that the architectural smell that occurred most frequently was smell number 6: *Controller includes Model's computations and/or data.* The only smell with no occurrences in all analysed systems was smell number 4: *View includes Controller's computations and/or data*.

```
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
```

Fig. 3. An example of the errors reported in the detailed report

```
CODE SMELLS SUMMARY
---------------------------------------------------
STANDARD     SMELL ID     SNIFF             COUNT
---------------------------------------------------
Yii MCV      6            Count found        91
Yii MCV      6            Delete found       50
Yii MCV      1            <div/> found       12
Yii MCV      6            Insert found        9
Yii MCV      6            Update found        6
Yii MCV      6            Select found        5
Yii MCV      6            Set found           2
Yii MCV      6            Reset found         1
---------------------------------------------------
A TOTAL OF 176 SMELLS FOUND
---------------------------------------------------
TIME: 69 MIN, 49 SEC
```

Fig. 4. Example of the summary report showing detected errors

For most of the systems, the time required for analysis was reasonably minimal. The exception was the analysis of system 1, which took 69 min, 49 sec. It should be noted, however, that the number of smells that this system had was significantly higher compared to the other systems and we believe that the time required for automatic detection is nonetheless a significant improvement compared to the complexity and challenge of detecting these smells manually.

Table 3. Results obtained by using PHP_CodeSniffer with the Yii MVC code standard.

System	LOC	Detected Smells	Most Occurring Smell	Smell with no Occurrences	Analysis Time
1. Linkbooks	66, 954	176	6	4	69 min, 49 sec
2. yii play ground	17, 341	20	6	4	10 min, 54 sec
3. Blog Bootstrap	6, 393	15	6	4	5 min, 5 sec
4. yii2-shop	5, 324	14	6	4	4 min, 37 sec
5. yii-jenkis	836	1	6	1, 2, 3, 4 and 5	1.57 sec

Having discussed the results of this basic analysis, in the next section we will cover areas of related work.

5 Related Work

There are two primary categories of related work: (*i*) code smell catalogues and (*ii*) code smell detection approaches. In this section, we relate our work to other literature on these two categories.

Code smell catalogues. Despite the range of works discussing the impact of bad smells in software architecture, very few catalogues of architectural smells have been proposed. One of these is proposed in [3] and includes four architectural smells, namely, *connector envy*, *scattered parasitic functionality*, *ambiguous interfaces*, and *extraneous adjacent connector*. In [16] the authors conducted a systematic review so as to to characterize architectural smells in the context of product lines. The authors reported a set of 14 architectural smells which, in addition to the 4 smells in [3] and SLP-specific versions of those smells (e.g. *connector envy SPL*), includes *component concern overload*, *cyclic dependency*, *overused interface*, *redundant interface*, *unwanted dependencies* and *feature concentration*. Additionally, of the few catalogues that do consider smells at the architectural level, none of these consider smells within the context of an architectural style, in sharp contrast to our work.

Code smell detection tools. In [6] the authors present the findings of a systematic literature review of 84 bad smell detection tools. Among other observations, the authors report that existing tools for analysing code concentrate on 3 main languages, namely, Java, C, and C++. They found only 4 tools for the analysis of systems coded in PHP. With regard to detection strategies, the authors discovered that most of the tools are metric-based. Less frequently used detection strategies include tree-based, text analysis, program dependence graph, machine learning, and logic meta-programming. Finally, the authors found 61 different bad smells detectable by tools, including Fowler's [1] as well as others discussed in sources such as [17], [18], [19] and [20]. In contrast with most of these described tools, the tool presented in this paper can detect smells in systems implemented in PHP. Furthermore, the smells detected by our tool are architectural and relevant to the MVC architectural style.

6 Conclusions and Future Work

In this paper, we presented our progress toward (*i*) characterising a set of smells that are relevant to the MVC architectural style, as well as (*ii*) assessing the feasibility of automatically detecting these smells. The results obtained from our experiments in automatic detection show that most of the characterised smells do exist in practice. Our use of text analysis for the purposes of smell detection is a primary focus of this paper, and experiments implementing this technique demonstrated effective, accurate results.

Although the material presented is this paper is still a work in progress, we believe that our preliminary results are valuable not only to researchers but also to developers, who may wish to begin using PHP_CodeSniffer with the defined Yii MVC code standard.

In our future work, we plan to investigate related aspects, including the identification and categorization of other smells that may occur in MVC architectures, smell detection in other coding languages, and the enhancement of tool support.

We believe that it is possible to add to our initial set of MVC Architectural Smells by incorporating smells that are, as yet, undocumented in literature. We envision a classification for MVC Architectural Smells based on dimension, such as behaviour and structure [5]. All smells detected by the technique detailed in this paper belong to the behavioural dimension.

As described in [6] most detection tools are restricted to detecting smells in specific programming languages, which is a significant limitation of existing smell detection tools. By contrast, machine learning techniques are computational methods that use "experience" to make accurate predictions. We believe that the application of machine learning techniques to the detection of architectural smells can provide performance and accuracy, as well as multi-language support, while only requiring a few sets of training examples.

Finally, with regard to enhancing tool support, we intend to explore the identification of refactoring for identified smells.

References

1. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999).
2. Source Makings, Code Smells, https://sourcemaking.com/refactoring/smells
3. Garcia, J., Popescu, D., Edwards, G., Medvidovic, N.: Toward a catalogue of architectural bad smells. In 5th International Conference on the Quality of Software Architectures, pp. 146—162. Springer-Verlag, Berlin, Heidelberg (2009).
4. Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R. L., Ozkaya, I., Sangwan, R. S., Seaman, C.B., Sullivan, K.J., Zazworka, N.: Managing technical debt in software reliant systems. In Workshop on Future of Software Engineering Research, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 47--52. ACM (2010).
5. Ganesh, S.G., Sharma, T., Suryanarayana, G.: Towards a Principle-based Classification of Structural Design Smells. Journal of Object Technology, 12 (2), 1-29 (2013).
6. Fernandes, E., Oliveira, J., Vale, G. Paiva, T., Figueiredo, E.: A review-based comparative study of bad smell detection tools. In 20th International Conference on Evaluation and Assessment in Software Engineering, pp. 109--120. ACM, New York (2016).
7. Best MVC Practices, http://www.yiiframework.com/doc/guide/1.1/en/basics.best-practices
8. yiiFramework, http://www.yiiframework.com
9. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley Professional (2012).
10. Spring, https://spring.io
11. Django, https://www.djangoproject.com
12. Rails, http://rubyonrails.org
13. Laravel, https://laravel.com
14. Lippert, M., Roock, S.: Refactoring in Large Software Projects: Performing Complex Restructurings Successfully. Wiley (2006).
15. PHP_CodeSniffer, https://pear.php.net/package/PHP_CodeSniffer
16. Vale, G., Figueiredo, E., Abílio, R., Costa, H.: Bad Smells in Software Product Lines: A Systematic Review. In *Eighth Brazilian Symposium on Software Components, Architectures and Reuse, pp.* 84--94. IEEE Computer Society, Washington, DC (2014).
17. Bavota, G., De Lucia, A., Di Penta, M., Oliveto, R., Palomba, F.: An Experimental Investigation on the Innate Relationship between Quality and Refactoring. Journal of Systems and Software, 107, 1--14 (2015).
18. Khomh, F., Vaucher, S., Guéhéneuc, Y., Sahraoui, H.: BDTEX: A GQM-based Bayesian Approach for the Detection of Antipatterns. Journal of Systems and Software, 84, 559--572 (2011).
19. Maiga, A., Ali, N., Bhattacharya, N., Sabané, A., Guéhéneuc, Y. G., Antoniol, G., Aimeur, E.: Support Vector Machines for Anti-pattern Detection. In 27th International Conference on Automated Software Engineering, pp. 278--281. IEEE Press, New York (2012).
20. Vidal, S., Marcos, C., Díaz-Pace, J.: An Approach to Prioritize Code Smells for Refactoring. Automated Software Engineering, 23, 501--532 (2014).