Evolution of method invocation and object instantiation patterns in a PHP ecosystem

Panos Kyriakakis[1], Alexander Chatzigeorgiou[1], Apostolos Ampatzoglou[2] and Stelios Xinogalos[1]

[1] Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece
pkyriakakis@uom.edu.gr {achat, stelios}@uom.gr

[2] Department of Mathematics and Computer Science, University of Groningen, Groningen, Netherlands
a.ampatzoglou@rug.nl

ABSTRACT

PHP is one of the most frequently used scripting languages for server-side programming, since approximately 75% of successful web applications have been developed with PHP. The main benefits of PHP are its low learning curve and the rich variety of dynamic features that it offers. These benefits have contributed towards the development of a large community of programmers around PHP, which in turn created a vast ecosystem of applications and frameworks. In this study we have empirically investigated ten famous PHP frameworks / applications and over 240 MLOC in order to explore their internal structure. More specifically, we present some demographics on method invocation and object instantiation patterns, empowered by the dynamic nature of the PHP language. To present the results we employ statistical methods inspired by ecology. In particular, we explore the diversity and dominance of these patterns, by using the Shannon-Wiener diversity index and a Dominance index that has been originally developed for Plankton. The main conclusion of our study is that the employment of the patterns, is related to developers, and therefore we can observe normality and repetition with small diversions.

Categories and Subject Descriptors

Software and its engineering → Software creation and management → Software post-development issues: Software reverse engineering, software evolution, maintaining software

General Terms

Design, Languages, Experimentation

Keywords

PHP, scripting languages, software maintenance, method invocation, object creation

1. INTRODUCTION

Scripting languages constitute the backbone of web applications. Maturing over the years these languages have offered developers a variety of tools for building fast and solid applications of any size. Among these languages, PHP is holding the lead in web application development for over than a decade. New language features have been introduced and excellent frameworks have been developed, rendering PHP an appealing, enterprise scale programming language. An important characteristic of PHP is the provision of dynamic features along with the weakly typed variables, which allow developers to implement elegant solutions, at the same time promoting modularity and extendibility.

In this study we focus on some of the aforementioned dynamic features of PHP that can be used for method invocations and object instantiations. In contrast to compiled languages, PHP offers developers the ease of *variable variables*[1], which allows the object name or the method name in method invocations or the class name in object instantiations to be a string variable. The primary goal of this study is to investigate the landscape of different method invocation and object instantiation approaches, which we collectively refer to as *patterns*. To this end, we analyzed a corpus of large and well known PHP projects in order to depict the frequency of those patterns from the perspective of static analysis. By understanding those patterns we attempt to shed light into the developers' habits with respect to the exploitation of the language's dynamic features, and answer our main research question, i.e., "*Do developers employ a minimal set of patterns, narrowing themselves to the most common ones, or do they try to harvest as many features of the language as possible?*". To answer this question we borrow data analysis techniques from ecology, by considering the set of analyzed PHP applications as an ecosystem, where the examined patterns are treated as species.

The most frequently employed patterns can be used as a guide to inexperienced developers or even assist educators to target the most efficient ways for invoking methods and instantiating objects. We are currently conducting further research to investigate the potential benefit from the use of such patterns, but initial evidence suggests that dynamic features offer improved maintainability and extensibility.

The rest of the paper is organized as follows: in Section 2 we present the ecosystem, while in Section 3 the data collection and analysis processes in the study. In Section 4, we present and discuss the results of the study. In Section 5 we list threats to validity and mitigation actions, whereas related work is presented in Section 6. Finally, the paper is concluded in Section 7.

2. THE ECOSYSTEM

In this section we describe the ecosystem of our empirical study, by first presenting the projects in the corpus of the ecosystem and then the patterns that form its species. To select a project for inclusion in our study, the project should: (a) be open source; (b) have its source code available in GitHub; (c) the majority of its source code should be written in PHP; (d) be a full blown application, and not a library; (e) have more than 20 releases available in GitHub; (f) come from distinct (as much as possible) application

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
PCI '16, November 10-12, 2016, Patras, Greece
© 2016 ACM. ISBN 978-1-4503-4789-1/16/11…$15.00
DOI: http://dx.doi.org/10.1145/3003733.3003777

[1] http://php.net/manual/en/language.variables.variable.php

domain; (g) have a large community around it; and (h) have an established reputation in its domain.

By applying the aforementioned criteria we have selected ten open source projects. Although the matching list of projects was larger, we had to limit our study to ten projects, due to the processing time required for the analysis and the limited computational resources. The list of analyzed projects is presented in Table 1. These projects are influencing the evolution not only of PHP applications, but the entire web application development industry. Cumulatively, we have analyzed more than 1,000 versions and over than 240 MLOC of PHP source code.

Table 1. Analyzed Projects

Project	Business Domain	First Version	Last Version	Cumulative	
				# of Versions	kLOC
WordPress	Blog	1.5	3.6.1	71	6,914
Drupal	CMS	4.0.0	7.23	120	8,321
phpBB	Forum	2.0.0	3.0.12	37	2,815
MantisBt	Bug tracking	1.0.0	1.2.15	33	4,124
phpMyAdmin	Admin tool	2.9.0	4.1.6	129	13,985
PrestaShop	e-commerce	1.5.0.0	1.6.0.10	29	7,248
Typo3	CMS/CMF	3.6.1	6.2.6	189	56,802
Joomla	CMS/CMF	1.7.3	3.3.6	59	17,705
Moodle	e-learning	1.0.0	2.8.1	165	91,412
MediaWiki	wiki	1.1.0	1.24.1	200	33,567

2.1 Method Invocation Patterns

In theory, methods are invoked using the object member access operator, which might be the same for static member access and object access (i.e. Java and Ruby) or in some languages two distinct operators. In the case of PHP there is a separate operator for each case, namely the object operator (`->`) for method access and the double colon (`::`) operator for static access. The operands are, on the left side an object or a class (which we call the object part), and on the right side the method (which we call the method part). In Table 2 the typical invocation patterns are shown for each operator.

Table 2. Trivial invocation patterns

Type	Example	Object Part	Method Part
Object	`$myPerson->myMethod();`	A variable holding a reference to an object	Explicit name of the method
Static	`Person::getTypes();`	The name of the class	

A dynamic aspect in PHP is introduced by the *variable variable* construct and the *curly braces syntax*[2], offering to the developer the possibility to refer to a variable using a string (literal or variable) containing its name. For example:

```
//assigns an Address object
$addressOfHome = new Address();
//assigns an Address object also
$theirWorkAddress = new Address();
//assigns an Address object too
$theAddressOfVacation = new Address();
//… code to assign values to object properties
// iterate address objects
$objNames = ['addressOfHome', 'theirWorkAddress',
'theAddressOfVacation'];
```

[2] http://php.net/manual/en/language.types.string.php

```
foreach($objNames as $varname) {
  echo $$varname->getCity();
}
```

In this case, address objects could have been stored in a data structure, as well. But there are cases where alternative approaches for accessing object methods can prove to be much more efficient in terms of size and extensibility. In this example a simple approach to iterate over the address objects and print the city property of each address would be to store their names in an array, iterate over their names and use the *variable variable* construct to access the objects. The *curly braces* syntax can be used to construct the string containing either the object or the method part. The next code snippet is a variation of the previous one, where the names of the variables holding the object reference are following a motif, prefixed with ‘address’ and followed by the address type. This allows iterating address types and constructing the variable name using the curly braces syntax.

```
$addressHome = new Address();
$addressWork = new Address();
$addressVacation = new Address();
//… code to assign values to object properties
foreach(['Home', 'Work', 'Vacation'] as $type) {
  echo ${"address$type"}->getCity();
}
```

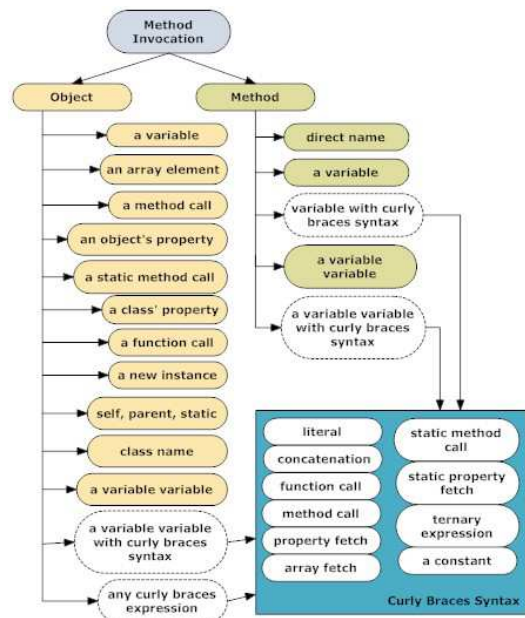

Figure 1. Breakdown of method invocation patterns

Additionally, as in many scripting languages, other approaches are also available to the developer. PHP offers a set of function handling functions[3] that can be used to call functions or methods using a *callable*[4]. A callable can be either a string containing the function to be invoked or an array with two elements, where the first contains the class name or an object and the second the name of the method (string variable or literal). For example the following line of code performs a static invocation of method `getBalance()` to the class `Client`. Finally, as in many OO lan-

[3] http://php.net/manual/en/ref.funchand.php

[4] http://php.net/manual/en/language.types.callable.php

guages, a reflection library is also available and its employment introduces alternative approaches for method invocation.

```
call_user_func(array('Client', 'getBalance'));
```

In Figure 1 the alternatives for method invocations are graphically depicted. On the left side, we present the theoretically possible patterns for designating the object, whereas the right side the patterns for the designating the method part.

In the lower right the patterns arising from the use of curly braces syntax are summarized. The number of possible combinations for method invocations is over 700 patterns. This yields a vast ecosystem of 'species' that can be used to invoke methods.

2.2 Object instantiation patterns

The conventional approach for object instantiation is using the new operator applied to a class in order to invoke its constructor, as for example shown below:

```
$obj = new Product();
```

But in PHP this pattern can be enhanced by allowing the class name to be a string (literal or variable). This opportunity allows the use of the *variable variable* construct and the *curly braces syntax*, expanding the diversity of patterns. Also beside the use of the *new* operator, an object instantiation can be made with other approaches as well. A variable of any type can be casted to an object. The resulting object is always an instance of PHP's stdClass[5] which is commonly used as PSD[6]. Finally, objects can be created using the `unserialize`[7] function, which takes as input a serialized representation of an object and returns a new object. Objects are serialized using the complementary `serialize` function. These functions can also be applied to arrays instantiating a stdClass. However, object instantiation types by casting and unserialization have not been further classified into species in this study, as they do not constitute a typical way of object creation for system classes.

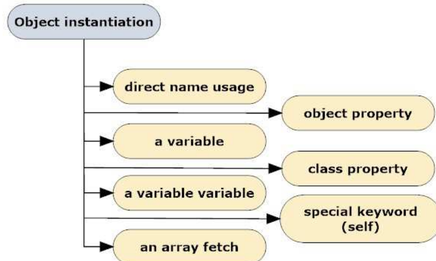

Figure 2. Object instantiation patterns

3. CASE STUDY DESIGN

The primary goal of this study is to analyze open source PHP projects, so as to identify the frequency of method invocation and object instantiation patterns. In the following sections the data collection approach and the methodology for the analysis of results will be presented.

3.1 Data collection

In this study each project (case) has been analyzed separately. For each project we have obtained multiple units of analysis (versions) and therefore our study can be characterized as an embedded multiple case study. The cases of this study have already been presented in Section 2 (Table 1).

For each project version available in GitHub the collected variables are: the number of occurrences of each method invocation pattern and the number of occurrences of each object instantiation pattern. These are used to calculate the diversity of species in each version, along with the species richness (see Section 3.2) and the dominant species (see Section 3.3)

Data collection has been performed through an evolution analysis tool, created by the first author. The complete history of the source code of each project is downloaded from GitHub and measurements are performed by distributed analysis workers. The tool has been implemented in PHP and employs an open source PHP parser[8], to build the AST representation of the code, which is compressed and stored in a mongoDB gridFS. The measurements are stored in a mySQL database. The implementation also includes an algorithm for the identification of invocations made with the Reflection library, since variables types in PHP are not declared. To address this challenge we employed a simple symbol table to keep track of the references of Reflection library objects.

3.2 Data Analysis

3.2.1 Shannon-Wiener Diversity Index and species richness

In order to depict the evolution of the diversity of the used patterns we employed the Shannon-Wiener Diversity Index. This measure is a popular diversity index in ecology that takes into account not only the number of species but also the population that belongs to each species [1], [2], which is also known in Information Theory as the Shannon Entropy. The objective is to measure the amount of order or disorder (entropy) in a system and the Shannon-Wiener diversity index is obtained as $H' = -\sum_{i=1}^{S} p_i \ln(p_i)$, where H' is the Index of species diversity, S is the number of species and p_i is the proportion of total sample belonging to i-th species.

In our case we have counted method invocations -the individuals- in each system and we have categorized them to invocation patterns -the species-. An increase in the index value over time implies that the population is distributed across a larger number of species. If a dominant species appears –with a large number of individuals belonging to that species- the index will decrease. In our case when new invocation patterns appear, the Shannon-Wiener index will increase. On the other hand if an invocation pattern becomes dominant -larger share of invocations are made with the same pattern- the index will decrease.

In order to obtain an insight of the diversity evolution we have plotted the species richness as well. Species richness is the simplest figure to show and simply refers to the number of species present in the sample. In Figure 3. Dropping diversity index and constant richness is shown. In the marked area of the chart the diversity index is decreasing (the trend is shown with the bold line), a fact that could be misinterpreted as a phase where the number of species is also decreasing. But in the same period the richness plot is constant (the trend is shown with the bold line). Taking into account both trends we are driven to right conclusion, that some species' proportion of the population is growing and they are getting dominant.

[5] http://php.net/manual/en/reserved.classes.php

[6] https://en.wikipedia.org/wiki/Passive_data_structure

[7] http://php.net/manual/en/function.unserialize.php

[8] https://github.com/nikic/PHP-Parser

Since the Shannon-Wiener index does not have an upper limit like e.g. the Gini index[9] , we have plotted the results with the same y-scale in all projects, so that the results would be comparable. Along with the Shannon-Wiener index we plot the species richness (number of patterns) with its scale on the right axis as shown in the example of Figure 3.

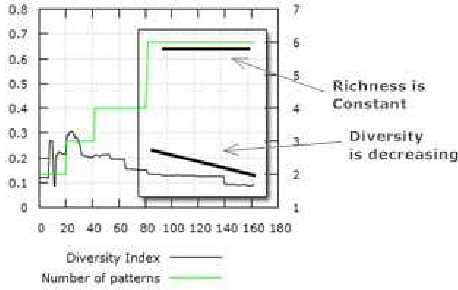

Figure 3. Dropping diversity index and constant richness

3.2.2 Dominance Index

In order to identify the dominant patterns we employed a calculation method for the dominant species in an ecosystem. In the context of ecology, to calculate species dominance in large surface areas where more than one observation stations are used, the so-called *dominance index* Y is used [3], [4]: $Y = \frac{n_i}{N} f_i$, where n_i is the abundance of species i, f_i is the occurrence frequency of species i and N is the total abundance. Occurrence frequency f_i refers to the proportion of stations reporting the occurrence of a species in the total number of stations [3]. Frequency scales population proportion according to the number of stations in which a species is observed. This way if a species has a large population proportion in one station only; its effect to the global population is scaled down. A threshold for defining dominant species [3] is usually set (e.g. in studies of zooplankton).

Table 3. Example of dominance calculation

Species	Stations 1 n_i	Stations 2 n_i	n_i	n_i/N	Stations	f_i	Y	Is dominant?
Sp.1	13	0	13	0.0219	1	0.5	0.0109	No, Y<0.02
Sp.2	233	34	267	0.4517	2	1	0.4517	Yes, Y≥0.02
Sp.3	5	4	9	0.0152	2	1	0.0152	No, Y<0.02
Sp.4	0	302	302	0.5109	1	0.5	0.2554	Yes, Y≥0.02
		N=	591					

In Table 3 an example of dominance calculations with two stations is shown. Four species have been found, three of them (Species1, 2 and 3) observed from station 1 and three of them in Station 2 (Species 2, 3 and 4). The abundance counted from each station is in the columns Station 1 n_i and Station 2 n_i, accordingly. The column n_i holds the sum of the total abundance for each species and *ni/N* is the proportion to the total population *N*. The column Stations holds the number of stations in which each species has been observed and f_i is the proportion of them to the total number of stations. For example the first species was observed in one of the two stations; hence the frequency f_i is 0.5. Also for species observed in all stations (i.e. Species2) *Y* is equal to their population proportion (n_i/N) since the frequency f_i is 1. The dominance index is in column *Y*.

In the examined ecosystem of PHP projects, taking into account that they are open source and therefore knowledge can flow freely among developers, we consider the individual systems as stations observing the occurrences of method invocation and object instantiation patterns among them. We will calculate the dominance index and apply the threshold in our context. We defined two criteria for a pattern to be dominant: (a) be present in all systems of the corpus (frequency f_i=1), and (b) its proportion over the total population (n_i/N) to be over 2%.

4. RESULTS AND DISCUSION

4.1 Method invocations

To analyze the evolution of programmers' habits while invoking methods we have calculated the Shannon-Wiener index for each version of the examined systems. Figure 4 shows the evolution of the Shannon-Wiener index along with the species richness (number of patterns) with its scale on the right axis. The horizontal axis represents the consecutive versions of each project.

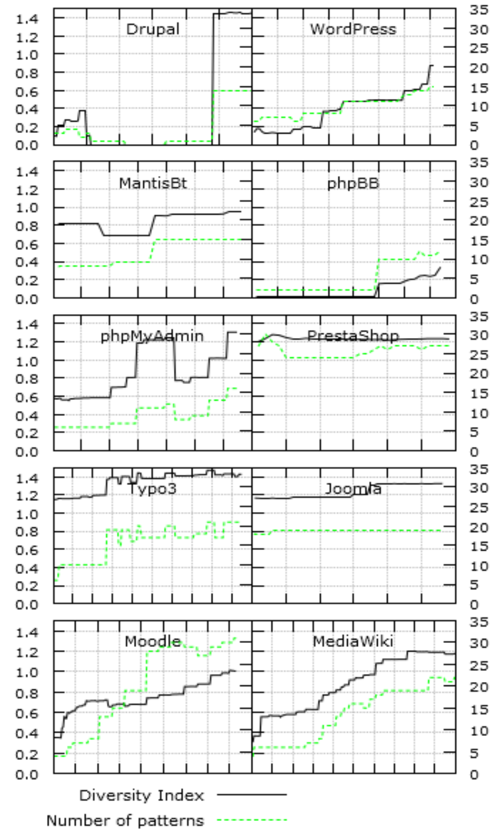

Figure 4. Diversity of method invocation patterns

Based on Figure 4, we can observe that the evolution of different projects varies, and three main groups can be formed. First, *WordPress, Moodle, MediaWiki* and *phpBB* present an increasing diversity and species richness, indicating that their new modules employ new and different method invocation patterns. Second, *PrestaShop* and *Joomla* exhibit an almost constant diversity and species richness. Constant richness indicates that even if many components of the project have been reengineered or new components have been added, almost the same patterns for handling dynamic invocations are used throughout the history of the project. Constant diversity suggests that these dynamic patterns are used with the same proportion among all method invocation patterns, regardless of the project size. Finally, by observing the evolution of *Drupal* and *phpMyAdmin,* we can suggest that their diversity and richness exhibit some peaks. Concerning Drupal, at

[9] https://en.wikipedia.org/wiki/Gini_coefficient

the point of the peak, Drupal 7 was released, introducing a turn to the object-oriented paradigm, which was almost not applied in previous versions. In the case of phpMyAdmin at the version where diversity presents a peak, a third party library, namely PHPExcel[10]. PHPExcel is a huge and complex OO library, employing patterns that were not used in the system before. A few versions later (3.4.5), the library was removed, resulting in a diversity and richness drop to the normal levels of the system's core.

By aggregating the results of all projects, we can identify the most frequently used method invocation patterns in the complete ecosystem. In Table 4, the results of the dominance index Y are shown for each pattern. The values over the threshold of Y≥0.02 are grayed indicating the dominant species.

Table 4. Dominance index threshold validation

	Species (Object part → Method Part)	Dominance Index Y
1	Variable → Explicit method name `$object->method();`	0.6271723
2	Class name → Explicit method name `Class::method();`	0.1802146
3	Object property → Explicit method name `$obj->objref->method();`	0.0875515
4	Array element → Explicit method name `$a[2]->method();`	0.0329878
5	Method call → Explicit method name `$obj->getORef()->method();`	0.0307420
6	Special key word → Explicit method name `self::method();`	0.0304449
7	Static call method → Explicit method name `Class::getORef()->method();`	0.0057142
8	Variable → Variable `$obj->$methodName();`	0.0006488
9	Variable → literal inside curly braces `$obj->{"methodname"}();`	0.0003247

*It should be noted that patterns in Table 4 are encountered in all projects ($f_i = 1$). As a result, the dominance index is equal to n_i/N. Only top 9 rows are shown due to space limitations, but some results refer to all of them.

As it is reasonable to expect, the most dominant method invocation pattern, accounting for 62.7% of the cases is the most conventional one, i.e. `$object->method()`. The explicit designation of the method part is encountered in all dominant patterns implying that PHP developers target the dynamic behavior on the object part of method invocations. The second most popular method invocation pattern refers to static method calls, which, although they violate object-oriented principles, are employed in 18% of the cases. Species 3, 4 and 5 employ an indirect way of retrieving the object on which the method will be invoked (e.g., by indexing an array of objects, by calling a getter function, etc.). Of these patterns, the most frequently used one (8.7%) is the one that retrieves the object through the public attribute of another object, which is also against proper design principles. As it can be observed, irregular patterns, such as the ones relying on *variable variables* and *curly braces* syntax are scarcely employed. Moreover, around 30% of method invocations in the examined systems (aggregating the entire result set) do not follow object-oriented guidelines as they essentially constitute a functional programming approach.

4.2 Object instantiations

In the corpus of analyzed systems we have been able to identify all approaches that can be used for object instantiation. As expected the `new` operator is by far the most popular one (i.e., 96.39% of the total number of object instantiations). Casting is used in 3.47% of the cases where an object is obtained. Finally, `unserialize` is of limited use, since it is found only in 0.14% of the total object instantiations, and half of the projects.

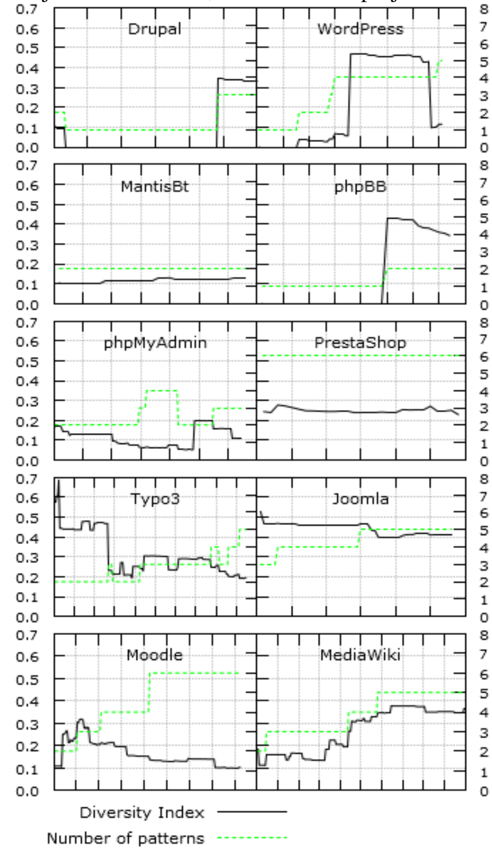

Figure 5. Diversity of object instantiation patterns

In order to analyze the evolution of programmers' habits while instantiating objects, we calculated the Shannon-Wiener index for each version of the examined systems (Figure 5). Along with the Shannon-Wiener index the species richness (number of patterns) is also plotted with its scale on the right axis. The horizontal axis represents the consecutive versions of each project. Similarly to method invocations, the evolution of object instantiation patterns varies between projects. *Drupal* and *phpBB* employed for a long period just one pattern, as a natural consequence from the fact that OO code was almost non-existent. *MantisBt* and *PrestaShop* have almost a constant diversity index and species richness, indicating stability in object instantiation mechanisms. *Typo3* and *Moodle* exhibit a reducing diversity index indicating that dominant patterns appeared in the system, possibly implying that developers agreed on a set of commonly used techniques, which they followed for newly added code. *MediaWiki* on the other hand has increasing diversity index and species richness, indicating that new patterns are employed in the project.

5. RELATED WORK

The study of scripting languages is a subject trending the last few years, but publications related to evolution of applications implemented in scripting languages and especially in PHP are limited. Dynamic features as those offered by scripting languages are not offered by compiled languages and are rarely studied in literature.

[10] https://github.com/PHPOffice/PHPExcel

Hills [5] studied the evolution of the usage of PHP's dynamic features including variable variables, magic methods, `eval` function, function handling functions and casting. He presented the aggregated trend of each group of dynamic features without focusing in detail on method invocations and object instantiations. The dataset consisted of WordPress and MediaWiki. Hills [6] studied also the exploitation of idiomatic usages of variable features in a set of 20 open source PHP projects (including 7 projects employed in our corpus). He developed a list of patterns to detect such usages and classified them based on their structural application. The three categories are, loop patterns, assignment patterns and flow patterns. For each category he developed a lightweight detection algorithm. Additionally he identified a set of "anti-patterns" that consist of cases that cannot be resolved using static analysis. The cases that can be resolved are those where the variable part of the variable variable is literally defined at function level scope.

Eshkevari et al. [7] studied the runtime type changes of variables in a corpus of PHP applications, consisting of phpBB, Drupal and WordPress, in order to study the effort needed to make existing PHP applications conform to HACK's type system requirements. They employed a hybrid approach combining static and dynamic analysis employing TXL[8] source transformation and Watir[11] crawler for runtime instrumentation. Amanatidis and Chatzigeorgiou [9] studied the evolution of PHP applications from the perspective of Lehman laws detecting evolution of changes in files and other software engineering metrics, like McCabee's complexity on a set of 24 projects. Their results support with confidence that open source PHP applications are maintainable and sustainable, despite the fact that they are implemented with a scripting language.

Wang et al. [10] studied dynamic features in Python. They chose four groups of the most commonly used, classified as Introspection, Object Changes, Code Generation and Library loading. Their functionality does not match the corresponding cases in PHP and therefore, a direct comparison to their results is not possible. Callaú et al. [11] performed an empirical study on the usage of dynamic features in Smalltalk. They investigated three sets of Smalltalk's dynamic features, classes as first-class objects, behavioral reflection and structural reflection. Those are similar to dynamic features offered by PHP, i.e. structural reflection is implemented in PHP with dynamic code includes. Their overall result is that the methods in their corpus employing dynamic features were 1.29% over the total number of methods.

6. LIMITATIONS & VALIDITY THREATS

One limitation of this study can be identified in the algorithm for the detection of the occurrences where the `unserialize` function returns an object. If no method is invoked on the created object within the same function or class then our algorithm cannot determine if the result of the `unserialize` function was an object or not. The same restriction applies to the algorithm that detects method invocations with employment of the Reflection library. However, we believe that this limitation is not threatening the validity of the results, since according to our own experience on various PHP projects, such cases are very rare in practice.

The presented empirical study suffers from threats to generalization, in the sense that the examined systems, although relatively large, constitute only a small portion of the available PHP code. This means that our findings might not be valid for other applications written in PHP. Nevertheless, we need to note that the projects that we have selected are among the most famous PHP web applications, and therefore can be considered as representative and influential to the community.

7. CONCLUSIONS

The dynamic features offered by PHP language to invoke methods and instantiate objects constitute a rather diverse ecosystem, from which developers can harvest the most suitable patterns that fit their goals. The analysis of ten extremely popular web applications suggested that in the case of method invocations developers use a very limited number of the theoretically possible patterns, implying that even a dozen of them can cover a substantial part of the knowledge required for building OSS PHP projects. This finding can be justified by the comfort coming from repetition, which can be stronger than the urge to explore exotic idioms of the language. On the contrary, for object instantiations the number of offered patterns is significantly smaller than for method invocations (limited to seven), forming a set of idioms which is easily memorable, since six of them have been consistently found in our corpus. The present work is in progress to investigate in more depth the employment of those language features and their impact on code stability in order to get an insight on whether there is a long term benefit from their employment.

8. REFERENCES

[1] I. F. Spellerberg and P. J. Fedor, “A tribute to Claude Shannon (1916–2001) and a plea for more rigorous use of species richness, species diversity and the ‘Shannon–Wiener’ Index,” *Glob. Ecol. Biogeogr.*, vol. 12, no. 3, pp. 177–179, 2003.

[2] C. J. Krebs, *Ecological Methodology*. New York: Harper Collins Publishers, 1989.

[3] Z.-L. Xu and C. Li, “Horizontal distribution and dominant species of heteropods in the East China Sea,” *J. Plankton Res.*, vol. 27, no. 4, pp. 373–382, 2005.

[4] K. Li, J. Yin, L. Huang, and Z. Lin, “Seasonal variations in diversity and abundance of surface ichthyoplankton in the northern South China Sea,” *Acta Oceanol. Sin.*, vol. 33, no. 12, pp. 145–154, 2014.

[5] M. Hills, “Evolution of dynamic feature usage in PHP,” in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 525–529.

[6] M. Hills, “Variable Feature Usage Patterns in PHP (T).,” in *ASE*, 2015, pp. 563–573.

[7] L. Eshkevari, F. D. Santos, J. R. Cordy, and G. Antoniol, “Are PHP applications ready for Hack?,” in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 63–72.

[8] J. R. Cordy, “The TXL Source Transformation Language,” *Sci Comput Program*, vol. 61, no. 3, pp. 190–210, Aug. 2006.

[9] T. Amanatidis and A. Chatzigeorgiou, “Studying the Evolution of PHP Web Applications,” *Inf Softw Technol*, vol. 72, no. C, pp. 48–67, Apr. 2016.

[10] B. Wang, L. Chen, W. Ma, Z. Chen, and B. Xu, “An empirical study on the impact of Python dynamic features on change-proneness.,” in *SEKE*, 2015, pp. 134–139.

[11] O. Callaú, R. Robbes, É. Tanter, and D. Röthlisberger, “How Developers Use the Dynamic Features of Programming Languages: The Case of Smalltalk,” in *Proceedings of the 8th Working Conference on Mining Software Repositories*, New York, NY, USA, 2011, pp. 23–32.

[11] http://watir.com