Exploring the Frequency and Change Proneness of Dynamic Feature Pattern Instances in PHP Applications

Panos Kyriakakis, Alexander Chatzigeorgiou, Apostolos Ampatzoglou, Stelios Xinogalos
Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece
panos@salix.gr, achat@uom.gr, apostolos.ampatzoglou@gmail.com, stelios@uom.gr

ABSTRACT

Although numerous technologies are available for developing web applications, PHP holds the lions' share of web content today. PHP offers several features that enable developers to easily produce dynamically extendible code, forming an entire ecosystem of standard as well as more 'exotic' opportunities that can be exploited. One reason that drives developers to rely on the dynamic features of a scripting language is to enable effortless functionality extensions. The aim of this work is twofold: initially, we (a) provide an overview of all possible dynamically extendible code patterns (i.e., either through method invocation, or object instantiation) and (b) investigate their frequency by mining the code base of ten milestone PHP projects to identify the subset of patterns that developers actually use. Next, in order to investigate whether the expected flexibility of these patterns stands in practice, we examine if code chunks that instantiate them are more stable than other parts of the code. In particular, we study whether methods that employ dynamic invocation and instantiation patterns are less change prone than the other methods. The findings imply that although a small subset of all the theoretically feasible patterns is actually put to use, the code that is developed upon such patterns is less change prone.

Keywords: PHP, software maintenance, method invocation, object instantiation, change proneness

1 INTRODUCTION

The PHP language has been extremely popular for more than a decade, while projects of any size and business domain have been developed using PHP as a server-side programming language. Considering that the majority of PHP applications are enterprise grade applications with heavy usage, and that in such projects requirements are changing fast, a key-driver for successful PHP applications is coding flexibility. One of the key characteristics of PHP, which is expected to support flexible coding, is that it is loosely-typed. Based on this, PHP offers dynamic features (a.k.a. PHP idioms) for method invocations and object instantiation, i.e., approaches that introduce flexibility during the creation of objects and invocation of methods. These dynamic features have been introduced in order to provide the ability to developers to build dynamically extendible modules adhering to the Open-Closed principle [1]. The goal is to enable such modules to be extended with minimal to no change effort in their code. Obviously, for practitioners and researchers the question of interest is whether this goal has been accomplished.

In this paper we attempt to quantify changes and investigate if dynamic features of the language render the methods that utilize them less change-prone. By less change-prone methods we refer to code that is either flexible enough to withstand changes to the overall functionality or structured in a way that adheres to the Open-Closed Principle according to which behavior is extended by adding new code rather than modifying existing code. Hence, we evaluate the extent to which the employment of dynamic features is beneficial, to the evolution of the code by reducing maintenance costs. However, as a first step of this work, we investigate and categorize the ecosystem[1] of PHP method invocation and object instantiation patterns by parsing existing literature, mostly the grey ones (i.e., blogs, webpages, etc.).

[1] We use the term *ecosystem* in analogy to software ecosystems, implying a collection of species (method invocation and object instantiation patterns), which are developed and co-evolve in the same environment (i.e. that of actual PHP projects)

This step is considered necessary, since there is a vast diversity of theoretical variations of these idioms, which are identified and listed along with syntactic examples. To this end we classify PHP method invocation and object instantiation patterns into certain families of patterns according to their main characteristics. To the best of our knowledge, this is the first time that all theoretically possible PHP method invocation and object instantiation patterns have been catalogued. Moreover, a minimal subset of these alternatives is identified, empowering practitioners to successfully study and adopt best practices that have been put into practice in well-known projects. The employed patterns can also be valuable to educators, allowing them to focus on efficient PHP object oriented techniques. Furthermore, our results indicate that employing "irregular" patterns can aid in developing code that adheres to the 'open-closed' principle and generally improve the design and implementation of applications.

The rest of the paper is organized as follows: In Section 2 related work is presented, while in Section 3 background material is presented in order to introduce PHP's method invocation and object instantiation ecosystem. In Section 4 the design of the study is analyzed, including details on the statistical analysis used and the normalized distance calculation. In Section 5 the results are presented and discussed, and in Section 6 some implications to researchers and practitioners are presented. In Section 7 threats to validity of this work are discussed, and finally in Section 8 conclusions are presented.

2 Related Work

The goal of this study is to identify method invocation and object instantiation patterns and create a taxonomy of them, as well as to investigate if there is an actual benefit to the usage of the dynamic ones. PHP's dynamic features are a subject that has started to attract the interest of the academic community in the last few years. However, to our knowledge there is no other work presenting the usage of dynamic method invocation and object instantiation patterns in PHP and quantifying the possible long term benefit of their employment. The work that is more relevant to the present study is that of Eshkevari et al. [2] who performed an empirical taxonomy of type changes employing a hybrid approach combining static with dynamic analysis in order to track the type changes of variables. The authors have performed dynamic analysis employing a test tool to cover all possible data flow paths, a fact that makes the analysis hard to utilize. They concluded that there is no actual benefit of dynamic type changes in PHP. Hills et al. [3] studied the usage of PHP's dynamic features and the frequency of their usage in order to define the most popular subset of them and highlight them as the features that should be further investigated, and Hills [4] studied the usage evolution of the dynamic features presented in their aforementioned work showing that features related to dynamic use of variables in object referencing and method invocation are more often used in the later versions of the studied corpus. Hills [5] analyzed the usage of variable features pointing to patterns detecting value flow and reachability, but without a conclusion if there is an actual benefit of their usage.

Amanatidis and Chatzigeorgiou [6] studied the evolution of PHP applications in the aspect of Lehman laws, including change detection but have not investigated the reasons that caused changes. Change proneness related to dynamic features in Python has been studied by Wang et al. [7]. They analyzed a corpus of 7 Python projects with, cumulatively, 52 versions. Their approach is contrary to ours: they first grouped the files of the source code to change prone and not, and, after this, counted the occurrences of dynamic features in each group. Their result was that files employing dynamic features are more change prone than the rest. An object-sensitive type analysis for PHP was introduced by Van der Hoek and Hage in [8], based on the notion of data flow analysis (monotone frameworks) in order to identify type checking issues related to objects. Employing control flow and data flow points-to analysis they have tried to resolve objects referred to by variables, thereby identifying object instantiations. They identified a small number of instances of method calls that fit PHP's idioms that cannot be resolved. Type analysis tools for PHP employing only static analysis have been developed: Phantm by Kneuss et al. [9]; SAFERPHP by Son and Shmatikov [10]; and Pixy by Jovanovic et al. [11]. Phantm performs a type mismatch analysis detecting the return type of functions and methods, and also comes with detailed information on the return values of each function, so it is helpful for improving documentation. SAFERPHP and Pixy are focused on security analysis, but still resolve many dynamic features of PHP utilizing static analysis. Finally, Hauzar and Kofron [12] presented a framework for static analysis of PHP applications that resolves dynamic features and uses materializations for heap value analysis. They employed their framework for taint analysis.

Code stability (or instability) has been studied extensively in the field of clone detection. Hotta et al. [13] measured code stability counting the frequency of modification of code with the hypothesis that a region is less stable if it has higher frequency of changes. Krinke [14] measured the ratio of changed lines of code over the total LOC. Metrics defined in the aforementioned studies are calculated cumulatively in all available revisions of the subject project. In the present study we are closer to Krinke's approach but we calculate change metrics per version and then apply statistical analysis in order to draw conclusions. Yamamoto et al. [15] measured code similarity based on source code correspondence, thus comparing only lines of code that are present in both clones. They all employed UNIX diff to determine the changes in source code.

It should be noted that in a previous paper [16] we presented the patterns of object instantiations and method invocations in the PHP language using the same corpus as the present study. In that study we employed ecology statistics in order to present the evolution (across versions) of the diversity and dominance of species (patterns) in the ecosystem (corpus) under study. In the current study we present in detail the patterns and focus our investigation on the relation between the use of irregular patterns and the change proneness in the methods employing them.

3 Background Material

An instance method or a class method can be invoked using a number of approaches depending on the implementation language. Object oriented grammars define a unique operator, such as the dot operator in Java for both cases or a different operator for each case. PHP uses two distinct operators. PHP leverages the use of dynamic aspects, for example, by providing reflection and 'function handling' functions and by offering unique language constructs such as variable variables and curly braces syntax. In the following section we will present in detail those variations.

3.1 Method Invocation

There are a number of approaches to invoke an instance method or a class method through the use of the **object operator** (->) and the **double colon operator** (::) respectively. Both operators are binary, with the object instance or class as the left hand side operand and with the method to be invoked as the right hand side operand, as is common in almost any object oriented language:

```
$object->method();    // invocation of object's method
ClassName::method(); // static method invocation
```

Additionally, PHP offers a set of functions called function handling functions (usually referred to as function functions) for function and method invocation, which comprise the functions: call_user_func[2], `call_user_func_array`[3], `forward_static_call`[4] and `forward_static_call_array`[5]. Their first argument has to be a callable[6]. A 'callable' in PHP can be either a string containing a function name (i.e. "`getPayment`"), or a static method including the class name and method name in the format "`classname::methodname`" (i.e. "`Taxes::calculateTax`"), or an array with two elements. The first element has to be an object instance or a class and the second the method to be invoked (i.e. `array($obj, "getPayment"`)). The cases of interest are those that use callables to invoke object methods or class methods. Therefore, the statement:

```
call_user_func(array($object, "methodName"));
```

is treated as equivalent to the expression `$object->"methodName"`. Also, as in many other languages, PHP offers a reflection library[7]. Employing the reflection library, the programmer can invoke instance or class methods in a dynamic manner.

```
$reflector = new ReflectionClass($className);
if( $reflector->hasMethod($methodName) ) {
        $reflection->getMethod($methodName)->invoke(null);
}
```

It is required to have an object reference to call an object's method, or the name of the class as a string to make a static call. The method is referred to by a string containing the method's name. Therefore, even an invocation employing the reflection library can be considered equivalent to the expression

[2] http://php.net/manual/en/function.call-user-func.php

[3] http://php.net/manual/en/function.call-user-func-array.php

[4] http://php.net/manual/en/function.forward-static-call.php

[5] http://php.net/manual/en/function.forward-static-call-array.php

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

[7] http://php.net/manual/en/book.reflection.php

$object->$method. However, reflection library usage introduces additional complexity when one attempts to track its usage.

TABLE 1 METHOD INVOCATION OPERANDS IN EACH INVOCATION TYPE

Invocation Type	**Object/Class (*Owner component*)**	**Method (*Method component*)**
Object operator	Left hand side operand	Right hand side operand
Double colon operator	Left hand side operand	Right hand side operand
Function handling functions	First element in callable array	Second element in callable array
Reflection	First argument of the invoked method or the reflected class for static call.	The reflected method

Consequently, each and every invocation approach is equivalent to the generic expression operand->operand. In order to resolve the ambiguity of the operands, we will refer to the generic expression's operands as the *components* of the expression. These two components (Table 1), namely object reference or class (the *owner component*) and method (the *method component*), are the aspects that have to be resolved during an invocation. Here we attempt to systematically record all possible method invocation patterns. The point that differs in dynamic languages like PHP is the patterns that can be used to obtain the object reference, the class name and the method to be invoked. Next, we will try to enumerate the possible patterns to obtain an object reference, in addition to the patterns for the class part of static calls, as there are some differences compared to the object reference. Finally, the patterns for the method name part will follow.

3.1.1 The Owner Component

In PHP there are various patterns by which an object or class can be obtained, listed in Table 2 along with an example of usage.

TABLE 2 SUMMARY OF OWNER COMPONENT PATTERNS

	Obtain object reference patterns	**Example**
O1	through a variable	**$variable**->method();
O2	through an array element	**$array[4]**->method();
O3	through a method call	**$obj->getObject()**->method();
O4	through a property of an object	**$obj->object**->method();
O5	through a static method call	**Clazz::getObject()**->method();
O6	through a static property	**Clazz::object**->method();
O7	through a function call	**getObject()**->method();
O8	through a new instance	**(new Clazz())**->method();
O9 O9.C*	through a variable-variable (and variations from the use of curly braces)	**$$varname**->method();
O10.C*	through any curly braces expression (derived from callable used in functions)	See Table 3
S1	Class name	**Product::**getPrice();
S2	Special keywords (self, static, parent)	**parent::**getPrice();

Patterns O1 to O8 are common and can be observed in almost any language, dynamic or not. On the other hand, patterns O9 and O10 are features that are probably unique to PHP. A variable can be accessed directly using its name, for example $var, or through the use of a *variable-variable*[8]. In this case, a variable holds the name of the variable that is to be accessed and, using the variable-variable language construct (using a second dollar sign in front of the variable), the value of the variable is accessed. For example:

```
$hello = "Hello world"; // variable holds the text "Hello world".
$varVar = "hello";  // a common string variable holding "hello"
echo $$varVar;      // will output the value of variable $hello
```

Variable variables provide programmers the ability to dynamically build the name of the variable to be accessed according to certain conditions or user input during runtime. Additionally, the complementary language construct in PHP that can be used with variable variables is the *curly braces*[9] syntax. The curly braces syntax has been introduced in order to evaluate variables placed into string literals. There are several ways to construct the variable name. Independently from the expression used to construct the string, the expression patterns that can be encountered are shown in Table 3. Especially in the case

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

[9] http://php.net/manual/en/language.types.string.php Complex (curly) syntax

of concatenation the operands of the concatenation can be any expression resulting to a string. These patterns can be used as the variable part of a variable-variable, since the variable name is a string.

```
$alpha = new Alpha(); // variable holds an instance of class Alpha
${"al"."pha"}->m(); // concatenated literals "al" and "pha" result
                    // in the name of the variable $alpha and an
                    // invocation of method m() of the object
                    // $alpha is performed.
```

Therefore the patterns listed in Table 3 are additional variations of pattern O9, named O9.C* (where * is 1...10). Even though curly braces and variable variables fall into the same general pattern, we count each case separately in order to gain a better insight into programmers' habits.

TABLE 3 CURLY BRACES SYNTAX INTRODUCED PATTERNS

	Curly Braces Patterns	**Example**
C1	A literal	`{"varname"}`
C2	A concatenation	`{"var"."name"}`
C3	A function call	`{func()}`
C4	A method call	`{$obj->method()}`
C5	A property fetch	`{$obj->property}`
C6	An array fetch	`{$arr[2]}`
C7	A static method call	`{Clazz::getName()}`
C8	A static property fetch	`{Clazz::$name}`
C9	A ternary expression	`{($expr ? $obj1 : $obj2)}`
C10	A constant	`{TAXATION}`

*The patterns correspond to top-level constructs only.

The double colon operator cannot be applied to a function or a method call or a property, but only to a variable. For example the statement `Product::getTaxClass()::calculate()` is not valid. Therefore, for static method invocations only patterns O1, O2 and O9 can be used (along with all O9.C*). Additionally, the trivial patterns of directly using the class name (S1) and any of the special keywords self, parent, static (S2) can be employed.

In addition, the callable used in the set of *function functions* that can be employed to invoke methods introduces an extra set of patterns. The first element of the array that defines the class or object in the callable duplet can be an object or any expression that evaluates to a string that represents a class name defined in the system. Since any of the C* patterns can be used, we will name this pattern O10.C* (where * is any of the C.1… 10 patterns).

Since in our study we do not need to distinguish method calls to objects and static method calls, we merged the object part patterns to a single list that refers to the class or object on which the operation is performed. The merged list contains the eleven O1-O9, S1-S2 patterns along with the ten C1-C10 patterns for each of O9 and O10. Therefore, we have identified 31 distinct patterns to get an object's reference.

3.1.2 The Method Component

The method component of the invocation can also vary but it is identical regardless of whether it is a static invocation or an invocation of an object method. Actually, there are three ways to invoke a method, namely (M1) using the method's name, (M2) using curly braces to evaluate a string expression to get the method name and (M3) using a variable-variable that returns a string with the method name. As already mentioned, the variable-variable pattern can be extended with the use of curly braces syntax as shown in the object reference patterns, and therefore patterns M2 and M3 can both be extended with the curly braces patterns named M2.C* and M3.C* accordingly. In Table 4 method component patterns are listed.

TABLE 4 METHOD COMPONENT PATTERNS

	Method component patterns	**Example**
M1	Direct name usage	`$obj->getProduct();`
M2	A variable	`$obj->$methodName();` or `$obj->{$methodName}();`
M2.C*	The additional curly braces patterns	`$obj->"get{getCountryCode()}TaxRate"();`
M3	A variable-variable	`$obj->$$var()`

M3.C*	The additional curly braces patterns applied to variable variable.	`$obj->$"{getCountryCode()}taxMethodVar"();`

To summarize, theoretically there is a vast number of possible patterns by which a method can be invoked. The Cartesian product of the set of the patterns for the object component and the set for the method part counts 713 members.

3.2 Object instantiation

PHP offers three approaches to instantiate objects: the use of the new operator, casting other data structures to objects and un-serialization. Within the context of object instantiation the subject of the three approaches does not have the same meaning. In case of the new operator the operand is a class name. Object casting in PHP is much different than from Java; in PHP object casting converts an array or any scalar type to an `stdClass`[10] object, hence the subject of casting cannot be a class. In un-serialization the subject is a string with serialized data. Both object-casting and un-serialization are not approaches to explicitly instantiate an object of a system class. Therefore, we will present measurements for the three techniques separately.

3.2.1 New operator

The *new* operator creates an object instance of the class, for example: `$a = new MyClass();` This is the straightforward use of the operator. The class name to be instantiated can be contained in a variable (e.g. `$a = new $classname();`) or variable variable (e.g. `$a = new $$classnamevar();`), but curly braces syntax is not allowed. The variations are listed in Table 5.

TABLE 5 CLASS NAME SOURCES

	Class name source	**Example**
I1	Direct name usage	`$o = new MyClass();`
I2	A variable	`$o = new $myClass();`
I3	A variable variable	`$o = new $$myClassNameVariable();`
I4	An array fetch	`$o = new $a[0]();`
I5	A property fetch	`$o = new $a->className();`
I6	A static property fetch	`$o = new Test1::$sClassName();`
I7	Special keyword	`$o = new self()`

3.2.2 Casting

Passive data structures[11] (PDS) in PHP can be created with PHP's stdClass without having to define a class for each use case. Therefore, if a developer wants to pack some data in a PDS without adding dependencies, she can create an stdClass instance and add the desired properties to it. A workaround is to exploit PHP's type juggling mechanism[12] and cast an array holding the data to an object (object casting always returns an instance of the stdClass) and afterwards handle the object. A typical example is shown in the next code fragment. An array is cast to an stdClass object.

```
$currency = Tools::setCurrency(
  (object)array('id_currency' => 1, 'rate'=> 0.5) // cast an array to object
);
```

The equivalent code without use of casting which is more fluent, is shown below.

```
$currencySettings = new stdClass();
$currencySettings->id_currency = 1;
$currencySettings->rate = 0.5;
$currency = Tools::setCurrency($currencySettings);
```

Casting an array to stdClass is the official case described in PHP's documentation, but in fact any type can be cast to an object. As an example, in our corpus we have located a case where an object is created by casting a Boolean to an object: `$o = (object)false;` Since properties can be added to a PHP object at runtime, the use of PHP's stdClass is in many cases used as a PSD, for example to be processed inside a class instead of using an array. The subject of casting can be any valid expression, there-

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

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

[12] http://php.net/manual/en/language.types.type-juggling.php

fore it is meaningless to define a list of expected patterns. Instead we count and list the patterns discovered in the corpus.

3.2.3 Unserialize

PHP also offers the complementary functions `serialize`[13] and `unserialize`[14]. Serialize returns a string representation of a variable, of any type, and `unserialize,` given input a string containing the result of the serialize function, returns the original data type. Objects can be serialized and stored, for example, in a database or a cache file and later retrieved and un-serialized. In case of objects, the unserialized object is an instance of the same class as the serialized instance and not simply an instance of the stdClass. A significant difference in object instantiation with `unserialize` is that the class's constructor is not called. An example of that case is found in retrieving an AST from an XML file

```
$node = unserialize(
  sprintf(
"O:%d:\"%s\":2:{s:11:\"\0*\0subNodes\";a:0:{}s:13:\"\0*\0attributes\";a:0:{}}",
  strlen($className), $className
  )
);
```

The listed code instantiates an object of the class named in the variable `$className` without calling its constructor. Attributes are assigned later in the code. The developer, with the assumption that the XML data is valid, employed this pattern in order to speed up execution and simplify code. Serializing and un-serializing objects is a common persistence mechanism since the early days of PHP. This approach is also used in storing data that do not follow a predefined schema in relational database fields. Lately, with the extensive usage of micro-services, it is a common way to transport data objects through REST services since serialized data can be sent via HTTP methods to the clients where they are un-serialized to objects again and vice versa, but no such examples were found in our corpus. Since the `unserialize` function accepts one argument with the data to be un-serialized, which can be a string literal or any other valid PHP construct that returns valid serialized data forming a data source, we will not enumerate any patterns, instead we will count and list the ones used in our corpus.

4 CASE STUDY DESIGN

The second objective of this study is to empirically explore the landscape of method invocation and object instantiation types in real-world PHP applications and investigate: (a) the frequency of these pattern instances, and (b) whether PHP-type patterns lead to less change-prone code. To achieve this goal, we have analyzed data from ten PHP projects of various sizes and domains. In the following subsections, the four parts of our design are described.

4.1 Goal and Research questions

The goal of this study, adopting the formalism of the Goal-Question-Metrics (GQM) approach [17] can be stated as: ***Analyze*** Open Source PHP projects ***for the purpose*** *of* evaluating method invocation and object initialization patterns ***with respect*** *to* their diversity and relation to change-proneness ***from the perspective of*** researchers and software maintainers ***in the context*** *of* PHP web applications. Based on this goal the following research questions have been formulated so as to guide the case study design and reporting:

RQ$_1$: *Which PHP dynamic features are the most frequently used ones in web applications?*

> ***RQ$_{1.1}$***: *What is the population of method invocation patterns in PHP web applications?*
>
> ***RQ$_{1.2}$***: *What is the population of object instantiation patterns in PHP web applications?*

RQ$_1$ aims to shed light on the features that PHP developers utilize in order to carry out two important tasks in the object-oriented paradigm, namely the invocation of methods and the instantiation of objects. This part of the study will reveal the frequency of cases where special characteristics of PHP are exploited and the cases for which reverse engineering is hindered by the employed patterns.

RQ$_2$: *What is the relationship between PHP dynamic features and change proneness?*

> ***RQ$_{2.1}$***: *Do PHP-type method invocation patterns help code to be less change prone?*
>
> ***RQ$_{2.2}$***: *Do PHP-type object instantiation patterns help code to be less change prone?*

[13] http://php.net/manual/en/function.serialize.php

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

RQ_2 will attempt to investigate whether the use of the special PHP patterns offers any benefits to software development and maintenance with respect to change proneness. Overall, the goal of this study is to analyze whether idioms offered by a widely adopted scripting language can be beneficial to software development.

4.2 Selection of cases

As already mentioned the language of interest is PHP due to its tremendous popularity in web applications during the last decade. The criteria for selecting the projects to be analyzed are:

- to be written in PHP (at least for the majority of their functionality)
- to have their source code publicly available to allow for code analysis
- to be full-scale applications rather than small-scale libraries. We have made this decision for two reasons: (a) library code tends to be reused in various projects. If libraries' code was assessed, the dataset would be likely to include duplicate lines, due to the reuse of library code; and (b) small-scale libraries (or components) are highly likely to be the product of a single developer or a small team following a tight coding style dictated from the project leader. The inclusion of such projects might introduce bias to the dataset, due to the programming habits of a small group or individuals.
- to cover different business domains to increase the representativeness of the study results
- to be established in their business field, to increase the probability of including projects that have somehow influenced the community with their development practices
- to have a long history of development in order to get insight into the evolution of practices. Moreover, a long history possibly implies changes or enhancements to the core team members (along with their programming style)
- to have a vast community of developers and users in order to maximize the diversity of employed idioms
- to be projects of a large scale to increase the likelihood of dynamic features and diverse implementation practices

The selected projects are listed in Table 6, along with an overview of their functionality, their lifespan, size in thousand lines of code and number of analyzed versions.

TABLE 6 OVERVIEW OF EXAMINED PROJECTS

Project	Functionality	First Version			Last Version			Cumulative	
		Release	Year	kLOC	Rel.	Year	kLOC	# Rel.	kLOC
WordPress	Blog	1.5	2005	20	3.6.1	2013	200	71	6,914
Drupal	CMS	4.0.0	2002	14	7.23	2013	173	120	8,321
phpBB	Forum	2.0.0	2002	18	3.0.12	2013	194	37	2,815
MantisBt	Bug tracking	1.0.0	2006	68	1.2.15	2013	165	33	4,124
phpMyAdmin	Admin tool	2.9.0	2006	39	4.1.6	2014	248	129	13,985
PrestaShop	e-commerce	1.5.0.0	2011	266	1.6.0.10	2014	327	29	7,248
Typo3	CMS/CMF	3.6.1	2004	108	6.2.6	2014	597	189	56,802
Joomla	CMS/CMF	1.7.3	2011	237	3.3.6	2014	551	59	17,705
Moodle	e-learning	1.0.0	2002	15	2.8.1	2014	1,342	165	91,412
MediaWiki	wiki	1.1.0	2004	11	1.24.1	2014	372	200	33,567

The selected corpus cumulatively consists of more than 1000 versions, 240MLOC and 83 years of development. We note that for RQ_1, we use only the last version of each project, whereas for RQ_2, we use the complete history since we are interested in changes into the source code base between versions. Some of the projects in their early versions are mixed in the sense that they adopt both a procedural and an object oriented (OO) approach and gradually migrate to the OO paradigm. Others are OO all along the observation period. PHP also evolved in the same period, providing better support for OO. As a result, designers of the systems gradually adopt those new features, often moving to more elegant solutions. In FIGURE 1 the share of procedural and OO code blocks is shown for the final version of each project in our corpus. The percentage of methods ranges from 20% to almost 100%, with the majority of the projects employing classes and methods to a very large extent in their latest version. Among the OO features introduced to PHP is the abstract type of interfaces[15]. Since "*coding against interfaces, not implementations*" [18] is considered a good OO practice we also show the number of the defined interfaces in each system. In PHP there is not really a need to define an interface but it can indicate the

[15] http://php.net/manual/en/language.oop5.interfaces.php

ontological organization of the code. Therefore, systems with defined interfaces put effort in not just implementing functionality in classes, but also into organizing the code to adhere more strictly to the OO paradigm. In FIGURE 1 we show that a considerable number of interfaces are defined in half of the projects.

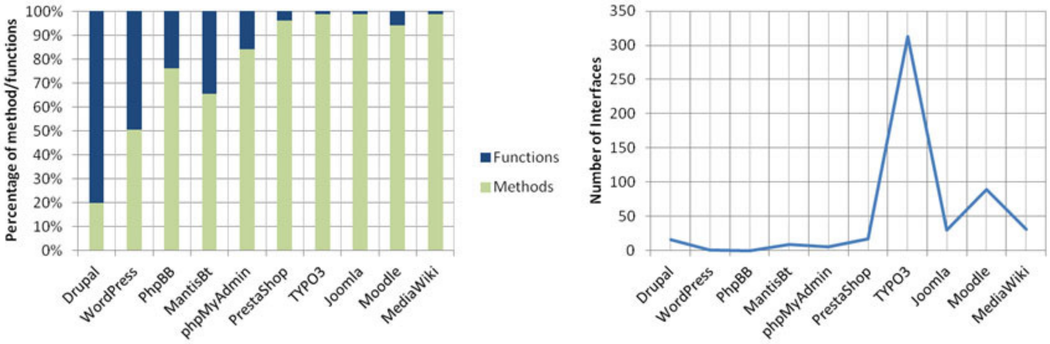

FIGURE 1 - RATIO OF METHODS OVER FUNCTIONS AND NUMBER OF INTERFACES PER PROJECT

4.3 Employed process and tools for data collection

In order to collect data for our study we used a PHP web application evolution analysis tool created by the first author (the architecture of the application is depicted in Figure 2). When the analysis job is executed the application downloads the source code of all available versions of the selected project from GitHub. Then, it parses the source code employing the Nikic/PHP parser[16] also developed in PHP and stores the serialized Abstract Syntax Tree (AST) of each file to MongoDB's GridFs[17] storage for further usage, so any analysis step that needs the AST will not have to parse the code again. MongoDB is a scalable No-SQL database that also offers the file storage engine GridFS which supports sharding and can be easily configured to utilize multiple nodes for storage. This way the analysis tool can be scaled to utilize multiple servers for better performance and handling larger amounts of data.

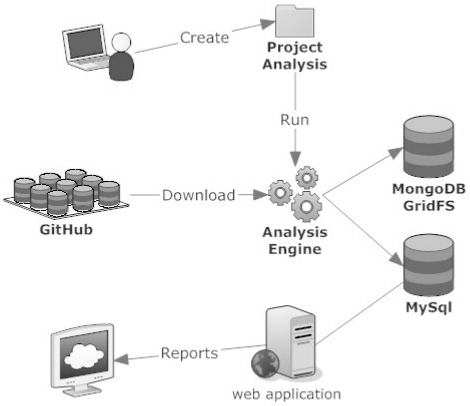

FIGURE 2 - ANALYSIS APPLICATION

Once the AST and directory structure are stored in the database, the process continues with the desired data collection and statistical calculations for each version and stores the results to a MySQL database. The user optionally can define an analysis workflow using any of the available modules of the tool. For example, if there is a need to collect extra data or calculate a statistic again, the user can define a workflow and run the analysis only with the needed modules, avoiding the need to run the complete analysis again. Each version is analyzed in a separate process and the application can be configured to use multiple parallel processes on the same machine or utilize more than one machine to run analysis processes. Our current configuration has four processes on a single machine and a second machine for the databases. The analysis application also provides a query tool, accessible from its web interface, to extract the datasets to be plotted or for tabular representation. Data collection for our study is done by traversing the AST for each source code file counting the occurrences of operators (object operator, double colon, new), castings, and their expression types and storing the results to the database. The cases of un-serialized objects and reflection usage are not straightforward to count since the data type

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

[17] https://www.mongodb.org/

returned from the `unserialize` function and the reflection instances is not explicitly declared. Therefore, we have developed two algorithms for counting those cases utilizing dynamic analysis features (the corresponding techniques are presented in the appendix). A thoroughly designed set of unit tests has been constructed and used in order to ensure the correctness of the implementation of pattern detection modules. For each detection module 5 to 10 unit tests were built covering the expected patterns. Moreover, we used as test cases 1 to 3 files pulled from the actual corpus to test the implementation on actual patterns. The method invocation module that is built to collect any pattern (not a pre-defined set), revealed to us some additional patterns for the cases of -> and :: that we were not aware of beforehand. The entire test process has been automated through phpUnit test suites.

4.3.1 Classification of method invocation patterns

To gain a detailed view of method invocation pattern demographics we applied multiple classifications. Initially patterns were classified with respect to the approach used for method invocation, i.e. through an object, a static method call, the employment of function handling functions and reflection. The second classification was with respect to the reengineering ability for each approach. We classified method invocation patterns in four families depending on the degree of ambiguity with respect to the identification of dependencies among modules (reengineering). The first family is the most trivial case where the class name and method name are explicitly referenced in the corresponding statement, and we denoted them as the *Fully Defined* (FD) family. Such patterns are static invocations without any dynamic features, for example `Registry::getInstance()`, where no part of the invocation introduces ambiguity on the participants. In the *Ambiguous Owner* (AO) family we classified the patterns where the object reference is a variable and the method part is explicitly named in the statement, for example `$o->getPercent()`, which is the most common family. The complementary family is the *Ambiguous Method* (AM) family where the object part is explicitly defined but the method name is a variable, encountered in static calls like `Block::$methodName()`. Finally, we defined the *Both-Ambiguous* (BA) family with patterns where both object and method are variables, for example `$o->$method()`. Members of this family introduce the greatest degree of ambiguity in both participants of the expression and have to be further investigated. These families are graphically depicted in Table 7 (the weather symbol represents the ability to read the class and method from the invocation).

TABLE 7 METHOD INVOCATION FAMILIES

Family name		**Object/Class**	**Method**
Fully Defined	FD	☀	☀
Ambiguous Owner	AO	🌫	☀
Ambiguous Method	AM	☀	🌫
Both-Ambiguous	BA	🌫	🌫

Finally, a third classification was made dividing the patterns in two groups. The first one consists of the irregular patterns, containing PHP idioms (or more generally scripting idioms), i.e. the ones that contain variable variables or curly braces syntax. The second category, the regular patterns, contains the rest of the patterns. Following Table 2 and Table 4 notation, PHP idioms are invocation patterns constructed employing O9 and O10 (and their variants) in the left side or any of M2 and M3 (and their variants) on the right side of the expression.

4.3.2 Classification of Object instantiation patterns

We classified object instantiation patterns depending on the three approaches (as described in 3.2) and then refined the classification of those employing the new operator depending on the ability to derive the instantiated class. A second classification was made, where patterns are divided in two categories. The first one consists of the irregular patterns, containing PHP-type patterns (or more generally scripting idioms), i.e. the ones that contain variable variables or curly braces syntax. The second category, the regular patterns, contains the rest of the patterns. Following Table 5 notation, PHP idioms are the patterns I2 to I6.

4.3.3 Measuring code change

In RQ$_2$ we compare two groups of system methods, those that employ only regular patterns and those that also employ irregular (not exclusively) patterns in their body, in order to investigate which of the two method categories leads to less change prone code. We quantified code change employing the editing distance from the current version of a method to its next. The editing distance is measured employing Bergmann's PHP implementation[18] of the Unix diff utility, that reports the lines that need to be inserted and the lines that need to be deleted from string A to be identical to string B, and therefore the editing distance is equal to #insertions plus #deletions. In order to obtain the normalized value of the editing distance, it should be divided by the maximum possible value of the distance [19]. Assuming two strings A and B, completely different from each other, #insertions=|B| and #deletions=|A| (|A| and |B| correspond to the cardinality of the list containing the lines of A and B, respectively) and therefore the maximum number of operations is |A|+|B| [20]. Therefore, the normalized distance is obtained as:

$$d(A,B) = \frac{\#insertions + \#deletions}{|A| + |B|}$$

The values of d are in [0,1] where 0 corresponds to no change for the method body between two successive versions and 1 corresponds to a completely changed method (we consider a lower d as better, since it implies fewer changes—the benefits of low change proneness have been discussed in Section 1). For each system version we calculated the normalized editing distance for all methods in each group to the next version and then calculated the mean editing distance of each group:

$$mean\ Editing\ Distance\ of\ \ regulars = \ \ (sum\ of\ normalized\ editing\ distance\ of\ methods\ employing\ only\ regular\ pattern$$

$$mean\ Editing\ Distance\ of\ \ irregulars = \ \ (sum\ of\ normalized\ editing\ distance\ of\ methods\ employing\ irregular\ patterns)$$

Prior to counting the editing distance of methods and functions, the source code is normalized using Pretty-Printing as Roy and Cordy have performed in their study [21], in order to ignore details like specific variable names and operators since we are interested more in structural changes of the code. Thus we have a measure of change that is fuzzy on details and easy to use in statistical analysis. We replace variable names, unary and binary operators and casting types as described in the following table while whitespace and comments are omitted. The substitutions are summarized in Table 8.

TABLE 8 PRETTY PRINT SUBSTITUTIONS

Type	Example source	Pretty printed
Variables	$var	VARIABLE
Literals	"ads"	LITERAL
Operators	2+3	LITERAL BOPER LITERAL
Casting types	(float)$var	(TYPE)VARIABLE
Whitespace	Removed	
Comments		

4.4 Data analysis

The context of this study is open source web applications written in PHP. It can be characterized as an embedded multiple case study since each project is a case analyzed separately and each version of the project is a unit of analysis. In this section we present the collected variables and the statistical analysis that we performed per Research Question.

$RQ_{1.1}$ What is the population of method invocation patterns in PHP web applications?
Variables: The patterns of method invocations and count of occurrences of each pattern.
Analysis: Characterization of each pattern with respect to the invocation approach, characterization of each pattern with respect to reengineering and characterization of each pattern with respect to being a PHP idiom or not.

$RQ_{1.2}$ What is the population of object instantiation patterns in PHP web applications?
Variables: The patterns of object instantiation and the count of occurrences of each pattern.
Analysis: Characterization of each pattern with respect to the instantiation approach and characterization of each pattern with respect to being a PHP idiom or not. The rest of the analysis as in RQ1.1 is followed here.

$RQ_{2.1}$ Do PHP-type method invocation patterns help code to be less change prone?
Variables: The mean normalized editing distance of the methods that employ only regular patterns and the mean normalized editing distance of those that employ irregular and regular patterns.

[18] https://github.com/sebastianbergmann/diff

Analysis: In order to conclude if the irregular patterns contribute in the development of less change prone code, their editing distance should be less than the editing distance of regular patterns. The corresponding hypotheses are formulated as follows: H_o*: Regular and irregular patterns lead to the same mean change (*$\mu_1=\mu_2$*)* and H_1*: Methods employing only regular patterns change more than methods employing both regular and irregular ones (*$\mu_1>\mu_2$*).* The performed test is a single tailed Wilcoxon rank sum (a.k.a. Mann-Whitney U test or Mann-Whitney-Wilcoxon test) non-parametric test. A single tailed test is dictated from our alternative hypothesis. The assumptions for Wilcoxon's test are [22]: (a) the two samples must be drawn randomly from the target population; (b) each measurement must correspond to a different participant; and (c) the measurement scale is of ordinal or continuous type. All the requirements are met, since our corpus is drawn from a huge pool of PHP projects with no selection criteria on their coding methodology. Each method participates only once in the measurements and our measurement scale is continuous with values in [0,1]. Data are not normally distributed (common in the case of data that has many values close to zero or a natural limit [23]). If the null hypothesis is rejected then a benefit from irregular patterns employment can be assumed. Non normality was confirmed with an Anderson-Darling Test[24] with H_0: "Are data normally distributed?". All tests are performed with significance α=0.05 using the statistical software R[19].

$RQ_{2.2}$ Do PHP-type object instantiation patterns help code to be less change prone?

Variables: The mean normalized editing distance of the methods that employ only PHP-type (regular) patterns and the mean normalized editing distance of those that employ irregular and regular patterns.

Analysis: The analysis is identical to RQ 2.1.

5 RESULTS AND DISCUSSION

The results of this study will be presented and discussed separately for each research question to facilitate the understanding of the findings.

5.1 Dynamic Features Popularity

Popularity of method invocation patterns ($RQ_{1.1}$): The overall statistics for the latest version of all projects in our corpus regarding the four main method invocation patterns are presented in Table 9. As it can be observed, the vast majority of methods are invoked on objects in accordance to the principles of the object-oriented paradigm. But it has to be noted that 22% of the invocations are made through static methods; however, the percentage of static invocations varies strongly between projects.

TABLE 9 INVOCATION METHODS

Invocation method	Count	Percent (%)	Num. of projects used
Invocations on objects	316,849	77.83	10
Invocations on static methods	89,831	22.06	10
Function handling functions	359	0.09	10
Usage of Reflection	84	0.02	2

In Figure 3 the percentage of static invocations over the total number of invocations is shown for the latest version of the projects in our corpus. For example, PrestaShop has the largest proportion of static invocations. This can show the poor application of object oriented design/modeling principles and in the extreme case it can be considered as a camouflaged application of procedural programming. The use of static methods might also stem from the fact that in PHP, prior to the introduction of namespaces it wasn’t possible to group functions in modules to avoid naming conflicts, or to implement an autoloading mechanism to avoid loading huge files just in case a function is needed, or to keep the interface to the template engine simpler and make it friendlier to template designers. Therefore packing functions in to classes as static methods gives the developer the ability to create modules that can be autoloaded and avoid naming conflicts.

[19] https://www.r-project.org/

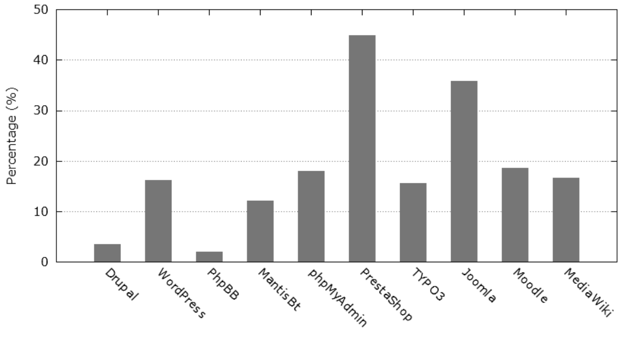

FIGURE 3 PERCENTAGE OF STATIC INVOCATIONS PER PROJECT

With respect to the ability of reengineering the dependencies between modules, in Table 10 the aggregated results for the used patterns from all projects and from their latest versions, in our corpus, are shown. Keeping up with the terminology of ecosystems, as in our previous work [16], the combination of the owner and method component is referred to as species and they are grouped in families. The total number of patterns that have been actually encountered in our corpus is 45 out of 713; so despite the vast diversity of theoretically possible patterns, programmers do not employ most of them. In the latest version of the projects shown here, 42 were encountered. It is clear that the patterns introducing the higher ambiguity have a very low share implying that for the vast majority of the cases dependencies can be extracted from source code. As expected according to usual object-oriented practice, Ambiguous Owner holds the lion's share, indicating that developers tend to use ambiguous patterns in the owner part of the expression, in contrary to Ambiguous Method that is rarely used. Fully Defined family is the second largest with only two patterns, supporting the aforementioned observation that static methods are highly used in the examined applications. The Both Ambiguous family, even though it is the third in size accounting for 0.19% of the overall population, is the most diverse family, containing 28 of the 42 employed patterns (66.6%). These observations reveal the plethora of ideas that developers come up with in order to make their code more flexible, extendible or comprehensible.

TABLE 10 PATTERN COUNT (ACCUMULATED FOR THE LAST VERSION OF ALL PROJECTS)

Species	Population	Projects	Family	Population	Species	Population	Projects	Family	Population
O1-M2	269	10	BA	790	O1-M1	241,781	10	AO	316,688
O1-M2.C1	155	10			O4-M1	36,831	10		
O4-M2.C1	86	8			O3-M1	24,849	10		
O4-M2	62	8			O2-M1	6,650	10		
O1-M2.C2	60	4			O5-M1	4,323	9		
O1-M2.C6	32	8			O7-M1	1,869	7		
O9.C1-M2.C1	28	5			O6-M1	340	6		
O4-M2.C5	17	4			O10-M1	44	3		
O1-M2.C5	10	3			O8-M1	1	1		
O2-M2.C6	10	3			S1-M2	12	4	AM	26
O3-M2	8	2			S1-M2.C6	7	1		
O2-M2	7	4			S2-M2	7	4		
O9.C1-M2	7	5			S1-M1	75,599	10	FD	89,619
O9.C1-M2.C6	6	2			S2-M1	14,020	10		
O2-M2.C1	5	2							
O5-M2.C1	5	1							
O9.C2-M2.C1	4	2							
O3-M2.C1	3	1							
O4-M2.C6	3	2							
O9.C2-M2	3	1							
O1-M2.C4	2	1							
O7-M2.C1	2	2							
O1-M2.C9	1	1							
O4-M2.C3	1	1							
O6-M2.C1	1	1							
O7-M2	1	1							
O9.C1-M2.C2	1	1							
O9.C9-M2.C6	1	1							

In Table 11 the aggregated results for the used patterns from all projects and from all their versions are shown, but this time they are classified as regular and irregular (see Section 4.3.1). The population of irregular patterns is dramatically lower than the regular patterns and the ratio is approximately 1 to 472 in favor of regular patterns (irregular patterns constitute 0.21% of the total population). Our interpretation is that developers are either not aware of those peculiar ways for invoking methods in PHP or that they do know of their existence but are reluctant to employ them because of their reduced comprehensibility.

TABLE 11 IRREGULAR VS REGULAR PATTERNS (ACCUMULATED FOR ALL VERSIONS OF ALL PROJECTS)

Species	Population	Projects	Family	Population	Species	Population	Projects	Family	Population
O1-M2	269	10	Irregular	860	O1-M1	241,781	10	Regular	406,263
O1-M2.C1	155	10			S1-M1	75,599	10		
O4-M2.C1	86	8			O4-M1	36,831	10		
O4-M2	62	8			O3-M1	24,849	10		
O1-M2.C2	60	4			S2-M1	14,020	10		
O10-M1	44	3			O2-M1	6,650	10		
O1-M2.C6	32	8			O5-M1	4,323	9		
O9.C1-M2.C1	28	5			O7-M1	1,869	7		
O4-M2.C5	17	4			O6-M1	340	6		
S1-M2	12	4			O8-M1	1	1		
O1-M2.C5	10	3							
O2-M2.C6	10	3							
O3-M2	8	2							
O2-M2	7	4							
O9.C1-M2	7	5							
S1-M2.C6	7	1							
S2-M2	7	4							
O9.C1-M2.C6	6	2							
O2-M2.C1	5	2							
O5-M2.C1	5	1							
O9.C2-M2.C1	4	2							
O3-M2.C1	3	1							
O4-M2.C6	3	2							
O9.C2-M2	3	1							
O1-M2.C4	2	1							
O7-M2.C1	2	2							
O1-M2.C9	1	1							
O4-M2.C3	1	1							
O6-M2.C1	1	1							
O7-M2	1	1							
O9.C1-M2.C2	1	1							
O9.C9-M2.C6	1	1							

Popularity of object instantiation patterns ($RQ_{1.2}$): In Table 12 the statistics on object instantiation techniques are shown for the latest versions of all the projects in our corpus. All instantiation approaches are used, but as expected the ‘new’ operator is by far the most popular one. Casting to objects is the second most popular form of obtaining an object reference, accounting for 1.87% of the cases and used in all but one project. The unserialize function is used in only 0.08% of the total object instantiations and in less than half of our corpus projects.

TABLE 12 OBJECT INSTANTIATION TECHNIQUES DEMOGRAPHICS

Instantiation method	Count	Percent (%)	Num. of projects used
New operator	37,462	98.05	10
Cast to object	715	1.87	7
Unserialize function	32	0.08	4

In Table 13 the aggregated results for the used patterns from the latest version all projects are shown. They are grouped in the three object instantiation approaches. The number of projects in which each pattern was discovered is also shown. The dominant pattern for new object instantiation is the most regular pattern (I1), accounting for approximately 95.4% of the occurrences. The rest of the patterns found in our corpus are PHP language idioms (irregular patterns) and are used for creating instances where the class to be instantiated is not predefined, implementing dynamic features of the corresponding application like loading modules defined in add-ons.

For example the second most frequent pattern I2 is found in PrestaShop's version 1.5.0.0 in their module loader. After the URI has been analyzed, the controller name that is requested is extracted. Next, that particular controller class has to be instantiated. The instantiation is performed by a Controller Factory which uses pattern I2 to instantiate the object. In Joomla version 2.5.19 in their event component, a listener is registered, not by passing an instance of the listener, but rather by passing its class name (as a string). Then, a registration function creates the object using pattern I2. There are numerous such examples in our corpus and the rest of the discovered patterns are variations of I2. The methods in which these invocations are found are acting as factory methods, enabling effortless extension of their functionality. This comfort and ease often drives application architects to ignore object orientation principles and good practices. In other words, in the aforementioned examples an interface should be implemented in order to give them a collective ontological meaning and allow them to be handled by a common factory.

TABLE 13 OBJECT INSTANTIATION PATTERNS (ACCUMULATED FOR THE LAST VERSION OF EACH PROJECT)

Species	Population	Projects	Species	Population	Projects
Family: Cast			**Family: New**		
Variable	410	7	I1	36,438	10
Array	249	6	I2	814	10
Function Call Result	25	3	I5	156	6
Array Element	22	3	I4	50	8
Property Fetch	17	4	I6	4	2
Literal Number	4	1	**Family: Unserialize**		
Static Call Result	4	1	Array element	11	2
Method Call Result	3	2			
Casted Array	1	1			
Static Property	1	1			

*Note that no occurrences of I3 and I7 were found.

Usage statistics (Table 12) show that there is no extensive use of casting since casting is not the only method to create stdClass instances. In eight of the projects in our corpus array casting to object is used, which is the most reasonable usage, to convert an associative array to an object. Since PHP allows object casting from any data type there are some irregular cases (like casting a number to an object) where this feature is used without any obvious reason. In Table 13 all discovered data source patterns used in the casting approach are shown. The population of the `unserialize` function is very low due to its strict domain of usage. Clearly the new operator is a far more popular method for creating objects. Finally, the taxonomy of the used patterns regarding to being PHP-type or not, shows that PHP-type patterns have a very low frequency compared to regular ones. There are 1,760 occurrences of PHP-type object instantiation patterns, and 36,438 occurrences of regular ones. *This leads to 4.6% of PHP-type patterns over the total. As in method invocation patterns the percentage of PHP-type patterns is rather low.*

5.2 Dynamic Features and Change Proneness

Relation of method invocation patterns and change proneness ($RQ_{2.1}$): Theoretically, the goal of using irregular patterns is to conform to the open-closed principle [1]. In other words, programmers rely on such 'exotic' patterns to develop code that is extendible and less change prone. The results of the statistical analysis performed on our corpus regarding method invocation patterns are summarized in Table 14 (U refers to Mann-Whitney U statistic). The table lists for each project the outcome of Wilcoxon's rank sum test on the difference between mean editing distance of methods employing only regular method invocations and the mean editing distance of methods employing regular and irregular invocations (see Section 4.3.3). In 9 out of the 10 projects, the methods employing irregular patterns achieve the goal of less change-prone code and the difference is statistically significant. Consequently, based on the findings of our study, the employment of irregular PHP-type patterns, although it causes ambiguity in the reengineering of dependencies between classes and often compromises the comprehensibility of code, can lead to more stable code. Thus, based on the results for RQ2.1 for this study one could claim that: *Irregular PHP-type method invocations lead to less change-prone code compared to their regular counterparts. This finding however, should be treated with caution: the lower change-proneness might be because developers are not familiar with irregular patterns of method invocation (as shown by Table 11), and thus are reluctant to change the invocations themselves as well as the surrounding code.*

TABLE 14 P-VALUES AND CONCLUSIONS FROM WILCOXON TEST

Project name	**Wilcoxon rank sum test**			
	U	**p-value**	**H_0**	**Conclusion**
Drupal	434.5	1.9e-5	Rejected	Irregulars are less change prone
Wordpress	841	3.9e-4	Rejected	Irregulars are less change prone
phpBB	110	0.014	Rejected	Irregulars are less change prone
MantisBt	210	3.9e-6	Rejected	Irregulars are less change prone
phpMyAdmin	1,623	1.02e-8	Rejected	Irregulars are less change prone
prestaShop	426	0.2887	Not Rejected	Irregulars are not less change prone

Typo3	18,288	2.2e-16	Rejected	Irregulars are less change prone
Joomla	2,581.5	9.5e-12	Rejected	Irregulars are less change prone
Moodle	15,568	2.2e-16	Rejected	Irregulars are less change prone
Wiki	14,185	2.2e-16	Rejected	Irregulars are less change prone

Relation of object instantiation patterns and change proneness ($RQ_{2.1}$): A similar approach has been employed to investigate the potential benefit from using PHP idioms in the instantiation of objects. The results are summarized in Table 15. Employment of irregular object instantiation patterns in 7 of our corpus systems has led to less change prone code. Thus, it can be concluded that even a small scale employment of irregular patterns in critical parts of the code leads to less change prone code. Thus, based on the results for RQ2.2 for this study one could claim that: *irregular PHP-type object instantiations lead to less change-prone code compared to their regular counterparts. Nevertheless, as in the case of method invocation patterns, the lower-change proneness might also be due to the lack of programmers' experience with such irregular ways of instantiating objects.*

TABLE 15 P-VALUES AND CONCLUSIONS FROM WILCOXON TEST

Project name	**Wilcoxon rank sum test**			
	U	**p-value**	**H_0**	**Conclusion**
Drupal	379.5	0.0002	Rejected	Irregulars are less change prone
Wordpress	1,101	0.0023	Rejected	Irregulars are less change prone
phpBB	97	0.0745	Not Rejected	Irregulars are not less change prone
MantisBt	584	0.0707	Not Rejected	Irregulars are lot less change prone
phpMyAdmin	9,690	8.97e-6	Rejected	Irregulars are less change prone
prestaShop	458	0.1379	Not Rejected	Irregulars are not less change prone
Typo3	21,605	1.48e-9	Rejected	Irregulars are less change prone
Joomla	2,124.5	8.8e-5	Rejected	Irregulars are less change prone
Moodle	15,804	3.76e-8	Rejected	Irregulars are less change prone
Wiki	25,595	1.07e-9	Rejected	Irregulars are less change prone

6 IMPLICATIONS TO RESEARCHERS AND PRACTITIONERS

In this section, we revisit the main outcomes of this study from the perspectives of practitioners and researchers. First, regarding practitioners, our study provides the following contributions:

- ***A comprehensive catalogue of PHP language features related to method invocation and object instantiation, including the use of dynamic features for these purposes***. This study provides a detailed list of all possible ways in which object instantiation and method invocation can be performed dynamically in PHP code. Although the study design has not focused on the particular circumstances under which developers tend to use each individual pattern, we believe that interested readers might be motivated from these findings and investigate some of the less frequently ways for invoking methods and instantiating objects. Knowing that languages such as PHP provide a rich arsenal of various alternatives for achieving a programming goal and the fact that several irregular patterns are exploited by developers of well known projects, can provide a motive for seeking further programming possibilities.
- ***Preliminary evidence that the use of PHP dynamic features related to method invocation and object instantiation leads to less change prone code***. Based on the findings of our study, the frequencies of irregular patterns for object instantiation and method invocation are quite low, compared to the traditional ones. This means that, although these patterns are known among developers (they exist at least once in the majority of the projects), they are not very frequently employed. Based on our empirical findings the source code chunks that are associated with such patterns are less change prone. Therefore, although dynamic features come with several drawbacks especially regarding understandability, they seem to offer more flexible implementation, which does not need to be frequently updated. However, this finding should be treated with caution as the actual reasons for which dynamic features lead to fewer changes has not been thoroughly investigated.

Regarding researchers, the findings of this study affirm the intuition that using such dynamic features improves code stability, and complies with previous results. However, some interesting future work opportunities have risen. First, the stability of the object instantiation and method invocation pattern instances needs to be further validated, by considering the requirements that they implement (i.e., examining the possibility that they are not employed in design hotspots). Other than that, the low number of these instances needs to be investigated so as to understand (through an explanatory study—the one performed in this paper is exploratory) when these patterns should be used, leading to safer conclusions on if their number needs to be increased, or if they should be used with caution. Focusing on the less regular patterns and the particular circumstances that drive developers to employ them can reveal interesting information regarding development practices. However, a study on these underlying reasons would be very challenging as numerous parameters might be involved, including the developer exposure to the language, the particular context in which a method invocation or object instantiation is placed and even more social factors, such as co-developers, sources from which knowledge has been acquired or overall attitude towards irregular language constructs. At this point, we need to note that despite the benefits that we highlighted in this paper, we acknowledge the fact that for novice developers who are not familiar with such PHP-idioms, the syntax of these patterns might significantly hurt source code understandability. This implication needs further verification and poses a very interesting extension for this study. Furthermore the analysis can be performed at a more fine-grained level by considering also the scope of the involved variables (method, class, global) in order to explore the impact of variable scope to the change proneness of the involved methods. Moving to some even more extreme cases, callables/anonymous functions defined within a class which might be invoked outside the class can also be the subject of an extended study on irregular pattern for function/method invocations in PHP.

Finally, in our study we have excluded libraries from the empirical analysis. Since libraries might reflect completely different programming practices, the investigation of possible differences in the use of irregular patterns between application software and libraries is also an interesting line of future research.

7 LIMITATIONS AND THREATS TO VALIDITY

Although we have strived to be as inclusive as possible, there might be some theoretical patterns regarding method invocation or object instantiation that might not be listed in this work. This stems from the fact that the proposed analysis has not analyzed the language structure to find all permitted patterns but rather is an empirical study aiming at identifying actual used patterns in open-source systems. For example, in the detection of method invocation patterns and particularly in the case of function handling functions we omitted `call_user_method` and `call_user_method_array` since they were deprecated very fast (introduced in PHP version 4.0.5 and deprecated in version 4.1.0).

One threat to the construct validity or our study stems from the fact that we might not have detected all the cast usage patterns, and some cases of function handling functions. For casting there are cases where the variable is returned from the method or function to a block outside the file before a method call is performed on it. In function handling functions there are cases where the method to be invoked is in a string variable containing a text like "`MyClass::MyMethod`" and not in a callable. It was not possible to detect such cases in this work. Also related to the external validity of the findings is the fact that the analyzed corpus of PHP projects unavoidably reflects the characteristics dictated by the time at which the corresponding projects have been developed. For example, some of the projects in the corpus require PHP 5.2 whereas features such as closures have been introduced in version 5.3. As a result, a newer dataset could probably reveal slightly different usage patterns for method invocations and object instantiations.

With respect to the observation that irregular patterns lead to less change-prone code, one threat to construct validity is that developers might be reluctant to modify irregular method invocations or object instantiations exactly because they are unaware of how they work. Nevertheless, we would like to emphasize that change proneness has been assessed at the method level. In other words, our evidence suggests that methods containing irregular patterns are less frequently changed and this concerns the entire method, not only the invocations or instantiations.

The conclusions from this paper cannot be generalized to any project written in PHP and even more for projects developed in another language since we have studied a corpus of ten projects, which is a rather small portion of the open source code ecosystem. However, the examined projects are large, not only in terms of code size but in terms of reputation in the open source community and therefore can be considered as a reasonable sample in order to draw meaningful conclusions. Also as we already mentioned, dynamic feature usage is not beneficial unconditionally. In our corpus there was a very low proportion of them and the project architects are supposed to be highly motivated to carefully design the code.

8 CONCLUSIONS

Scripting languages and PHP in particular can add several arrows to the quiver of web developers. The analysis of more than 1000 versions of ten large scale PHP web applications revealed a vast ecosystem of theoretically feasible method invocation and object instantiation patterns. However, developers tend to use a small fraction of these patterns in practice implying that practitioners familiar with this subset of patterns are well equipped for developing and maintaining long lasting and reputed open source projects. Irregular method invocation and object instantiation patterns lead to less structured and readable code hindering any attempts to reengineer existing code. Therefore, at a first glance, it appears that irregular patterns are a bad practice and developers should be encouraged to restrict their use. However, evolution analysis has shown that these exotic patterns are constantly applied and in many cases, increase over time. To shed light into the reasons that lead to the adoption of these irregular patterns we have investigated the change proneness of methods by comparing methods in successive software versions. The results indicate that methods employing irregular patterns are less change prone than methods employing regular patterns, a finding which, if validated by other studies, might imply that dynamic features can be exploited to the benefit of more stable code.

REFERENCES

[1] R. C. Martin, “More C++ Gems,” R. C. Martin, Ed. New York, NY, USA: Cambridge University Press, 2000, pp. 97–112.

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

[3] M. Hills, P. Klint, and J. Vinju, “An Empirical Study of PHP Feature Usage: A Static Analysis Perspective,” in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, New York, NY, USA, 2013, pp. 325–335.

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

[5] M. Hills, “Variable Feature Usage Patterns in PHP (T).,” in *Proceedings of the 2015 Automated Software Engineering*, 2015, pp. 563–573.

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

[7] 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 *Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering*, 2015, pp. 134–139.

[8] H. E. Van der Hoek and J. Hage, “Object-sensitive Type Analysis of PHP,” in *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, New York, NY, USA, 2015, pp. 9–20.

[9] E. Kneuss, P. Suter, and V. Kuncak, “Phantm: PHP Analyzer for Type Mismatch,” in *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, New York, NY, USA, 2010, pp. 373–374.

[10] S. Son and V. Shmatikov, “SAFERPHP: Finding Semantic Vulnerabilities in PHP Applications,” in *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, New York, NY, USA, 2011, pp. 8:1–8:13.

[11] N. Jovanovic, C. Kruegel, and E. Kirda, “Pixy: a static analysis tool for detecting Web application vulnerabilities,” in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006, pp. 6 pp. – 263.

[12] D. Hauzar and J. Kofron, “Framework for Static Analysis of PHP Applications,” in *Proceedings of the 29th European Conference on Object-Oriented Programming*, Dagstuhl, Germany, 2015, vol. 37, pp. 689–711.

[13] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto, “Is Duplicate Code More Frequently Modified Than Non-duplicate Code in Software Evolution?: An Empirical Study on Open Source Software,” in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, New York, NY, USA, 2010, pp. 73–82.

[14] J. Krinke, “Is Cloned Code More Stable than Non-cloned Code?,” in *Proceedings of the 2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, 2008, pp. 57–66.

[15] T. Yamamoto, M. Matsushita, T. Kamiya, and K. Inoue, “Measuring Similarity of Large Software Systems Based on Source Code Correspondence,” in *Proceedings of the 6th International Conference on Product Focused Software Process Improvement*, Berlin, Heidelberg, 2005, pp. 530–544.

[16] P. Kyriakakis, A. Chatzigeorgiou, A. Ampatzoglou, and S. Xinogalos, “Evolution of method invocation and object instantiation pattern in PHP ecosystem,” in *Proceedings of the 20th Pan-Hellenic Conference on Informatics*, 2016.

[17] V. R. Basili, “Software modeling and measurement: the Goal/Question/Metric paradigm,” Techreport UMIACS TR-92-96, University of Maryland at College Park, College Park, MD, USA, 1992.

[18] J. Tulach, *Practical API Design: Confessions of a Java Framework Architect*. Apress, 2008.

[19] N. Tsantalis and A. Chatzigeorgiou, “Ranking Refactoring Suggestions Based on Historical Volatility,” in *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, 2011, pp. 25–34.

[20] G. Navarro, “A Guided Tour to Approximate String Matching,” *ACM Comput Surv*, vol. 33, no. 1, pp. 31–88, Mar. 2001.

[21] C. K. Roy and J. R. Cordy, “NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization,” in *Proceedings of the 16th IEEE International Conference on Program Comprehension*, 2008, pp. 172–181.

[22] N. Nachar, “The Mann-Whitney U: A Test for Assessing Whether Two Independent Samples Come from the Same Distribution,” *Tutor. Quant. Methods Psychol.*, vol. 4, no. 1, pp. 13–20, 2008.

[23] A. M. Joglekar, *Statistical methods for six sigma: in R&D and manufacturing*. John Wiley & Sons, 2003.

[24] N. M. Razali and Y. B. Wah, “Power comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling tests,” *J. Stat. Model. Anal.*, vol. 2, no. 1, pp. 21–33, 2011.

Appendix

DETECTION ALGORITHMS

In the following we present the detection algorithms for un-serialization and reflection usage in method invocations.

Unserialized object detection.

The result of an `unserialize` function can be a boolean, float, integer, string, array or object. In order to discover if the returned value is an object we make the assumption that the function's return value will be assigned to a variable and an object operator (->) will be applied on that variable. In the case of a local variable we track starting from the assignment point to the end of the function or method if the object operator is applied. In case of a class property we apply their use tracking at class level, we track object operator usage on this property to all methods of the class.

We employed a simple symbol table for tracking variables assigning them a tag to mark their state, if they are used as objects, if they hold an `unserialize` result and finally if they are discovered as un-serialized objects. Traversing the AST when an assignment of an `unserialize` function return value to a local variable or a property of the class is detected, we add the variable to the symbol table with tag `unserialize`. When an object operator is used on a variable, we check if the variable exists in the symbol table with the tag `unserialize`. If it exists, then we mark the variable as `discovered`. When the object operator is used on a variable that is not in the symbol table we add the variable to the table with the tag "object", so if `unserialize` result is assigned to it at some point will be turned to `discovered`. This way we handle class properties hence their use can be found in the code, in another method, before the `unserialize` usage. The algorithm to detect the objects instantiated using the `unserialize` function is shown in Listing 1.

```
if object operator used on class property then
        if property in symbol table with tag 'unserialize' then
                count +1
                change tag to 'discovered'
        else
                add to symbol table with tag='object'
        endif
endif
if object operator used on local variable then
        if variable in symbol table with tag ='unserialize' then
                count+1
                change tag to 'discovered'
        endif
endif
if unserialize used and return value assigned to variable or property then
        add variable or property to symbol table with tag='unserialize'
endif
```

LISTING 1 UNSERIALIZED OBJECT DETECTION

Method invocation using reflection

Detecting reflection use for method invocation cannot be done *directly* with AST traversal hence the variables holding a class' reflection are not declared to be reflection class, as like in Java, therefore static analysis has to be extended with dynamic aspects. We used a set of use cases to define the rules to build a tracking algorithm for method invocations with reflection.

```
$reflectionMethod = new ReflectionMethod('HelloWorld', 'sayHelloTo');
echo $reflectionMethod->invoke(new HelloWorld(), 'Mike');
```

LISTING 2 REFLECTIONMETHOD CLASS USAGE

The simplest way to make a method invocation is to use the `ReflectionMethod` class, as in Listing 2. Constructing a `ReflectionMethod` object with arguments, class and method, the result is the reflection of the method only. Then the method can be invoked calling the invoke method with the object reference and any possible arguments of the method. In the example a method reflection is created for the method `'sayHelloTo'` of the class `'HelloWorld'`. In the second line the method is invoked on the newly created object with parameter 'Mike'.

Again we employed a simple symbol table to track the types of the variables to which `ReflectionClass` or `ReflectionMethod` objects are assigned so when the invoke method is invoked we can determine if it is a call on a `ReflectionMethod` object. The goal is not to do sound type analysis but trace the flow of Reflection classes references. Only variables with instances of `ReflectionMethod` are added to the symbol table. Therefore when an object operator node is found a lookup is made to the symbol table if the variable is present. If so, and the method name is "invoke" then an occurrence of method invocation is counted.

```
 1  function trigger($event, $args) {
 2      foreach ($this->_listeners as $listener) {
 3          $reflector = new ReflectionClass(get_class($listener));
 4          $methods = $reflector->getMethods();
 5          foreach ($methods as $method) {
 6              if ('on' . $event == $method->getName()) {
 7                  $method->invoke($listener, $args);
 8              }
 9          }
10      }
11  }
```

Listing 3 ReflectionClass usage

In Listing 3 the usage of `ReflectionClass` is shown. This is a fragment of an event dispatcher. There are some listeners registered and this method is invoked in order to call the `on`*`Event`* (where *Event* is an event name) method of any listeners that have such a method. The dispatcher loops over the registered listeners and for each one creates a `ReflectionClass` (line 3) that has the reflection of the class not just a method. Then, the dispatcher performs a search for methods that match the naming pattern and invokes them. This is accomplished by getting an array of `ReflectionMethod` objects (line 4) and in lines 5-9, iterate over the array `$methods` with a `foreach` statement.

Our algorithm assumes that the $methods variable is of `ReflectionMethod` type and so is the variable `$method` that array elements assigned to `$method` in the `foreach` statement. So in line 7 the invocation of `invoke` method on `$method` variable, signals a usage of reflection for method invocation, hence the variable `$method` is in the symbol table with type `ReflectionMethod`.

```
1  $reflector = new ReflectionClass($className);
2  if( $reflector->hasMethod($methodName) ) {
3         $reflection->getMethod($methodName)->invoke(null);
4  }
```

Listing 4 Invocation of static method

The last use case is shown in Listing 4 where two new elements are added. The first element is the chained invocation in line 3. There is no assignment to a variable here, therefore our algorithm has to follow chained invocations as well. To accomplish this goal, we employed an expression stack. In the stack any known results of expressions are pushed and if an assignment follows, the variable is tagged with that type, or if an invocation is done, the expression stack will be checked to test whether the result of the previous part of the expression is a reference to an object of interest. In our case (line 3), first the invocation of `getMethod` on `$reflection is` processed. Since the variable `$reflection` in the symbol table is tagged as `ReflectionClass,` the result `ReflectionMethod` is pushed on the stack. Then the second object operator is processed. The object part is the result in the stack and the method part is "invoke". The last value is popped from the expression stack. If the value is tagged as `ReflectionMethod` then it is counted as an occurrence of method invocation with reflection.

The second element in the use case, is in the use of `ReflectionMethod`'s invoke method. The first argument, which is the object whose method will be invoked, is null. In this case a static call is made to the reflected class. Therefore, adding this identification, our algorithm can also distinguish occurrences of method invocations on objects and static calls.