Maintenance Patterns of large-scale PHP Web Applications

Panos Kyriakakis
School of Science and Technology
Hellenic Open University
Patras, Greece
panos@salix.gr

Alexander Chatzigeorgiou
Department of Applied Informatics
University of Macedonia
Thessaloniki, Greece
achat@uom.gr

***Abstract*—Scripting languages such as PHP have been criticized as inadequate for supporting maintenance of large-scale software projects. In this paper we attempt to provide insight into the way that five large and well-known PHP applications evolved over time. Several aspects of their history are examined including the amount of unused code, the removal of functions, the use of libraries, the stability of their interfaces, the migration to object-orientation and the evolution of complexity. The results suggest that these systems undergo systematic maintenance which is driven by targeted design decisions and evolution is by no means hindered by the underlying programming language.**

Keywords—software evolution; web applications; survival analysis; software libraries; PHP; scripting language

I. INTRODUCTION

Various anecdotal sources in computer science claimed for long that despite the tremendous popularity of scripting languages [2], such as those employed in LAMP (Linux-Apache-MySQL – Perl/Python/PHP), are not suitable for proper and professional software engineering [10]. In other words, the proponents of traditional compiled languages such as Java and C++ claimed that software projects based on scripting languages lack the architectural properties that allow systematic, effortless and viable maintenance.

Such attacks to the scripting languages are less frequently documented in scientific papers, although the academic community usually tends to reject the change in programming practices brought about by scripting [10]. This skepticism is also reflected by the fact that in most academic institutions around the world, computing curricula do not rely on scripting or dynamic languages for their CS101 course. The number of empirical studies in the software engineering community on projects built with typeless scripting languages is also significantly smaller than that of strongly-typed, 'system programming' languages.

On the other hand, evidence suggests that scripting languages enhance programmer productivity [12]. Prechelt [14] presented results according to which implementation times for programs written in scripting languages, such as Perl, Python, Rexx, and Tcl, were about one-half of the time required to implement the same functionality in C/C++/Java. The adoption of scripting languages by software practitioners is also reflected in the increased penetration to open-source development. In Sourceforge[1] PHP's project count is in the third place after Java (52,234 projects) and C++ (42,081 projects) numbering 33,259 projects and overcoming C counting 31,194 projects.

In this paper we present an empirical study on five large open-source web applications implemented with the popular scripting language PHP, to investigate the evolution of web applications regarding their maturity, quality and adoption of the object oriented paradigm. We have examined several aspects of software evolution that might provide hints as to whether good practices in development and management have been followed: The existence of dead/unused code in any software system is a burden consuming resources and posing threats to maintainability. We examine the presence and survivability of unused code as a means of detecting architectural changes in the history of the examined systems. In scripting languages a major source of unused code is the employment of third party libraries, which at the same time is an accepted good practice in software development and a possible indication of maturity [1], [2]. In this context, we investigated the amount of library code being used over time in each system. Another factor implying software maturity is the stability of the corresponding APIs, and therefore, we have also examined six classes of possible API changes. Moreover, we investigated the migration of the analyzed projects to the object oriented paradigm as well as the evolution of their complexity.

The rest of the paper is organized as follows: In Section II we introduce the web applications that have been analyzed, while in Section III we discuss issues and challenges related to the analysis of the examined versions. Results on each of the investigated aspects of software evolution are presented and discussed in Section IV. Threats to validity are summarized in section V. Related work on similar efforts for analyzing software systems and previous work on scripting languages is presented in Section VI. Finally, we conclude in Section VII.

II. APPLICATIONS

The software systems used in the case study have been selected according to the following criteria:

[1] http://sourceforge.net

1063-6773/14 $31.00 © 2014 IEEE
DOI 10.1109/ICSME.2014.60

- they should be well known projects with established reputation in the open source community.
- they should have started in PHP versions prior to version 5. The reason for this choice is that despite the introduction of object-orientation in PHP 4, prior to version 5 the support was quite limited (e.g. there were no scope modifiers).
- they should have their code available in GitHub.
- they should have at least 30 unique tags in GitHub.

Our major concern was to select acknowledged projects with a long history, large number of committers and even larger number of users. According to Samoladas et al. [15] the majority of open-source projects are abandoned after a short time period, rendering them inappropriate for systematic analysis of programming and maintenance habits.

The case study has been conducted on the following five open source projects implemented in PHP:

1) **Wordpess**[2]. The most popular blogging software; it has a vast community of both contributors and active users.
2) **Drupal**[3]. One of the most advanced CMS (Content Management System). It is also characterized by a large and active community.
3) **PhpBB**[4]. One of the most widely used forum software.
4) **MantisBt**[5]. Probably the most popular bug tracking application written in PHP.
5) **PhpMyAdmin**[6]. The well-known MySQL administration tool.

The abovementioned software systems are to a large extent community driven and could be characterized as the founding projects of web application development (considering the PHP as programming language). They have set the standards and powered most of the web content created in the last decade.

The fact that the examined projects have an enormous code base and numerous user plug-ins dependent upon them, implies that backward compatibility should never be broken. Due to the projects long existence there are many versions available. In Table I we show some statistics about the selected projects. Cumulatively, we have studied 390 official releases aggregating to 50 years of software evolution.

TABLE I. RELEASE STATISTICS FOR THE EXAMINED PROJECTS

Project	Years	First		Last		Number of Releases
		Release	Year	Release	Year	
WordPress	9	1.5	2005	3.6.1	2013	71
Drupal	12	4.0.0	2002	7.23	2013	120
phpBB	12	2.0.0	2002	3.0.12	2013	37
MantisBt	8	1.0.0	2006	1.2.15	2013	33
phpMyAdmin	9	2.9.0	2006	4.1.6	2014	129

In the next table we outline the growth of basic size measures. For the first and last release that has been examined, the size of the source code in KLOCs and the size in code blocks (number of functions and class methods) is shown.

2 http://wordpress.org/
3 https://drupal.org/
4 https://www.phpbb.com/
5 http://www.mantisbt.org/
6 http://www.phpmyadmin.net/

TABLE II. SIZE MEASURES FOR THE FIRST AND LAST RELEASE OF THE EXAMINED PROJECTS

Project	First			Last		
	Release	KLOC	Code Blocks	Release	KLOC	Code Blocks
WordPress	1.5	20	763	3.6.1	200	5154
Drupal	4.0.0	14	692	7.23	173	5140
phpBB	2.0.0	18	256	3.0.12	194	2389
MantisBt	1.0.0	68	2447	1.2.15	165	4904
phpMyAdmin	2.9.0	39	693	4.1.6	248	5343

The size growth reflected on the lines of code confirms Lehman's 6th law of software evolution [8] which stipulates that programs grow over time to accommodate pressure for change and satisfy an increasing set of requirements. As proposed by Xie et al. [21] the number of definitions can provide an accurate indicator to study the size growth. Since in PHP the only clearly defined structures are functions and class methods, we measured their total number (designated as number of code blocks). Lehman's 6th law is also vividly confirmed in this context.

These systems have their roots back before PHP fully supported object orientation and this adds another point of interest because they combine procedural code and classes. The poor support of object orientation in PHP to versions prior to 5.3 lead to that coding style. Due to the mix of procedural and object programming we do not differentiate between functions and class methods in the applied survival analysis.

III. THE APPROACH

Each application has been analyzed with a software tool that we developed for conducting source code analysis and data collection in a uniform way (Section III.E). Available versions have been downloaded from GitHub and then static analysis of the source code has been performed to identify function and method signatures, class definitions and function usage data for the each version. The extracted information has been stored to a database. Using the stored data we estimated the survival function regarding unused and removed functions. Moreover, we analyzed library usage and the degree of conformance to the object-oriented paradigm by measuring the ratio of class methods over the total number of functions/methods. Finally, we investigated the stability of function signatures as well as the evolution of complexity. Next we discuss some key points in our analysis.

A. Challenges in the identification of unused code

In the first part of this study we focus on unused functions and class methods i.e. functions which are not invoked by other system functions or methods (i.e. FanIn=0). We have deliberately avoided using the term 'dead code' as this usually includes non-reachable code blocks, whereas our analysis focuses only on functions which are not invoked. PHP is a highly dynamic language and functions might be called in various ways that static analysis as performed by currently available tools cannot identify. These cases include:

- calls employing the Reflection API[7],

7 http://www.php.net/manual/en/book.reflection.php

- `call_user_func()`[8] and `call_user_func_array()`[9] calls
- method invocations using the `new` operator with variable class names
- Variable function names for static method calls such as `$class::method()`
- variable function or method names such as `$function()` or `$object->$method()`.
- Automatic calls to methods such as `__toString()`[10] or `Iterator::*()`[11]

For the cases where the function name is not a literal, it may even be retrieved from database data. Functions can also be invoked by extracting their name from the URL that users enter in their web browser. Especially in applications that employ the single front controller paradigm, which acts as the unique entry point for an entire application, a large number of controller actions might seem to be unused code. Thus, the identification of unused code poses significant challenges and can hardly be accurate. We refer to this type of calls as *stealth calls*, and since they cannot be detected by static analysis tools they are inevitably considered as unused code, posing a threat to the analysis.

Another complication related to the aforementioned limitations is the widely used practice of *hooks*. Hooks refer to a functional implementation of the observer pattern that provides an extension mechanism to functions. The simplest implementation of a hook (Figure 1) is to have a global array holding the registered hooks, a function to make hook registrations (e.g. *add_action*) and finally a function to call the corresponding function handlers for a given hook (e.g. *do_action*). Thus, for a function that we want to extend, a call to *do_action* can be made, with parameter a key for the desired hook (e.g. *'my_hook'*). That last call is made using PHP's `call_user_func()` function, whose first argument is a literal containing the desired function name. The use of hooks is clearly not traceable with general purpose static analysis tools.

Many projects employ hooks not only as an extension point for third parties, but they also implement core functionality in this way. For example, a significant portion of the core functionality in WordPress takes advantage of the hooking system. Additionally, plug-ins implemented in the examined systems systematically take advantage of the hooking mechanism.

The ideal solution to determine all actual function invocations is to examine the code manually, a process that is however infeasible even for medium sized applications. In our study we have used Bergmann's phpDCD[12] tool to analyze the source code and extract unused code. To the best of our knowledge, phpDCD is one of the most reliable dead/unused code detectors for PHP, employing static analysis. However, the limitations that have been discussed are valid.

[8] http://www.php.net/manual/en/function.call-user-func.php
[9] http://www.php.net/manual/en/function.call-user-func-array.php
[10] http://www.php.net/manual/en/language.oop5.magic.php
[11] http://www.php.net/manual/en/class.iterator.php
[12] https://github.com/sebastianbergmann/phpdcd

```php
/* global array of registered hooks */
$hooks = array();
/* function for registering hooks */
function add_action($hook, $funcName) {
  global $hooks;
  if( !isset($hooks[$hook]) ) {
    $hooks[$hook] = array();
  }
  $hooks[$hook][] = $funcName;
}
/* function for executing a particular hook */
function do_action($hook) {
  global $hooks;
  foreach($hooks[$hook] as $fn) {
    call_user_func($fn);
  }
}
/* extensible function */
function myFunc() {
  /* ... function code ... */
  do_action('my_hook');
}
/* extension */
function myHookFunc() {
}
/* registering a hook function */
add_hook('my_hook','myHookFunc');
```

Figure 1 Simplest implementation of hooks

B. *Survival Analysis*

Survival analysis models the time it takes for events to occur and focuses on the distribution of the survival times. It has been applied in many fields ranging from estimation of time between failures for mechanical components, lifetime of light bulbs, duration of unemployment in economics, time until infection in health sciences and so on. In each context of application it is necessary to provide an unambiguous definition of the termination event.

Survival can by modeled by employing an appropriate survival function. Non-parametric survival analysis does not assume any underlying distribution for survival times. The most common non-parametric analysis is Kaplan-Meier [6], [13] where the survival function, illustrated graphically by the corresponding Kaplan-Meier curve, refers to the probability of an arbitrary subject in a population to survive t units of time from the time it has been introduced.

Classical Kaplan-Meier analysis treats all subjects as if they existed in the beginning of the study and therefore does not allow the mapping of drops in the survival probability with particular time points (releases in our case). To enable a more detailed analysis of software evolution we also employ charts illustrating the survival function (rather than the cumulative probability to survive from time zero to t_i) as follows: Restricting observation to the discrete time points when termination events occur t_1, t_2, t_3, …, t_n, we define r_1,... ,r_n to be the number of subjects at risk and d_1, d_2,..., d_n to be the number of events at these time points. The probability of surviving from zero to t_1 is estimated by $S(t_1)=1-d_1/r_1$, where d_1/r_1 is the estimated proportion of subjects undergoing the termination event in that interval. The probability of surviving from t_1 to t_2 is given by $S(t_2)=1-d_2/r_2$. Thus, the survival function is equal to unity minus the hazard ratio d_i/r_i:

$$S(t_i)=1-d_i/r_i$$

In contrast to most studies that employ survival analysis, where the initial population is not enhanced by new entries, in our case, we follow the approach by Pollock et al [13] and Scaniello [16]. In particular, new code blocks added in each version are treated as 'staggered' entries [13] and are added to the existing population of functions which are "at risk".

C. Survival regarding function usage

To apply the aforementioned survival analysis on the usage of functions, the termination event refers to the time point at which a function "becomes unused". A function or method (for the sake of simplicity we will refer to both as functions) becomes unused when there are no observable calls to that function. The assumption that has been used, similarly to Scaniello [16], is that once a function becomes unused in a version, it remains unused to the end of the observation.

In order to calculate the survival function the following mapping is proposed. The discrete time points of observation t_i are the consecutive versions of the project under study. In each version, the functions and methods existing in that version are said to be at risk, and the number is denoted as r_i, excluding those that were marked as terminated in previous versions. Finally, d_i indicates the number of functions and methods that become unused at version i.

Drupal and PhpMyAdmin contain a significant amount of unit tests indicating the effort that is made towards quality assurance; however, unit tests should not be counted as the projects' functional code. Unit tests are invoked from external tools, like phpUnit[13], and therefore their construction forces them to appear as unused code in the project itself. Therefore, unit tests are excluded from the survival analysis.

Moreover, we make the assumption that all functions and methods can be characterized by the same survival function regardless of the time they were added to the project. In other words we assume that all functions can be characterized by the same probability of becoming unused.

D. Survival regarding function removal

In subsection III.C we presented our approach for analyzing the survival of functions and methods considering as termination event the time point at which they become unused. In this subsection we focus the survival analysis on the removal of functions. In other words, we consider as termination event the time point at which a function is removed from the system. The reason for performing the analysis at the function level (i.e. excluding class methods) is that the systems under study are mainly functional and their public API is mostly functional.

Moreover, since PHP does not provide visibility modifiers to functions, the identification of the public API is not possible through source code analysis. Developers should consult the documentation in order to retrieve the subset of functions belonging to the public API. However, updates to the documentation are not performed systematically and as a result API documentation matching the official releases is not always reliable. Survival analysis regarding function removal enables us to obtain an insight into the evolution of the public API given that functions are by definition public.

The mapping for estimating the survival function with respect to the removal of functions is similar. The discrete time points of observation t_i are the consecutive versions of the project under study. At each version, r_i represents the number of functions being at risk including the functions that survived from the previous version and the newly introduced ones. The termination event is defined as the removal of a function, meaning that the function signature is not defined in the project any more. Thus, d_i refers to the number of functions that are removed at version i.

E. Data collection and presentation software

A software tool has been developed to automate the analysis. The tool has been implemented in PHP on top of the Symfony[14] framework. It has a web interface in order to add projects for analysis and enable viewing of the results. The backend of the tool runs several steps in order to collect and analyze the projects. Initially it downloads from GitHub all versions of the project's source code, runs pDepend[15] to retrieve basic file information and elementary metrics. Then, it runs and imports the phpDCD tool to collect unused code data. Source code parsing and abstract syntax tree analysis follows to obtain function and method signatures. The aforementioned data are stored in a database for each version of the project. Using the stored data, the various analyses discussed in this study are performed. Results are accessible from the "public" section of the tool's web interface[16].

IV. Results and Discussion

A. Survival regarding function usage

Kaplan-Meier curves illustrating the percentage of system functions surviving over successive software versions are shown in Figure 2 for all examined projects (the horizontal axis corresponds to the number of versions).

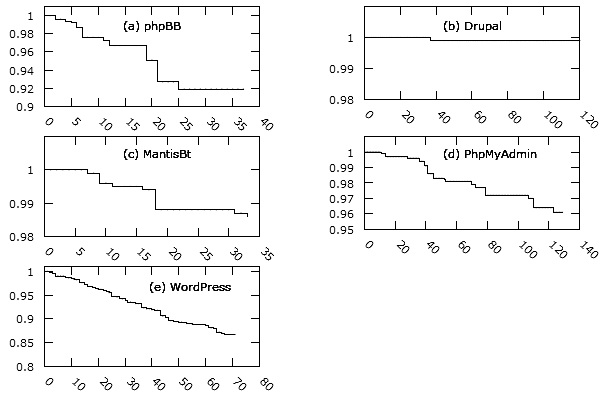

Figure 2 Kaplan-Meier survival plots regarding function usage

As it can be observed, for Drupal and MantisBt, the vast majority of functions tend to remain used. For example, in

[13] http://phpunit.de/

[14] http://symfony.com

[15] http://pdepend.org/

[16] http://snf-451681.vm.okeanos.grnet.gr/

MantisBt, after 25 successive versions, almost 99% of the initial number of functions are still invoked by other functions in the project. For the other three systems, a non-negligible percentage of functions is gradually becoming unused.

To provide further insight into the usage of functions and allow a mapping to the corresponding releases, we plot the survival function $S(t)$ for each examined project in Figure 3. The horizontal axis represents the consecutive versions and the vertical axis the value of the survival function. The results for each project are examined separately along with a discussion of the major findings. In contrast to the previous Kaplan-Meier curves, in these plots containing staggered entries, we can also observe the impact of functions being introduced in one version and not being used thereafter. Moreover, library code is included in the analysis to investigate its effect.

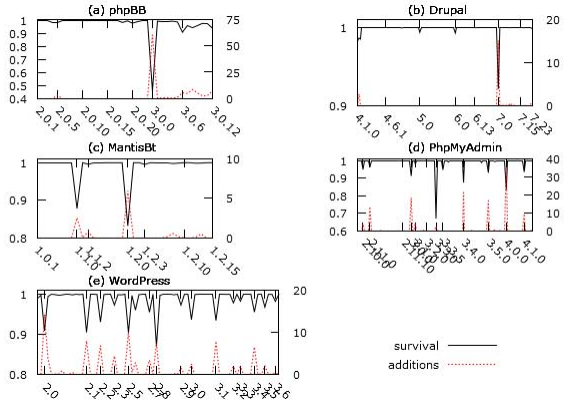

Figure 3 Survival functions regarding function usage and method additions. Horizontal axis represents the consecutive versions. Left vertical axis represents the value of the survival function, right vertical axis represents the percentage of method additions.

The plots for phpBB and Drupal share the same characteristics, since their survival function remains constant for a long time period and then exhibits a significant drop at a particular point of their evolution. For phpBB there is a large drop in version 3.0.0 and for Drupal in version 7.0. According to manual examination of their source code and their documentation, both projects underwent major changes in the corresponding version. Both projects extensively employed the notion of hooks, not only as an extension mechanism, but also for the implementation of the core functionality. Additionally, in these versions object-orientation literally invaded those projects.

In the other three projects, in the versions where the survival function drops, after source code examination we found that employment of vendor libraries took place. For example, MantisBt uses ADOdb[17] library, a widely used database layer, and in both cases (versions 1.1.0 and 1.2.0) there was an update of that library. Also in 1.1.0 the development team introduced a SOAP API to the application. For this reason the developers added the nuSoap[18] library, a very widely used library implementing SOAP server and client protocols. In version 1.2.0 they also added the ezc[19] library, a library of general purpose components.

In WordPress according to the change log and blog entries announcing version 2.7, enhancements have been so important that it could be tagged as a major release, namely 3.0. Despite these major changes the drop in the value of the survival function in version 2.7 is rather small. In the next major version 2.8, the drop of the survival function is larger; however, according to the announcement blog entry the enhancements have been less important. The question that arises is why less important changes caused a larger change in the survival function. A closer look at the unused functions in version 2.8 provides further insight. The authors added the SimplePie[20] library for RSS feed consumption. A building block has contributed to more than 50% of that version's unused code. Despite the major enchantments in version 2.7 no building blocks were added. This supports our conclusion that in web applications the key contributor to unused code are the used building blocks and that developers pay attention to keep the code clean from unused code.

In scripting languages like PHP, using a third party library, implemented also in PHP, means that the library's code has to be added to the application's code. There is no binding of binaries as in the case of .jar files in Java. Unavoidably, using a proportion of the library's functionality leads the rest of the library code to remain unused.

Along with phpBB and Drupal, WordPress also employs hooks as an extension mechanism to the public API, and additionally uses it to implement core functionality. Its API is also constantly growing due to the addition of stealth calls resulting in regular drops in the survival function.

To summarize, four of the projects introduced code appearing as unused due to the implementation of hooks and three out of five, introduced a high percentage of unused code, due to the incorporation of third party libraries.

Another point of interest is related to the versions where drops in the survival function appear. In general, drops in the survival function coincide with project's major versions. For example, in phpMyAdmin, all eleven drops of the survival curve coincide with the project's major versions. This implies the maintenance strategy that has been used. New code, thus new features, is added in major versions and in minor versions only bug fixing is performed.

To illustrate this form of evolution, we tracked function additions in the sample. In Figure 3 we show the additions of functions/methods made in each version as a percentage of the number of total functions/methods that were employed in the previous version. A stalactite-stalagmite phenomenon is evident: new code is added in major versions, where the introduction of unused code is taking place as well.

Finally, we calculated the survival separately for functions and class methods for MantisBt, PhpMyAdmin and WordPress. Methods exhibit higher drops (implying that

[17] http://adodb.sourceforge.net/

[18] http://nusoap.sourceforge.net/

[19] http://ezcomponents.org/

[20] http://simplepie.org/

functions have a higher survival probability than methods) conforming to findings comparing C and C++ [18].

From the aforementioned observations it can be concluded that in the examined large-scale PHP projects a rather "rich" form of continuous maintenance is taking place involving the incorporation of external libraries and the addition of new code that takes advantages of the new libraries. In case scripting languages were not suitable for evolving software, these kinds of changes would be scarce and degrading over time, which is not the case in the examined systems.

B. *Survival regarding function removal*

The focus of the analysis now changes, as the terminating event is the removal of a function (rather than the ending of its usage). Kaplan-Meier survival plots are shown in Figure 4. In three out of the five projects a major drop can be observed at a particular point in their history, implying the removal of a large percentage of their functions. For Drupal and Wordpress the drops are rather regular and less abrupt.

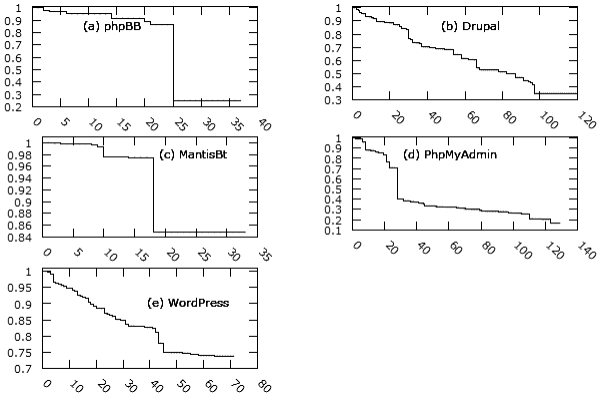

Figure 4 Kaplan-Meier survival plots regarding function removal

As in the previous Section, to obtain a better insight into the corresponding phenomena, we show in Figure 5, plots of survival functions for all projects, including library code, as well as the percentage of added methods over the total number of functions of each project. The results will be discussed according to the pattern of the survival curve in relation to the method additions.

MantisBt and phpBB exhibit the same pattern. There is one hotspot, where the survival function drops and a significant population of methods is added. The hotspot for MantisBt is version 1.2.0, and for phpBB version 3.0.0. After a manual inspection of the changes we found that, approximately 62% and 45% respectively, of the removed functions were replaced with class methods with the same functionality. For example, in MantisBt the developers moved wiki integration, upgrade module and graphs module into object oriented, enumeration functions to the ENUM class and part of the bug_api to the BugData class. In phpBB they moved the bbcode parser, session handling and search functions into classes. The remaining removals are due to functionality modifications and architectural changes. For example, in phpBB, the database upgrade module was entirely rewritten resulting in the removal of numerous functions.

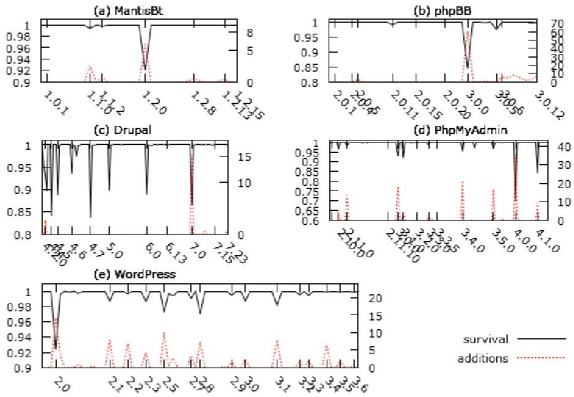

Figure 5. Survival functions regarding function removal and method additions. Left vertical axis represents the value of the survival function, right vertical axis represents the percentage of method additions and horizontal axis the consecutive versions.

PhpMyAdmin, WordPress and Drupal do not exhibit a single hotspot but function removals are distributed over almost all major releases. Concentrated additions of class methods in major releases occurred only in PhpMyAdmin and WordPress. As already mentioned, Drupal adopted object orientation in version 7.0 and since that version only a limited number of classes have been added while no functions have been removed thereafter. So in Drupal only internal functionality improvements via function rewrites is the source of function removals in all versions, but version 7.0. In that version, their database layer was completely replaced by a new object oriented database layer, substituting 70 functions with over 450 class methods.

Despite the continuous additions of class methods in Wordpress, after manual inspection we found that only in version 2.8 there was a migration to object orientation, where the majority of the removed functions have been replaced with class methods that have the same functionality. As in Drupal, in the rest of the cases where function removal is high, the reason is the rewriting of the same functionality employing the procedural paradigm.

In phpMyAdmin there are two versions with a higher change in the survival function, versions 4.0.0 and 4.1.0, indicating that major changes took place. A manual review of the source code confirmed our speculation. In version 4.0.0 267 functions have been removed, of which 76% were moved to classes. For example, the transformations module, auth module and export module were rewritten as object oriented plug-ins. A set of 77 utility functions (common.lib.php) were packed to a class (Util.class.php) and similarly a set of functions used to display query results were packed to a class (DisplayResults.class.php). In version 4.1.0 rewriting in the same manner took place. Functions in the same context were packed to classes, as for example, validation functions which were moved to a Validator class and database interface functions which were moved to the DatabaseInterface class. Finally, a large set of functions implementing the dbi (DataBaseInterface) library have been refactored to classes. Cumulatively, 77% of the removed functions migrated to classes.

An observation made during the inspection of the source code is that a number of functions reported as removed were renamed due to the change of coding standards. The initial naming convention for function names was that names should be snake case[21] following PHP's conventions, but the late trend, as also proposed in PSR-1[22] standard, in PHP is camel case[23], so the developers changed the project's coding standard applying camel case to function and method names.

To summarize, three of the five projects are gradually migrating their functions to class methods, and thus function removal is justified by this fact. The other two projects, Drupal and Wordpress, are employing object orientation for the implementation of new features and removed functions are replaced with new function implementations. One possible interpretation is the plethora of user contributed plug-ins and themes to Drupal and Wordpress and as a result, breaking compatibility is not an option. At the time of writing, Wordpress' download site contained almost 30,000 plug-ins and more than 2,000 themes. For Drupal, there are more than 8,000 modules and almost 600 themes. Most of them are user contributed and distributed under an open source license.

C. Library usage

PHP is a rather new programming language and according to TIOBE[24] has gained popularity during the last decade. An indirect indication of maturity for a given programming language is the development of third party libraries and the employment of them in other projects. In three out of the five projects in our study we have observed a strong trend in using such libraries. As Tulach [19] observes, the trend in modern software development is the use of such pre-made building blocks in order to ease and speed up the development of applications. As we have shown, a side effect is the introduction of unused code blocks, due to the scripting nature of the language. However, the fact that the library's source code becomes part of the system's source code, enables us to measure the ratio of library code over system code, something that is not straightforward with compiled languages.

In Figure 6 the plots show the used library code over the system code of each application (please note that the y-axis range is not the same across all projects). PhpBB and Drupal appear to employ a limited portion of third party libraries in their code (less than 2% and 6%, respectively). Although there are add-ons for these systems that use libraries, the core project seems to be free of third party library code. Drupal gradually replaced any third party libraries with in-house code trying to build its own ecosystem and moving from being just a CMS to be a CMF (Content Management Framework) providing all required components. This strategy of development allowed all code to be delivered under a single license, since some of the initial libraries were published under different licenses. PhpBB on the other hand in an effort to address user feature requests as quickly as possible and with a developer community that is not as active as Drupal's,

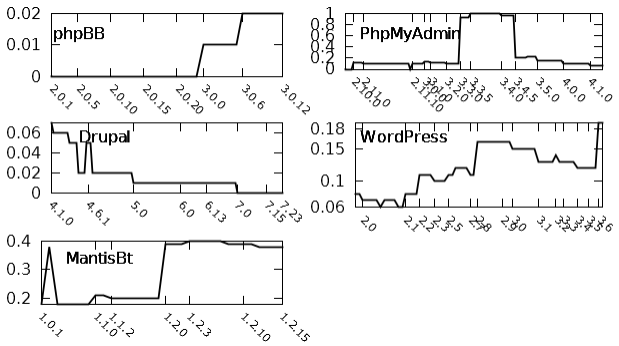

Figure 6 Used Library code over system code

started employing third party libraries at a very slow pace. However, it is worth observing that the trend is increasing.

The other three systems are gradually introducing library code (the percentage of library code is significantly larger). An exceptional case is phpMyAdmin where for a small period the library code was equal to system code. In this project, developers updated in version 3.3.0 the TCPDF library to its latest version and added PHPExcel[25] library to support import and export functionality from Microsoft Excel files. PHPExcel is a vast and complete library that gained quickly the respect of the community. Those two additions exploded the ratio to 1. But in version 3.4.5 the developers eliminated Excel files import/export support and removed the PHPExcel library. The removal of such a huge library dropped the ratio to 0.2.

By observing the variations and the extent of library usage (but also the reason of not using them in some cases) makes evident that developers of PHP applications are systematically updating their projects. In other words, maintenance of these systems is not simply restricted to additions of functionality in order to address clients' requests, but includes major architectural decisions such as the choice to rely on a particular library or not.

D. Interface stability

The stability of an interface can be characterized by the number and types of changes to the functions' signatures. According to the strict PHP definition, a function signature is only the name of the function, but this does not reflect the interface correctly, since no parameters are included. To track interface changes in more detail, we have also considered the mandatory and optional function parameters as well as the default values of the optional parameters. We classified the possible changes to six categories as shown in Table III.

For each version of the examined systems we have computed the ratio of changes over the total number of signatures, differentiating between the six cases shown in Table III. Next, we computed the mean of all versions for each project and the results are summarized in Table IV. The values for cases C1 to C5 are extremely low, considering the almost ten years of evolution for each project. This fact implies that development teams have paid attention in order not to break backward compatibility and that the corresponding APIs are mature. Changes of the 6th type exhibit a mean ranging from

[21] http://en.wikipedia.org/wiki/Snake_case
[22] http://www.php-fig.org/psr/psr-1/
[23] http://en.wikipedia.org/wiki/CamelCase
[24] http://www.tiobe.com/index.php/content/paperinfo/tpci/PHP.html
[25] https://phpexcel.codeplex.com/

3.75% for phpMyAdmin, to 14.22% for phpBB, providing further support to the aforementioned claim, since despite the implementation changes for a number of functions, the corresponding signatures remained stable.

TABLE III. CHANGE CASES OF FUNCTION SIGNATURES

	Category	Impact	Severity
C1	Change of mandatory parameters	Breaking function's compatibility, i.e. client has to refactor function invocation	
C2	Addition of optional parameters	Possible extension of function's functionality or enhanced detail in their results. No impact on compatibility, i.e. existing clients do not have to be adapted	
C3	Removal of optional parameters	Possible breaking of function's compatibility: issues to calls that used the removed optional parameters	
C4	Change of default values	Possible breaking of function's compatibility: issues to calls that expected a different value.	
C5	Change of function's return type (identified by PHP annotations)	Possible breaking of function's compatibility: issues to calls that expect different return type.	
C6	Change of function's implementation.	No impact on interface compatibility but a factor that shows interface stability since developers pay attention when evolving a function to keep their interface intact.	

TABLE IV. RATIO OF CHANGES IN FUNCTION SIGNATURES

Project	C1 (%)	C2 (%)	C3 (%)	C4 (%)	C5 (%)	C6 (%)
Drupal	0.15	0.08	0.03	0.09	0.16	6.02
WordPress	0.06	0.16	0.03	0.27	0.42	6.70
phpBB	0.14	0.35	0.02	3.19	0.97	14.22
MantisBt	0.04	0.11	0.00	0.46	0.50	6.65
phpMyAdmin	0.09	0.09	0.02	0.70	1.26	3.75

E. Classes invasion

Object orientation in PHP was fully supported in version 5.3, but it was partially supported and used few years before that, staring in early 4.x versions. So there was a period where procedural systems could migrate code to classes.

In Figure 7 we present the ratio of the number of methods over the total number of functions and class methods of the system code, excluding third party libraries to show the trend of converting the core codebase of the systems to classes. We observe that Drupal after a long period of denial to the object oriented paradigm, even eliminating the small fraction of classes that existed in the early versions, made a turn in version 7.0 with the introduction of classes. The project with the major change to its coding paradigm is phpBB, where in version 3.0.0 that was a milestone in the project's history, it massively adopted object orientation. WordPress keeps it's slow but steady trend to object orientation, but the huge user contributed code in plug-ins and themes keeps the development team from making major rewrites to the public API of the application. Instead, the developers gradually perform refactoring applications to the internals of the system without breaking backward compatibility. On the other hand, phpMyAdmin that is a widely used project, found in almost any Linux powered web server, has a powerful momentum towards being a fully object oriented system. This is due to the minimal number of user plug-ins or themes, entailing no threat for breaking the public API of the application.

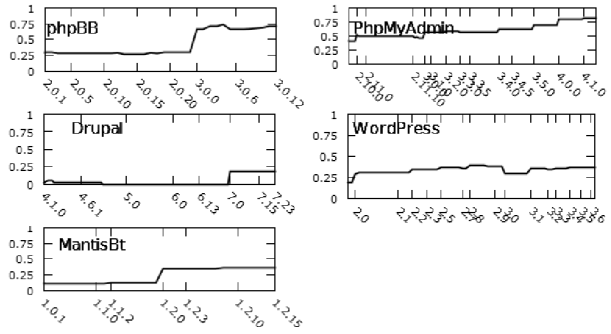

Figure 7 Methods ratio over total number of functions and methods

Our conclusion is that migrating applications from procedural to the object oriented paradigm is not only a matter of developers' will or implementation language, but if the project can afford the cost of breaking backward compatibility imposing significant issues to their clients.

F. Evolution of Complexity

To complement our study with a rather traditional measure, we computed McCabe's cyclomatic complexity (CCN), thereby investigating if PHP practitioners implement comprehensible and thus maintainable code. We calculated CCN per function and then obtained the average CCN of all functions for each version. To make results more readable we categorized the functions according to their CCN in three ranges. A value of 10 is usually considered as a critical threshold [3], [4]. To enable a more fine-grained classification and to comply with critical levels identified by various quality assessment tools, we considered a second threshold at the value of 5. As a result, values in the range [0..5) imply excellent readability, [5-10) medium complexity but still readable code and values higher than 10, code that should be examined closely. Next, we calculated the percentage of functions belonging to each range. The percentages over time are almost constant for all five projects as shown in Figure 8.

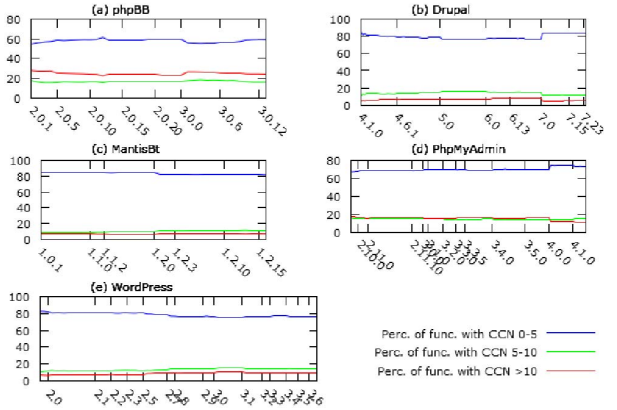

Figure 8 Evolution of functions in three complexity ranges over time

The percentage of functions in the high complexity class remains almost the same across all versions and is relatively low, suggesting that new code added to the projects does not contribute to quality degradation. The resulting conclusion regarding this aspect of software quality, is that the examined projects are developed properly leading to maintainable code.

G. Overview of findings

To facilitate the interpretation of the results, an overview of the investigated phenomena, the employed unit of analysis and the conclusions derived based on the findings for each project is provided in Table V. A '✓' marking implies that the derived conclusion can be considered as validated for the corresponding project, while a '×' mark implies that the conclusion is not validated. ('N/A' for the claim that classes introduce more unused code than functions has been used for two projects that had a very limited number of classes).

TABLE V. *OVERVIEW OF FINDINGS FOR ALL EXAMINED PHENOMENA*

			Validation on Projects				
Phenomenon	Unit of Analysis	Conclusion	phpBB	phpMyAdmin	Drupal	WordPress	MantisBt
A. Survival regarding function usage	Survival function (term. event fan-in=0)	The main source of unused code is library usage.	×	✓	×	✓	✓
		Classes introduce more unused code than functions	N/A	✓	N/A	✓	✓
		Unused code appears in major versions	✓	✓	✓	✓	✓
B. Survival regarding function removal	Survival function (term. event function deletion)	Function removal appears in major versions	✓	✓	✓	✓	✓
		Only new features are implemented with OO	×	×	✓	✓	×
		Functional code replaced with OO code	✓	✓	×	×	✓
C. Library usage	Percentage of library source code over project's own code	Projects reuse code incorporating third party libraries	✓	✓	×	✓	✓
D. Interface stability	Percentage of functions in each change category	Function interface remains stable	✓	✓	✓	✓	✓
E. Classes invasion	Percentage of class code over total source code	Projects gradually migrate to OO paradigm	✓	✓	✓	✓	✓
F. Evolution of Complexity	Percentage of modules in each complexity category	Complexity remains stable	✓	✓	✓	✓	✓
Conclusion: The examined PHP applications undergo systematic maintenance							

V. THREATS TO VALIDITY

As already mentioned, one important threat to the construct validity of our study is the presence of *stealth calls*, i.e. functions/methods which are identified as unused, despite the fact that there are actually invocations on them. According to Wohlin et al. [20], construct validity reflects the extent by which the phenomenon under study really represents what is investigated. Although it is not possible to estimate the extent of stealth calls in the examined systems, this threat is partially mitigated since the analysis is evolutionary and thus changes in the observed phenomena and trends are relative, partially factoring out the effect of unidentified function invocations.

Another threat of the same type pertaining to the applied survival analysis is related to the consideration that once functions become unused they remain unused for the rest of the evolution. Obviously, one function might become unused and then be invoked again in a subsequent version or even might become unused in the future again and so on. This threat is mitigated by the choice to consider the survival function rather than the cumulative survival estimator which would overstress the impact of unused functions.

Since the analysis is based on results from 5 web applications, threats to external validity are present limiting the ability to generalize our findings. Moreover, the fact that the examined applications are large, widely known and heavily used applications, implies that the development practices in these projects might differ from other, less professionally developed systems. However, since the goal was to investigate whether development with scripting languages can comply with the proper practices laid out by software engineering, the authors believe that the examined systems are representative of multi-version, multi-person projects in Web applications addressing the needs of a vast community of clients.

VI. RELATED WORK

Software evolution is one of the most studied areas in software engineering originating to the 1970's when M. Lehman laid down the first principles of software evolution [7] which gradually evolved to eight laws. The validity of Lehman's laws in various contexts has been studied by several researchers. Recently, Xie et al. [21] studied the software evolution of seven open source projects implemented in C.

McCabe's CCN and LOC were used to investigate the validity of the second and the sixth law, respectively. Both laws have been validated. The findings for PHP projects are in agreement to these conclusions for C projects.

Survival analysis to estimate aspects of software projects has been employed by Sentas et al. [17] as a tool to predict the duration of software projects. In a similar manner, Samoladas et al. [15] employed the Kaplan-Meier estimator to predict the duration of open source projects. Scanniello [16] applied the Kaplan-Meier estimator on Java open source projects, to study the effect of dead code in the evolution of projects. The results show that high rates of unused code are detected in most of the projects in that study.

Regarding the use of libraries, Heinemann et al. [5] studied the extent of software reuse in Java open source software. The authors made a distinction between black box and white box usage which does not apply to scripting languages and in order to quantify the extent of reuse they measured byte code of jar files used. They showed that in most cases over 50% of the code size has its source in third party libraries.

Mockus [11] investigated large-scale code reuse in open source projects by identifying components that are reused among several projects. However, Mockus' work quantifies how often code entities are reused, rather than the actual third party code. Based on their results, code reuse is a common practice in open source projects, a fact which is confirmed by the findings in our study.

VII. CONCLUSIONS

Scripting languages and PHP in particular form the cornerstone of an increasing number of widely acknowledged and heavily used web applications. Five such projects have been analyzed in this paper in an attempt to investigate the maintenance habits followed by open-source developers relying on PHP. Several aspects of software evolution have been investigated, including the presence of unused code, the removal of functions, the use of third-party libraries, the API stability, complexity as well as the migration to the object-oriented paradigm.

The results are conclusive: The examined PHP projects undergo systematic maintenance driven by targeted design decisions and PHP does not seem to hinder the adaptive and perfective maintenance activities. In particular, projects rely on the use of third-party libraries which in turn introduce unused code. All systems gradually migrate to the object-oriented paradigm. Migration to object-orientation in three of the five projects is performed by replacing functions with objects, while for two projects only new features are implemented with classes. The interface of functions remains strictly stable avoiding compatibility problems with existing clients. Finally, the complexity of the projects appears to remain stable, in terms of the percentage of system modules in distinct complexity levels. All of these findings suggest that maintenance is performed with care and in a well-organized manner for the examined PHP applications and significant lessons can be learned from their evolution history.

ACKNOWLEDGMENT

This research has been co-financed by the European Union (European Social Fund – ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) – Research Funding Program: Thalis – Athens University of Economics and Business - SOFTWARE ENGINEERING RESEARCH PLATFORM.

REFERENCES

[1] L. Arhippainen, Use and Integration of Third-party Components in Software Development. http://www.vtt.fi/inf/pdf/publications/2003/P489.pdf, VTT, 2003.

[2] S. Bedi and P. J. Schroeder, “Observations on the Implementation and Testing of Scripted Web Applications.,” in WSE, 2004, pp. 20–27.

[3] S. Bergmann and S. Priebsch, Real-World Solutions for Developing High-Quality PHP Frameworks and Applications. Wiley, 2011.

[4] N. E. Fenton and S. L. Pfleeger, Software Metrics: A Rigorous and Practical Approach. PWS Publishing Company, 1997.

[5] L. Heinemann, F. Deissenboeck, M. Gleirscher, B. Hummel, and M. Irlbeck, “On the Extent and Nature of Software Reuse in Open Source Java Projects.,” in ICSR, 2011, vol. 6727, pp. 207–222.

[6] E. L. Kaplan and P. Meier, “Nonparametric Estimation from Incomplete Observations,” J. Am. Stat. Assoc., vol. 53, no. 282, pp. 457–481, 1958.

[7] M. M. Lehman, “Programs, life cycles, and laws of software evolution,” Proc. IEEE, vol. 68, no. 9, pp. 1060–1076, Sep. 1980.

[8] M. M. Lehman et al., “Metrics and Laws of Software Evolution – The Nineties View”, in Proceedings of the 4th International Symposium on Software Metrics, IEEE, 1997, pp. 20-32.

[9] W. C. Lim, “Effects of Reuse on Quality, Productivity, and Economics.,” IEEE Softw., vol. 11, no. 5, pp. 23–30, 1994.

[10] R. P. Loui, “In Praise of Scripting: Real Programming Pragmatism.,” IEEE Comput., vol. 41, no. 7, pp. 22–26, 182008 2009.

[11] A. Mockus, “Large-Scale Code Reuse in Open Source Software,” in Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development, Washington, DC, USA, 2007, p. 7–.

[12] J. K. Ousterhout, “Scripting: Higher Level Programming for the 21st Century,” Computer, vol. 31, no. 3, pp. 23–30, Mar. 1998.

[13] K. H. Pollock, S. R. Winterstein, C. M. Bunck, and P. D. Curtis, “Survival analysis in telemetry studies: the staggered entry design,” J. Wildl. Manag., vol. 53, pp. 7–15, 1989.

[14] L. Prechelt, “Are scripting languages any good? A validation of Perl, Python, Rexx, and Tcl against C, C++, and Java.,” Adv. Comput., vol. 57, pp. 205–270, 2003.

[15] I. Samoladas, L. Angelis, and I. Stamelos, “Survival analysis on the duration of open source projects,” Inf. Softw. Technol., vol. 52, no. 9, pp. 902–922, 2010.

[16] G. Scanniello, “Source code survival with the Kaplan Meier.,” in ICSM, 2011, pp. 524–527.

[17] P. Sentas, L. Angelis, and I. Stamelos, “A statistical framework for analyzing the duration of software projects.,” Empir. Softw. Eng., vol. 13, no. 2, pp. 147–184, May 2008.

[18] A. Srivastava, “Unreachable Procedures in Object-Oriented Programming.,” LOPLAS, vol. 1, no. 4, pp. 355–364, 021992 2002.

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

[20] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen, Experimentation in Software Engineering. Springer, 2012.

[21] G. Xie, J. Chen, and I. Neamtiu, “Towards a better understanding of software evolution: An empirical study on open source software.,” in ICSM, 2009, pp. 51–60.