Proceedings of the Informing Science + Information Technology Education Conference

An Official Publication of the Informing Science Institute
InformingScience.org

InformingScience.org/Publications

A DESIGN SCIENCE TOOL TO IMPROVE CODE MAINTAINABILITY FOR HYPERTEXT PRE-PROCESSOR (PHP) PROGRAMS

Grant Trudel	livepro Australia, Brisbane, Australia	grant.trudel@alumni.ctuonline.edu
Samuel Sambasivam*	Woodbury University, Burbank, CA, USA	Samuel.Sambasivam@Woodbury.edu

*Corresponding author

ABSTRACT

Aim/Purpose	A design science tool to improve code maintainability for PHP programs.
Background	This paper addresses this issue by providing an enhancement to an existing tool that specifically addresses and improves program maintainability for PHP programs.
Methodology	This paper uses design science research which involves creating or modifying one or more artifacts. In this case, the artifact is PHP Code Sniffer, which was modified to improve code maintainability of PHP programs.
Contribution	This paper provides a method of improving code maintainability of PHP programs.
Findings	Code quality in terms of maintainability is a modern-day issue. No tool specifically addresses maintainability. When updating PHP code that has poor maintainability, there is a much higher likelihood that errors will be introduced than with code that is more maintainable.
Recommendations for Practitioners	For PHP developers and project leaders dealing with PHP programs, running the modified PHP Code Sniffer will improve code maintainability.
Recommendations for Researchers	Examine ways to improve other code quality aspects such as reliability, security, usability, efficiency, and so on.
Impact on Society	PHP programs, which run the background processes behind most web pages, will be more robust, reliable, and correct.
Future Research	Expand this to investigate improving maintainability for other common programming languages.

Accepted by Editor Michael Jones | Received: December 20, 2020 | Revised: April 2, 2021 | Accepted: April 19, 2021.

Cite as: Trudel, G., & Sambasivam, Sam. (2021). A design science tool to improve code maintainability for Hypertext Pre-processor (PHP) programs. In M. Jones (Ed.), *Proceedings of InSITE 2021: Informing Science and Information Technology Education Conference*, Article 1. Informing Science Institute. https://doi.org/10.28945/4769

(CC BY-NC 4.0) This article is licensed to you under a Creative Commons Attribution-NonCommercial 4.0 International License. When you copy and redistribute this paper in full or in part, you need to provide proper attribution to it to ensure that others can later locate this work (and to ensure that others do not accuse you of plagiarism). You may (and we encourage you to) adapt, remix, transform, and build upon the material for any non-commercial purposes. This license does not permit you to use this material for commercial purposes.

Keywords software quality, software development, PHP, programming, quality assurance, maintainability, metrics, design science, refactoring

INTRODUCTION

Software maintainability is an important aspect of software quality that deserves close attention (Coleman et al., 1994; Đorđević, 2017). Software development has experienced rapid changes in the past decade (Denning, 2016; Rodríguez et al., 2017). New tools, languages, platforms, and paradigms have presented software developers with a constantly shifting landscape (Denning, 2016; Wang et al., 2012). At the same time, there have been increasing demands for software solutions, particularly for the Internet (Mazaika, n.d.; McKendrick, 2015). Organizations are requiring developers to keep up with the latest trends in technology while meeting deadlines, to remain competitive and keep up with a continuously growing customer base (Johnson, 2018). Poorly maintained software has dire consequences, resulting in systems that are rife with software bugs and code that has to be debugged or rewritten from scratch, leading to high costs and lost customers (Freyja, 2017; Jones, 2015). Research is needed to examine ways to improve this situation.

The goal of this study was to evaluate the effectiveness of modifying an existing artifact to improve the maintainability of Personal Home Page Hypertext Preprocessor (PHP) programs. Despite the high availability of programming tools on the Internet, as well as training and development programs, questions arise regarding the trend of the maintainability of the software being written by developers in the most popular Internet programming language, PHP (Netcraft, 2016). Maintainability is an aspect of code quality that should be of primary concern to programmers since they proportionally spend a great deal more time maintaining existing code than writing new programs (Campbell et al., 2012; Stephens, 2015). It is noteworthy that code that is difficult to maintain leads to issues such as software bugs (Coleman et al., 1994).

The purpose of this design science exploratory study was to discover the architectural changes to the current existing quality improvement tool that were needed to establish better maintainability for PHP programs. The research method was design science, and the research design was exploratory. The study's method of conveying learning was to explore and explain the architecture changes in an existing quality improvement tool that improved code maintainability for PHP programs. The central phenomenon of the study was exploring a way to improve the maintainability of the PHP code. The data source used in this study was open source PHP programs. The primary idea driving the study was that PHP code maintainability could improve by using an automated tool.

The PHP programming language was created in 1994 by Rasmus Lerdorf, released to the public in 1995, and evolved into a complete programming language in 1996 (PHP Group, 2018b). In 2009, the Framework Interoperability Group (FIG) was formed to recommend PHP coding standards known as PHP Standard Recommendations (PSRs) (Independent Software, 2016; PHP Framework Interop Group, 2018). Given the more than ten-year gap from when the language was released until the formation of the group to suggest coding standards for PHP, it seems likely that a lot of PHP was coded without a lot of structure, forethought, and quality in mind particularly regarding maintainability. By 2010, PHP made up 72.5 percent of all server-side languages for websites worldwide (W3Techs, 2018).

Despite the efforts of the FIG, the quality of code developed by programmers working in PHP has not improved significantly (Ricca & Tonella, 2005; Xu & Chen, 2001). In order to help alleviate this situation, the PHP Code Sniffer tool has been developed to improve coding standards such as adherence to PSRs (PHP Group, 2018a). Standards, when followed, enhance the maintainability of programs (Thomas, 2017). Further, a piece of software called PHP Code Sniffer was developed to

change PHP code to conform to coding standards. However, PHP Code Sniffer does not focus specifically on maintainability. This study sought to address this shortcoming by enhancing PHP Code Sniffer to improve the maintainability aspect of PHP programs.

Significance of the Study

The quality of PHP programming is worth studying for three reasons. First, PHP is a free open-source programming language for developing server-side functionality for web pages (Brookshear & Brylow, 2014). Since its launch in 1995 by Rasmus Lerdorf, PHP has grown in popularity to become the most widespread web programming language, running on well over 200 million websites (Netcraft, 2016). The setting in which this research occurred is in software development in the PHP programming language which embeds into HyperText Markup Language (HTML) pages (Aryanto et al., 2015). Typically, PHP is used as a server-side system, housing business rules and interacting with a database such as MySQL via an application programming interface (Lu et al., 2011). According to research studies, the quality and reliability of web applications are often sub-standard or unsatisfactory (Ricca & Tonella, 2005; Xu & Chen, 2001).

The consequences of inferior quality code include resources spent to fix them, security holes, and maintainability. Poor software quality leads to developers spending excessive amounts of time on tracking down and fixing software bugs (Jones, 2015). Misleading or obsolete information that is readily available on the internet can mislead these upcoming PHP developers to create code that does not meet basic standards of security and maintainability (Lockhart, 2016). Security concerns have been on the rise, and traditional approaches to locking down websites with one-time fixes have been ineffective (Vashist & Gupta, 2014).

Second, the answers to the research questions can help programmers fulfill their duties to produce software that is highly maintainable. In order to achieve this, various control processes should be in place, and quality must be the focus of the entire project team throughout the software development lifecycle (Conboy, 2010; Irani, 2010; Moha, 2007). Quality improvements in a software development company are possible by examining project data, characterizing productivity and quality, and deriving strategic and tactical options (Siok, 2008).

Third, this study attempted to fill a gap in the literature. From the researcher's perspective, there are no tools available that modify PHP code to improve its maintainability. There have been no design science research studies that seek to address the problem of software maintainability by designing a solution that could be applied and produce measurable results in the form of metrics to show the effects of the artifact. This design science study demonstrated how such a tool could be developed that when applied to any PHP program or system by developers or project leaders, improves code maintainability for a more maintainable, robust, secure system.

The proposition for this study was that maintainability of PHP code could improve by making architectural changes to the selected PHP program code quality improvement tool, PHP Code Sniffer.

Consequences of Software with Inferior Quality

From the outset, researchers (e.g., Boehm, 1978; Singh et al., 2011) have advocated tight linkages between software development and software quality, asserting that software quality impacts business success and that organizations must somehow integrate quality to reduce failure rates and improve system maintainability. Boehm (1978), for example, noted that code quality had been a concern since the early days of programming: well before the advent of the personal computer.

The lack of quality in software has been the cause of multiple failures in various fields (Jee, 2018; Lake, 2010). For example, in May 2017, breaches have occurred in the National Health Service in England and other organizations in the UK from WannaCry ransomware. In another case, British Airways had a massive software failure leading to a cancellation of over a thousand flights in May

2017. In mid-January 2016, the Nest smart thermostat had a software update that left customers unable to control the temperature to heat their homes or get hot water during one of the coldest weekends (Jee, 2018).

Poor quality software can have consequences ranging from a costly failed project to a loss in human lives (Lake, 2010). An infamous computer software problem involved the Mars climate orbiter when in 1998 the different units of measurement used by different groups of engineers caused the spacecraft to have 4.45 times more thrust than it should have been (Lake, 2010). This seemingly small software glitch cost $337.6 million when the spacecraft was destroyed (Lake, 2010). Another incident occurred in the military (Lake, 2010). During the first Persian Gulf War, a software error allowed a Scud missile to get past the defensive Patriot missile, resulting in the death of 28 soldiers (Lake, 2010). These examples highlight the effects of software errors, which are more likely to be introduced into systems that have low maintainability.

PHP Coding Standards

PHP has a set of accepted standards, and other standards that are currently being worked on (PHP Framework Interop Group, 2018). These standards are known as PHP Standards Recommendations (PSRs). Each PSR refers to a grouping or type. For instance, PSR-1 is the basic coding standard, PSR-2 is the style guide, PSR-3 is the logger interface, and PSR-4 is the standard for autoloading classes. There are currently ten accepted PSRs (including 1 through 4 mentioned above), one in review status (PSR-12: the extended coding style guide), and three in draft status. These standards are provided to guide programmers to produce good quality PHP code that is easy to maintain; however, no one is holding programmers accountable to comply with these standards (Denning, 2016).

Maintainability

One of the most common quality code characteristics in all models is maintainability (Aversano & Tortorella, 2013; Boehm et al., 1976; Haigh, 2010; ISO-9126, 1991; Spinellis et al., 2009). Although not visible to the user, maintainability has been used as a measurement of software complexity since, from the developer's perspective, the more intricate a piece of software is, the more likely it is to contain errors (Coleman et al., 1994). Maintainability of a program should be a key focus of a programmer during the early phases of development, particularly regarding the length of a subroutine or function (Malhotra & Chug, 2016; Misra, 2005). In 2009, Spinellis et al. (2009) discovered that from 1995 to 2005, as the lines of code (LOC) increased in the kernel and user programs of the FreeBSD system, there was a corresponding decrease in the maintainability index (MI) (see Figure 1).

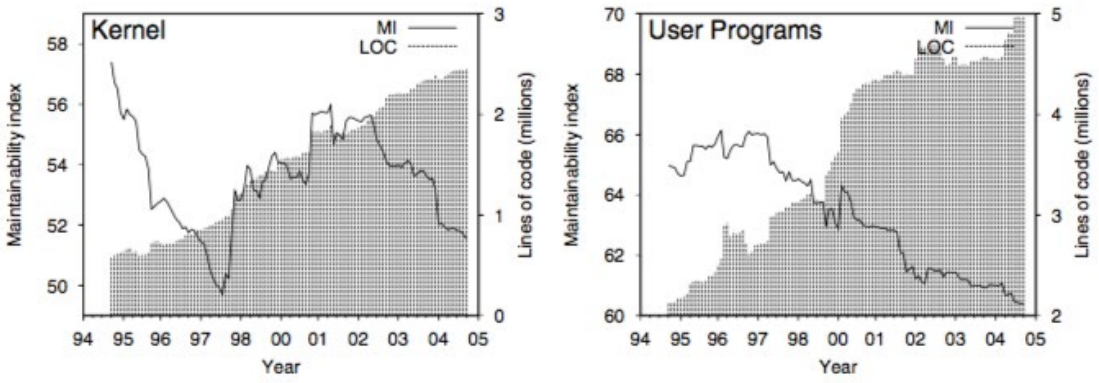

Figure 1. Program growth and maintainability index over time in the FreeBSD kernel and user programs. Reprinted with permission (see Appendix B) from Spinellis et al. (2009, 7).

THE MAINTAINABILITY INDEX (MI)

A popular code quality metric is the maintainability index (MI) (Asadi & Rashidi, 2016; Capiluppi et al., 2009; Elish & Elish, 2009; Ganpati et al., 2012; Spinellis et al., 2009). The typical range for MI for programs is between 200 to -100, with higher values implying better maintainability (Spinellis et al., 2009). The MI has been used effectively to identify and quantify maintainability and improve code quality (Welker, 2001). Kaur et al. (2014) identify the formula for MI is:

$$MI = 171 - 5.2 * \ln(avgV) - 0.23 * avgV(g) - 16.2 * \ln(avgLOC) \quad (1)$$

where:

avgV represents the average Halsted volume

avgV(g) represents the average cyclomatic complexity (the number of possible paths or branches a program or piece of code could take when executing)

avgLOC stands for the average number of lines of code

The formula for avgV (Halsted volume) is:

$$avgV = N \times \log2\ n \quad (2)$$

where

N is program length = total number of operators + total number of operands

n is program vocabulary = number of distinct operators + number of distinct operands

The formula for avgV(g) (cyclomatic complexity) is:

$$avgV(g) = E - N + 2P \quad (3)$$

where

E = the number of graph edges

N = the number of graph nodes

P = the number of connected components

A variant of the above formula is used in the open source program PHP metrics which is a tool used by this researcher to quantify the maintainability of a piece of software written in PHP (Lepine, 2015). It returns a value from 0 to 221, with the higher score indicating better maintainability. The variant formula as modified by Lepine is:

$$MI = 171 - 5.2*\log2(V) - 0.23*CC - 16.2*\log2\ (LOC) + 50*\sin(\sqrt{2.4 * CM}) \quad (4)$$

where

V represents the Halstead Volume

CC represents the cyclomatic complexity

LOC stands for lines of code

CM is the portion of comment lines in the code (a number from 0 to 1) (Lepine, 2015)

RESEARCH DESIGN

Design science research in information systems involves creating one or more artifacts including, but limited to, modeling tools, methods for evaluating systems, and decision support systems (Gregor & Hevner, 2013). An artifact is described in this context as an entity such as a model or process (software), such as an algorithm that can be converted into a piece of software (Goldkuhl, 2002; Gregor

& Hevner, 2013). In this study, the artifact is a piece of software which is designed to improve the quality of PHP software.

The purpose of this research was to determine the effectiveness of modifying an artifact to improve the maintainability of existing PHP programs. The software artifact was designed to improve PHP code quality based on current best quality practices in code design and development but did not focus on maintainability. The artifact was modified to enhance the maintainability characteristic of PHP programs. The artifact was tested by running it against a sample of randomly selected open source PHP programs before and after it was modified, and the results were recorded, analyzed, and discussed. Figure 2 illustrates the research methodology and steps taken by the authors for this study.

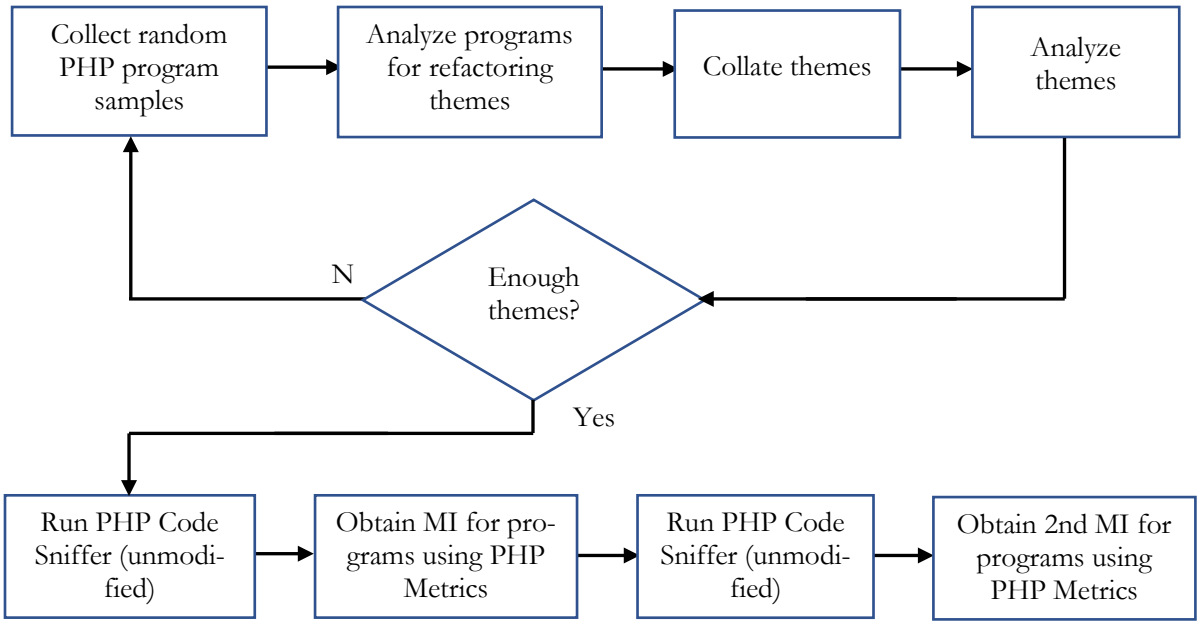

Figure 2. Implementation flowchart

POPULATION AND SAMPLE

This study did not involve any human or animal subjects. This study did not depend on or refer to any information of a personal nature that related to human subjects. For this research, random PHP code samples were obtained from several open source projects:

http://freshmeat.sourceforge.net
https://www.cloudways.com/blog/top-php-github-projects/
https://sourceforge.net
https://www.phpclasses.org

These websites provided the data for the experiment since they are all open source and publicly available. Many code samples are written in many different programming languages, but for this study, only examples of complete, functioning programs written in PHP were selected. Further, only those programs that have been downloaded within the past year were considered, to ensure only the most current programs were included in this research.

The sample size is important to determine for a study. To determine an appropriate sample size depends on the context and type of research being conducted (Boddy, 2016). The purpose of this design science exploratory study was to identify common themes that improved the quality of PHP programs, rather than conduct interviews. During the data analysis phase of the study when the themes emerged, a sample of 15 PHP programs was determined to be sufficient. This number, 15,

was determined based on the analysis of the data after performing refactoring on the modules and observing the themes that emerged (see Appendix A, Table A4). Once a theme was discovered that offered the best method to improve maintainability, there was no need to explore additional code samples.

SAMPLING PROCEDURE

The abovementioned websites provided multiple code samples. To ensure randomness, a select number of programs were downloaded from each of the above sites. A stratified sample method is recommended for non-homogeneous populations (Kitchenham & Pfleeger, 2002), which is certainly the case for software programs. To obtain a stratified sample, the programs were selected based on having recent updates and being coded in the PHP language. Those randomly selected programs were downloaded for inclusion and analysis in this study.

INSTRUMENTATION

In this study, no interviews or other instruments were used. A software artifact was modified for this study to conduct the experiment and test the proposition. The current code quality metric measure measured the results of the experiment for software maintainability, known as the maintainability index (MI). The software artifact is detailed in the data analysis section and Appendix C.

The software artifact in question was a piece of open source software called PHP Code Sniffer (PCS) designed to improve the software quality of PHP programs (PHP Group, 2018a). When running with default settings, PCS takes a piece of PHP code and rewrites it to conform to PEAR programming standards (PHP Group, 2018c). To conform to PSR-1 and PSR-2 standards, PCS can run with those parameters as input to override the default setting. The PSR-1 and PSR-2 standards were the settings used in this study.

The input to the artifact (PCS) consisted of a select number of open source PHP programs. PHP programmers often share software solutions to assist other PHP programmers in their development efforts, and these open source code repositories are well known amongst the PHP community of coders. However, the code quality of all these open source solutions may not be of the highest grade, causing issues when it comes to maintaining the code when, or after, installation.

The first set of outputs consisted of the downloaded PHP programs which had the original PCS tool configured to coding standards PSR-1 and PSR-2. These programs then had their maintainability index (MI) calculated and recorded. The PCS tool was then modified by the researcher to focus specifically on code maintainability. Then modified PCS tool was run against the original set of downloaded PHP programs a second time with coding standards PSR-1 and PSR-2. The MI was then calculated for each of these programs and compared against the original MI from the first run. The results were collected, analyzed, and discussed.

DATA COLLECTION

Design science research (DSR) measures the effectiveness of an artifact such as a program that will enhance the quality of PHP code (Hevner et al., 2004). In the discipline of computer science, DSR is the method that provides the best fit in demonstrating improvements to artifacts such as that used in this study (Venable et al., 2012). The first step in data collections was to download the random code samples from various open source projects on the Internet to a predetermined folder, one for each program or project. The next step was to use the PHP Code Sniffer tool to modify the programs by PSR-1 and PSR-2 standards. The next step was to find out the maintainability of the code by running the PHP metrics program to obtain the maintainability index. Then the PHP Code Sniffer was modified to enhance maintainability. The same PHP code samples were then modified by the redesigned artifact and analyzed once more for maintainability using the PHP metrics program. Both before and

after maintainability metric measures for the downloaded sample of PHP programs were collected, analyzed, and discussed for this study.

ACCESS TO SAMPLE

The sample data consisted of open source programs from various websites that were well known in the programming community. Anyone with Internet access will be able to access any of the programs that were selected for this study.

DATA ANALYSIS

The results of applying the artifact were analyzed to determine the effectiveness of the artifact in improving the maintainability software quality characteristic of the random sample of PHP programs. The programs were a collection of open source PHP programs written by and for developers. The first step was to download the programs which were freely available from several well-known PHP repositories. The next step was that each program was modified by the researcher using refactoring methods and the maintainability index measured. In the next step, the refactoring methods identified the themes. The next step tabulated the improvements by theme (Venable et al., 2012). The researcher modified the artifact according to the theme providing the greatest improvement to maintainability (Hevner et al., 2004). Finally, the test was repeated multiple times, making modifications to the artifact for improving maintainability, to obtain the maintainability index of each program from each run (Hevner et al., 2004). The two software tools used in this process were PHP Code Sniffer (the artifact that was modified to improve maintainability) and PHP metrics (to measure maintainability). Data was triangulated from the PHP Code Sniffer tool and PHP metrics tool to discover the themes that would drive the changes to the artifact.

This section explains the modifications made to the selected artifact used in this study that focuses on improving PHP code: PHP Code Sniffer. The first section discusses the new class created for this study and examines each function, describing how each works individually. Then how these functions interact with each other is described. How to use the new class is discussed. The next section describes the changes made to the PHP Code Sniffer to make use of the new class. The results from one of the test samples used in this study are then presented, to demonstrate the effect the refactoring has on some PHP code

The refactor class

The new refactor class provides a basic refactoring process by applying the method identified as theme two, moving code to another class, to a function found within a class that meets certain criteria. To aid in maintainability, any files containing multiple class definitions were first divided into multiple files, so that there was one file per class. The criteria for viable functions are: the function must consist of at least 100 lines of code (including comments), and it must not use any class variables. In such cases, a new class is created consisting of that one function, and the original code modified to make a call to that class. The result will be a collection of smaller and therefore more maintainable code modules. However, the downside of this process is that there will be more modules (files) to maintain than before. Each file will be individually more maintainable than the combined files before the refactoring process executed, as proven by the maintainability index. The Refactor class code was based on PHP Class Splitter (Turland, 2018). The code in the original Class Splitter was extensively modified and expanded by this researcher for this study.

The runProcess function

The runProcess function first determines if the incoming parameter is a PHP file and, if so, will run the refactor function on the file and exit. Otherwise, the runProcess function first finds all the paths in the input path and stores them in an array. Then for any files with "inc.php" (usually signifying a PHP file with multiple class definitions in it), it splits class files into individual class files. If necessary,

the paths are found again, and the array with each path is repopulated. Then for each path in the array, the refactor function is called to perform the code refactoring. See Figure C1 (Appendix C) for the code listing for the runProcess function.

The getPaths function

In Figure C2 (Appendix C) the getPaths function uses the RecursiveIteratorIterator class to obtain all the paths and files within a directory (Barbaud, 2018; PHP Group, 2018d). The realpath function in PHP is called to expand all symbolic links and resolve extra forward slash characters to return the canonicalized absolute pathname (PHP Group, 2018e). Each of the key elements in the returned objects array contains the files and paths within the $path directory and is stored in the $return array, which is returned to the caller.

The classSplitterFile function

The classSplitterFile function takes a file as input and, if it contains more than one class in it, creates separate files for each class found. The code for classSplitterFile is based on PHP Class Splitter (Turland, 2018), but modified for this study. The Boolean variable $splitThisTime passed in as a parameter is passed by reference using the "&" prefix, which means the variable will contain the value set by the function when the function exits. This Boolean variable tells the caller whether the function created new files or not. Any comments associated with a class are also moved into the new class file. The reasoning behind this function as related to maintainability is that it should be easier for a programmer to maintain smaller class files than one large class file containing multiple classes. The programmer can more easily locate the classes by their file names, which are identical to the classes, instead of having to do a scan through various PHP include files to find a class. Figure C3 (Appendix C) lists the code for the classSplitterFile function.

The countClasses Function

The countClasses function returns an integer representing the number of classes found in the input file parameter. The class accomplished this by looking for and counting the class tokens in the file. Figure C4 (Appendix C) lists the code for countClasses.

The refactor Function

The refactor function does the refactoring for a file by looking for any functions that have at least 100 lines of code in them and are not using class variables (which are variables that begin with $this). For any such functions, a new class is created for the function by the same name as the function, and the class and function are placed in a new file. Then the code containing the function code in the incoming file is replaced with a call to the new class function. This code replacement occurs in the replaceCode function. See Figure C5 (Appendix C) for the code for the refactor function.

The replaceCode function

The replaceCode function takes as input two parameters: a file which has the class and function to be replaced, and a function name which is the name of the function within the class that should be replaced. The first portion of the code is a while loop which steps through the file, storing any code for the function defined in the second input parameter, $function_name. After the while loop, a test determines if the code was found and if so, the next part of the program proceeds to execute the replace logic. First, a new file is created containing all the logic from the function code. Then, the original code is replaced with a call to the new function in the new class just created. See Figure C6 (Appendix C) for the program code for the replaceCode function.

Function Interactions

The class functions interact in the following manner. The runProcess function calls the getPaths function to obtain all the paths and files within the input parameter, $path, which should contain the directory to be refactored. The paths and files are stored in an array, which is stepped through to perform any splitting by calling the classSplitterFile function. The classSplitterFile function calls the countClasses function to see if there are more than one classes in the file. The classSplitterFile function will only split up files if the count of classes is greater than one. If any file splitting occurred, then the paths and files are obtained once again by calling getPaths. Then, the array of files and paths is stepped through again and for each PHP file found (determined by having a suffix of ".php") a call is made to the refactor function. Within the refactor function, a check is performed by calling countClasses on the file to ensure we have a file with more than one class. If there are two or more classes in the file and on if a function is found which is not using class variables and has more than 100 lines of code, then the replaceCode function is called with that file and function name as the two input parameters, respectively. See Figure 3 for a structure chart diagram illustrating these function calls.

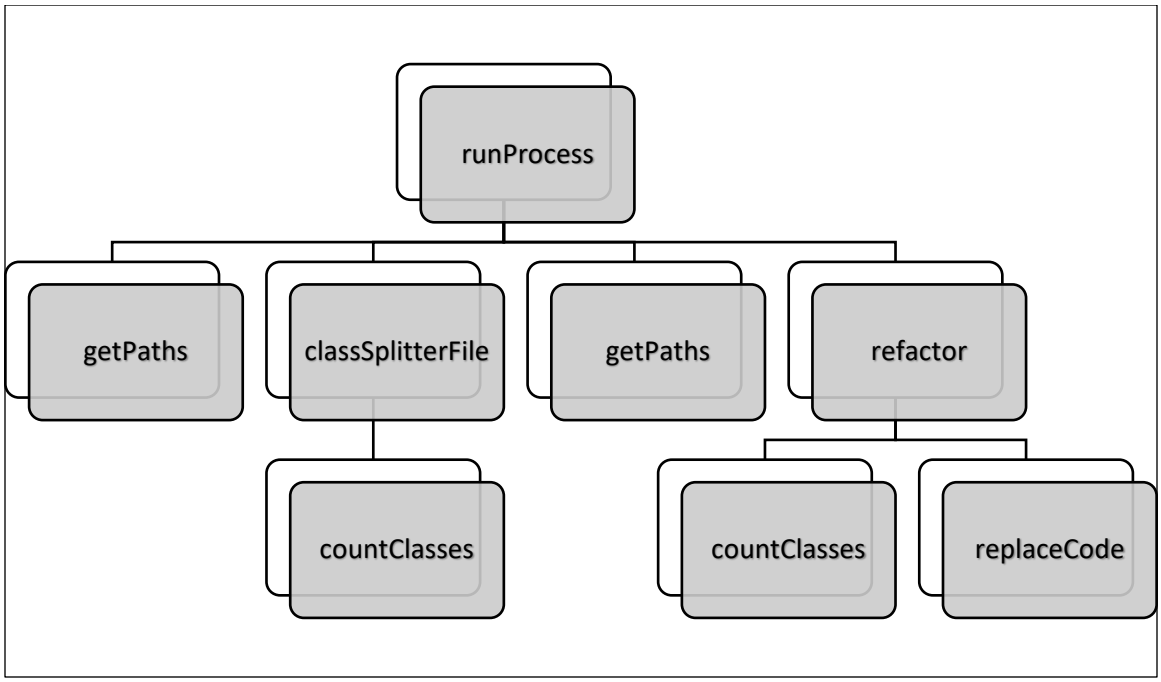

Figure 3. Structure chart of Refactor class functions

Using the Refactor Class

Since all the functions within the Refactor class are static, there is no need to create a Refactor object. The call to Refactor::runProcess($path) where $path is the string containing the main directory that is to be refactored is all that is required to execute the refactoring process. The code must be accessible within the class that Refactor is used, which could be done using the PHP include (or require) statements, or by using namespaces. The changes to the PHP Code Sniffer used the latter, described below.

PHP Code Sniffer Changes

To implement the Refactor class within the PHP Code Sniffer requires a few changes to Runner.php. First, to be able to access the class, the line "namespace PHP_CodeSniffer;" was added to the top of the Refactor class. Second, the line "use PHP_CodeSniffer\Refactor;" was added underneath the

other "use" statements in CodeSniffer/src/Runner.php. Third, the last four lines in the code in Figure 4 was added to Runner.php. This code loops through the config files array initialized by the creation of the Config object and runs the refactor process for each file. Permission to modify PHP Code Sniffer was provided to the researcher from the author. Appendix D displays this permission.

```
/**
* Run the PHPCBF script.
*
* @return array
*/
public function runPHPCBF()
{
    if (defined('PHP_CODESNIFFER_CBF') === false) {
        define('PHP_CODESNIFFER_CBF', true);
    }

    try {
        Util\Timing::startTiming();
        Runner::checkRequirements();

        // Creating the Config object populates it with all required settings
        // based on the CLI arguments provided to the script and any config
        // values the user has set.
        $this->config = new Config();
        foreach ($this->config->files as $file)
        {
            Refactor::runProcess($file);
        }
```

Figure 4. Code added to Runner.php in PHP Code Sniffer (the last 4 lines in the above piece of code were added)

RESULTS

Note that sometimes when running PHP Code Sniffer, it could run out of memory. When this occurs, it may be run against specific directories or files to avoid the larger files or directories. Alternatively, it can be run with parameters to exclude certain files in the file src/Standards/PSR2/ruleset.xml, as in Figure 5.

```
<exclude-pattern>*/prototype.js</exclude-pattern>
```

Figure 5. Running PHP Code Sniffer with an exclusion

The results of running the refactor process consists of the splitting of class files and one or more separated function calls. To illustrate these differences, the affected before and after file structures and functions are listed. Figure 6 shows the file structure before and after the process for the AShopGPL600 product. Figure 7 shows a part of the SimplePie_Misc encoding function before the process, and Figure 8 shows the same function after the process. Figure 9 shows the beginning of the new class and function created from the refactoring process.

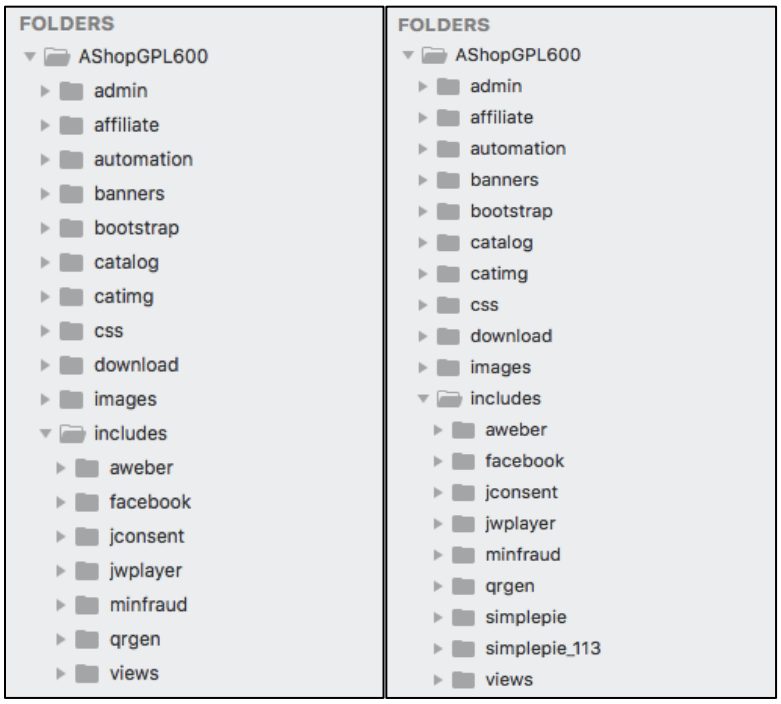

Figure 6. Folder structure before (left) and after (right) the refactoring process (notice the two extra folders [simplepie and simplepie_113] on the right)

simplepie.inc.php

```
12591       /**
12592        * Normalize an encoding name
12593        *
12594        * This is automatically generated by create.php
12595        *
12596        * To generate it, run `php create.php` on the command line, and copy the
12597        * output to replace this function.
12598        *
12599        * @param string $charset Character set to standardise
12600        * @return string Standardised name
12601        */
12602       public static function encoding($charset)
12603       {
12604           // Normalization from UTS #22
12605           switch (strtolower(preg_replace('/(?:[^a-zA-Z0-9]+|([^0-9])0+)/', '\1', $charset)))
12606           {
12607               case 'adobestandardencoding':
12608               case 'csadobestandardencoding':
12609                   return 'Adobe-Standard-Encoding';
12610
12611               case 'adobesymbolencoding':
12612               case 'cshppsmath':
12613                   return 'Adobe-Symbol-Encoding';
12614
```

Figure 7. The encoding function for the SimplePie_Misc class in the simplepie.inc.php file before refactoring.

SimplePie_Misc.php

```
289     /**
290      * Normalize an encoding name
291      *
292      * This is automatically generated by create.php
293      *
294      * To generate it, run `php create.php` on the command line, and copy the
295      * output to replace this function.
296      *
297      * @param string $charset Character set to standardise
298      * @return string Standardised name
299      */
300     public static function encoding($charset)
301     {
302         return simplepie\Encoding::encoding($charset);
303     }
304
```

Figure 8. The encoding function for the SimplePie_Misc class (notice the encoding function now occurs in the SimplePie_Misc.php file instead of the simplepie.inc.php file, and all it does now is call a new function called encoding in the Encoding class within the simplepie namespace)

Encoding.class.php

```
1   <?php
2   namespace simplepie;
3
4   class Encoding
5   {
6       public static function encoding($charset)
7       {
8           // Normalization from UTS #22
9           switch (strtolower(preg_replace('/(?:[^a-zA-Z0-9]+|([^0-9])0+)/', '\1', $charset))) {
10              case 'adobestandardencoding':
11              case 'csadobestandardencoding':
12                  return 'Adobe-Standard-Encoding';
13
14              case 'adobesymbolencoding':
15              case 'cshppsmath':
16                  return 'Adobe-Symbol-Encoding';
17
```

Figure 9. The new encoding function created in the new class and file called Encoding (notice the code is identical to Figure 7)

TESTING

Fowler et al. (1999) highly recommend testing after refactoring code to validate that the of the program functionality has not been changed. Testing was performed by the researcher to validate the before and after program functionality was not affected by the refactoring process, which is the definition of refactoring. Additionally, the code authors were contacted to validate further that the code works correctly. Any alterations made to program code should undergo tests to ensure the program continues to execute as per the author's intent and user requirements.

DEMOGRAPHICS

The goal of this study was to identify the architectural changes in an existing tool that was designed to make PHP code more readable and, therefore, easier to maintain. The current instrument selected for this study is the PHP Code Sniffer (PHP Group, 2018a). A selection of a set of recently updated open source PHP programs was the first step in identifying the changes needed in the PHP Code

Sniffer program. These programs were selected based on how recently they were modified to ensure this study examines current systems and programming practices. This study sampled a total of 15 applications. Table A1 (Appendix A) represents the demographics of the chosen programs. The size column represents the size of each program in a compressed format (i.e., zip or tar). The updated column displays the date when the application was last updated. The source column lists the download location for each program.

PRESENTATION OF THE DATA

Each program was analyzed for maintainability before any changes using the PHP metrics tool. The open source program PHP metrics is a tool used by this researcher to quantify the maintainability of a piece of software written in PHP (Lepine, 2015). It returns a value from 0 to 221, with the higher score indicating better maintainability. The maintainability index formula used in this tool is:

$$MI = 171 - 5.2*\log2(V) - 0.23*CC - 16.2*\log2(LOC) + 50*\sin(\text{sqrt}(2.4*CM)) \quad (1)$$

where

- V represents the Halstead Volume
- CC represents the cyclomatic complexity
- LOC stands for lines of code
- CM is the portion of comment lines in the code (a number from 0 to 1) (Lepine, 2015)

To increase the MI (which would improve maintainability) for any program would, therefore, require a decrease in V, CC, or LOC or an increase in CM. The formula for the Halsted volume is:

$$V = N \times \log2\ n \quad (2)$$

where

- N is program length = total number of operators + total number of operands
- n is program vocabulary = number of distinct operators + number of distinct operands

The formula for cyclomatic complexity is:

$$CC = E - N + 2P \quad (3)$$

where

- E = the number of graph edges
- N = the number of graph nodes
- P = the number of connected components

The maintainability index (MI) was computed for each module within each program as well as the systems' average cyclomatic complexity (CC). Table A2 (Appendix A) lists the results of these computations for each module. The overall system's average CC is on the P-CC column. The module with the lowest MI and highest CC within each program (i.e., the module that was the least maintainable) is listed in the module column and will be the focus of this study. The module's CC is in the M-CC column, and the MI column lists the module's MI.

According to Fowler et al. (1999), refactoring code to improve maintainability can take any of the following forms:

- Composing Methods
- Moving Features Between Objects
- Organizing Data
- Simplifying Conditional Expressions
- Making Method Calls Simpler
- Dealing with Generalization

The MI will experience an increase for any module by decreasing program length (operators, operands, lines of code), decreasing cyclomatic complexity (the number of logical paths), or increasing the number of comment lines. From the list of refactoring types by Fowler et al. (1999), and considering the MI formula, the refactoring forms that are most likely to affect the MI are composing methods and moving features between objects.

Composing methods (extract method, replace temp with a query, replace the method with method object, substitute algorithm) will reduce the lines of code in a module, or the cyclomatic complexity of a module, or lessen the Halsted volume. The extract method takes a code fragment and turns it into a method, reducing the complexity of the code. Replace temp with a query is where an expression used in multiple places as a temp variable turns into a method. Replace method with method object is where a long method changes into its object. A separate module declares the new object that has improved maintainability, and the modified code reduces in size and complexity. Any of these refactoring actions will result in an increased MI.

Moving fields between objects (extract class, inline class, hide delegate). The extract class method is where a class is doing too much and should split into two (or more) classes. Then each class will do less, resulting in a decreased cyclomatic complexity factor. The inline class refactoring method is when a class is doing so little that the code it contains should move to an existing class and then remove the class. In this manner, inline class refactoring reduces the length of the program (lines of code) and therefore increases the MI.

Table A3 (Appendix A) represents the results of the tests, applying refactoring techniques as described above. Each module also represents an object class definition. The original maintainability index (MI) of the module is in the Original column. The Post-sniffer column displays the effect of applying the PHP Code Sniffer to the maintainability index. The Post-refactor column displays the MI after any possible refactoring on the module. In some cases, no refactoring was possible, or refactoring did not affect the MI. The types of refactoring techniques applied to the modules form the themes of this study.

Theme 1: Replace the recurring code

In the Sync module for addressbookv9.0.0.1, recurring (duplicated) program instructions was replaced twice with an array and a single set of program instructions. This action improved the MI from 32.4 to 33.06, whereas the PHP Code Sniffer (applying standards PSR1 and PSR2) only increased it to 32.58. Similar changes were made to the base module in moodle-auth_oidc-master, increasing the MI from 45.36 to 45.85. This technique was also applied in the PowerBBFunctions class in PBBoard_v3.0.2, improving the MI from 23.99 to 24.23. Last, this technique improved MI from 27.37 to 27.63 for CAcion in zabbix-3.4.12 by creating a new function with appropriate comments. These are examples of the replace temp with a query method which is a composing method refactoring type.

Theme 2: Move code to another class

Further maintenance fixes in the Sync module for addressbookv9.0.0.1 were to move a set of program code from the Sync class Handle function into a new function in the SyncParameters class. This action increased the MI for Sync to 35.24. In the case of the SimplePie_Misc class in the AShopGPL600 program, moving the encoding function into a separate new class improved the MI from 18.85 to 28.45. Similarly, for Mail_mimePart, MI went from 39.18 to 40.19 when the encoding function moved to a new class. These are examples of the extract class method which is within the moving fields between objects group of types of refactoring.

Theme 3: Use PHP functions

For the Person class in the iaddressbook-3.1 program, the get_array function was returning an array where each element corresponded to a class variable. Using the built-in PHP function get_class_vars

enables us to get the list of class variables in an array, and then we can loop through the array to do the assignments in the return array rather than assigning each one manually, reducing the function from 35 lines down to 7 lines of code. Similarly, the set_array function was setting each class variable from an input array (i.e., the reverse of get_array). In this case, we loop through the array elements and assign the class variables from the array keys, since the array keys were identical to the class variable names. This change reduced the function from 38 lines down to 10 lines of code. These are examples of the substitute algorithm method which is another type of composing refactoring method.

Theme 4: Large classes

There were some cases where the classes were performing the exact set of instructions as they needed to do, and each function optimally performed their assigned function. In these cases, the low MI was simply due to the large size of the class file (i.e., the number of lines of code) because of a large number of functions in the class. MI could improve in such cases by splitting the class into multiple classes and assigning a certain number of functions to each of these sub-classes. However, doing so would not significantly increase maintainability from a programmer's point of view since they now would have to look in multiple places (classes or files) for various functions. Therefore, although the MI would be lower for the changed class and each of its subclasses, in reality, the maintainability from the coder's viewpoint has worsened except possibly if this situation (i.e., directing the programmer to the other subclasses) had this documented in all the classes and subclasses. These cases included easyauditbundle-2018-07-04 (SubscriberPass), mibew-3.2.0 (ThreadProcessor), php_ci-expense_manager-script (CI_Email), phpcollab-v2.6.3 (Cpdf), phplist-3.3.3 (kcfinder\browser), qdPM_9.1 (Spreadsheet_Excel_Reader), and t3bot-2018-09-01 (ReviewCommand). Therefore, in these cases, no refactoring occurred. However, note that maintainability can always improve by adding comments to any piece of code, which would especially improve MI for these modules which had very few comments: kcfinder\browser, Spreadsheet Excel_Reader, and ReviewCommand.

PRESENTATION AND DISCUSSION OF FINDINGS

Data analysis consisted of delving into each module and looking for opportunities to improve maintainability by applying one or more of the refactoring techniques described by Fowler et al. (1999). The untouched version of each module measured maintainability using the MI by the PHP metrics tool. Then each module had the PSR1 and PSR2 standards applied to it by the PHP Code Sniffer tool, and the PHP metrics tool measured maintainability once more. Last, if any refactoring was possible, it was manually applied to the module, and the MI was again measured and recorded. The types of refactoring applied to the modules formed the themes mentioned in the previous section.

Table A4 (Appendix A) presents a summation of the results of this study. Each module selected for this study, which is also a class definition, is listed in the Module column. The LOC column represents the number of lines of code in the module. The theme number is what type of refactor method applied to the class. In this final table, the PHP Code Sniffer was modified according to theme two, and the results of running this modified version on the maintainability index were recorded. The last three columns are the maintainability indexes before any changes, after the PHP Code Sniffer changes, and after the modified version of PHP Sniffer respectively.

Theme 1, replacing recurring code, had only a minor effect on each module's MI. Theme 2, where code moved to another class, had a very significant impact in one case (SimplePie_Misc) and a moderate effect in another case (Mail_mimePart). Theme 3, using PHP functions, could only be applied in one case (Person) and had a modest impact on MI. Last, Theme 4 occurred in most cases, whereby no refactoring seemed possible, suggesting that the MI was already at the maximum level for these modules.

To answer the research question, what architectural changes to the current existing quality improvement tool are needed to establish better maintainability for PHP programs, from the above data it appears that refactoring by moving code from one class to another (or creating a new class from such

code) has the most significant effect on improving code maintainability for PHP programs. To accomplish this task in the current quality improvement tool PHP Code Sniffer involves these steps:

1. Identifying a block of code that is suitable to move to another class
2. Identifying the class that the code can be moved to (or creating a new class)
3. Creating the function in the class
4. Moving the code from the old class to the new function
5. Making the call out to the new function from the moved code location

Identifying the block of code to be moved involves looking for a substantial chunk of code within a control block such as an "if", "while", or "case" statement. Then, make sure that the function uses no class variables. Identifying the class to move the code block to involves discovering the variables and functions used in the code block, and then seeing if they are available or belong to another class. The number of lines of code is also a factor in this process. Through testing, it was determined that a minimum number of lines of code for refactoring would be 100. If all these criteria are met, then a new class is created. Creating the function involves giving the function an appropriate name and passing in parameters and determining what is to be returned by the function (if anything).

FINDINGS AND CONCLUSIONS

Software quality is an important aspect for developers to be aware of since the consequences of inferior quality code can be catastrophic and costly (Jee, 2018; Lake, 2010). Specifically, maintainability is an essential aspect of quality because programs that are difficult to maintain are prone to have errors in them, or have errors introduced by programmers that are unfamiliar with various parts of the system (Coleman et al., 1994). The population for this study consisted of 15 randomly selected open source PHP programs that were updated or released within the past 18 months. The methodology used for this study was exploratory qualitative design science. This study focused on how an artifact (a PHP program quality improvement tool) could be modified such that it can automatically improve the maintainability of PHP programs.

The proposition for this study was that maintainability of PHP code could improve by making architectural changes to the selected PHP program code quality improvement tool, PHP Code Sniffer. Based on the results of this study, it appears that this goal was achieved, within certain limitations. During the study, it was discovered that by applying various types of refactoring, the maintainability index (MI) improved in 7 of the 15 programs. The specific enhancements to the PHP Code Sniffer tool that achieved the greatest improvement in maintainability in 2 of the programs are explained in the presentation and discussion of findings section of this paper. This result provides evidence that an automated tool can enhance the quality of code by increasing its maintainability, as measured by a widely accepted quality metric, the maintainability index.

IMPLICATIONS FOR PRACTICE

Programmers and information technology project managers who work with PHP programs can benefit from this study. By applying the code changes mentioned in the PHP code sniffer changes section, the PHP Code Sniffer will improve maintainability by splitting files containing multiple classes into individual class files and creating new classes for the largest of functions within each class. These refactoring actions will improve code maintainability by reducing the lines of code within each file that a programmer has to view, understand, modify, and test. This tool provides a useful first step in improving maintainability. Further maintainability improvements would require programmers to scrutinize the program to implement other types of code changes and add informative comments. At a minimum, each class and function should have comments describing what they are there for and the use of parameters and other significant variables and structures.

Modifying code in any manner requires testing to ensure the program maintains its functionality. Veteran programmers are fully aware of the pitfalls of implementing any untested code. Therefore, it is

highly recommended that developers using this tool thoroughly test the changes, since the tool significantly restructures the code. Automated tools should not take the place of testing and quality assurance processes. This tool could be an important step towards refactoring code to make it more maintainable, but it should not replace good programming and quality assurance practices and procedures. Instead, it is a tool that can be used to encourage programmers to see how they can take existing systems written in PHP and improve their maintainability.

Limitations of the Study

Limitations for this study include limiting the study to focus on a measurable code quality characteristic, the quality of the code randomly selected for this study, and the availability of automated tools. Another limitation was that this study focused on the maintainability index, while other factors such as the human assessed COCOMO II model can also affect maintainability (Chen et al., 2016). Other limitations are the size of programs that the selected tool (PHP Code Sniffer) could handle, and lastly, that the refactoring method assumes that the function removed from the existing class will not be using any class variables and instead, all the variables needed by the function will be passed in as parameters.

Summary and Conclusions

The purpose of this design science qualitative exploratory study was to discover the architectural changes to the current existing quality improvement tool that are needed to establish better maintainability for PHP programs. The quality improvement tool selected for this study was PHP Code Sniffer (PHP Group, 2018a). The quality characteristic selected for improvement was maintainability, using the maintainability index (MI). The tool used to measure MI was PHP Metrics (Lepine, 2015).

Implications of Study and Recommendations for Future Research

This qualitative design science study aimed to discover the architectural changes to be made to an existing quality improvement tool that, when run with the proposed changes, would result in improving the maintainability of PHP programs. The changes to the PHP Code Sniffer as identified in the data analysis section of this paper improved the maintainability of several programs, as measured by the maintainability index (MI). However, from this researcher's standpoint, this represents a first step towards developing automated tools, processes, and procedures that would improve PHP program maintainability.

The implications of this study are first that it is possible to design tools such as the one in this study to automatically improve the design of PHP programs. Second, programmers can make use of the various foundational principles of code refactoring mentioned by Fowler et al. (1999) to constantly refactor their code to improve its quality and, particularly, its maintainability during the design and development stages of the software development lifecycle. Third, in addition to conforming PHP code to accepted standards such as PSR1 and PSR2, the redesigned tool can also be extended to improve program maintainability. Fourth, the improved PHP Code Sniffer can be further enhanced and expanded as outlined below.

For future research, this study could be expanded in the following ways:

- Expand the enhancements to include functions that make use of one or more class variables, by including them in the function's parameter list.
- Automatic selection of the classes with the worst MI.
- Include other refactoring methods as mentioned in Fowler et al. (1999).
- Look at implementing similar techniques for other popular programming languages.

- Run a quantitative study using a large sample size to examine how effective these changes to the tool are in the broader programming community.

The limitations section mentioned that the use of class variables precluded the selection of a function for refactoring since this would involve a second pass through the program and complicate the changes. The parameter list of the function would need to be altered to include the class variable as a new passed variable, and then all occurrences of the class variable within the new function would need to be changed to the passed variable name. With some extra effort and programming knowledge and skill, these changes could be made, resulting in more functions being refactored and the possibility of further improving program maintainability.

PHP Metrics (Lepine, 2015) was another tool used to measure the maintainability index. If that tool could be combined with the modified PHP Code Sniffer tool, then the PHP Metrics tool could pass the set of classes with the lowest maintainability index score to the Code Sniffer tool. These actions would improve the efficiency of the altered PHP Code Sniffer tool since only those classes with the worst MI would be prime candidates for refactoring. The modified PHP Code Sniffer tool currently attempts to refactor those top three functions in each class with the largest number of lines of code, which is one of the significant factors in the MI formula.

Fowler et al. (1999) identified six different types of code refactoring (composing methods, moving features between objects, organizing data, simplifying conditional expressions, making method calls simpler, and dealing with generalization). This study focused on one refactoring type (moving features between objects) since that was the method that could be readily applied to 3 of the code samples and showed a marked MI improvement. Larger samples sizes or samples with other programs could identify other types of refactoring that might show similar or even greater improvements in MI.

Although PHP is the most popular web programming language (Netcraft, 2016), other object-oriented programming languages deserve attention and could benefit from improved maintainability. These include, but are not limited to, Java, Python, Ruby, and C++. The concepts of maintainability and the MI remain the same regardless of programming language or environment. Therefore, a tool like the one in this study could be created to improve maintainability for any of these object-oriented languages. Due to the nature of these changes, the selected language must be object-oriented, to make use of class and function structures.

This qualitative exploratory study used a small sample of programs to examine the feasibility of applying code changes to an existing code quality improvement tool. A logical next step in research would be to examine how effective these changes would be in a much larger sample size, to understand their significance to the larger programming community. It would be helpful to see the extent of MI improvement in a large pool of programs using the same modified tool, or perhaps another version of the tool.

REFERENCES

Aryanto, K. Y., Broekema, A., Langenhuysen, R. G. A., Oudkerk, M., & van Ooijen, P. M. A. (2015). A web-based institutional DICOM distribution system with the integration of the clinical trial processor (CTP). *Journal of Medical Systems, 39*(5), 1-8. https://doi.org/10.1007/s10916-014-0186-y

Asadi, M., & Rashidi, H. (2016). A model for object-oriented software maintainability measurement. *International Journal of Intelligent Systems and Applications, 8*(1), 60. https://doi.org/10.5815/ijisa.2016.01.07

Aversano, L., & Tortorella, M. (2013). Quality evaluation of floss projects: Application to ERP systems. *Information and Software Technology, 55*(7), 1260-1276. https://doi.org/10.1016/j.infsof.2013.01.007

Barbaud, J. (2018). *PHP recursive directory path with RecursiveIteratorIterator in combination with RecursiveDirectoryIterator.* https://gist.github.com/jipeprojects/4608519

Boddy, C. R. (2016). Sample size for qualitative research. *Qualitative Market Research, 19*(4), 426-432. https://doi.org/10.1108/QMR-06-2016-0053

Boehm, B. W. (1978). *Characteristics of software quality*. Elsevier.

Boehm, B. W., Brown, J. R., & Lipow, M. (1976). Quantitative evaluation of software quality. In R. Yeh & C. V. Ramamoorthy (Eds.), *ICSE '76 Proceedings of the 2nd international conference on Software engineering* (pp. 592-605). IEEE Computer Society Press.

Brookshear, G., & Brylow, D. (2014). *Computer science: An overview* (12th ed.) https://bookshelf.vitalsource.com/#/books/9781323135648/

Campbell, B., Iyer, S., & Akbal-Delibas, B. (2012). *Introduction to compiler construction in a Java world.* CRC Press. https://doi.org/10.1201/9781482215076

Capiluppi, A., Boldyreff, C., Beecher, K., & Adams, P. J. (2009). Quality factors and coding standards – A comparison between open source forges. *Electronic Notes in Theoretical Computer Science, 233*, 89. https://doi.org/10.1016/j.entcs.2009.02.063

Chen, C., Alfayez, R., Srisopha, K., Shi, L., & Boehm, B. (2016). Evaluating human-assessed software maintainability metrics. In L. Zhang & C. Xu (Eds.), *Software engineering and methodology for emerging domains* (pp. 120-132). Springer. https://doi.org/10.1007/978-981-10-3482-4_9

Coleman, D., Ash, D., Lowther, B., & Oman, P. (1994). Using metrics to evaluate software system maintainability. *Computer*, *27*(8), 44-49. https://doi.org/10.1109/2.303623

Conboy, K. (2010). Project failure *en masse*: A study of loose budgetary control in ISD projects. *European Journal of Information Systems, 19*(3), 273-287. https://doi.org/10.1057/ejis.2010.7

Denning, P. J. (2016). Software quality. *Communications of the ACM, 59*(9), 23-25. https://doi.org/10.1145/2971327

Đorđević, N. D. (2017). Usability: Key characteristic of software quality. *Vojnotehnicki Glasnik, 57*(2), 513-529. https://doi.org/10.5937/vojtehg65-11028

Elish, M. O., & Elish, K. O. (2009, March). Application of TreeNet in predicting object-oriented software maintainability: A comparative study. *Proceedings of the 13th European Conference on Software Maintenance and Reengineering* (pp. 69-78). IEEE. https://doi.org/10.1109/CSMR.2009.57

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the design of existing code.* Addison-Wesley Professional.

Freyja. (2017, March 22). How much could software errors be costing your company? *Developer Tips.* https://raygun.com/blog/cost-of-software-errors/

Ganpati, A., Kalia, A., & Singh, H. (2012). A comparative study of maintainability index of open source software. *International Journal of Emerging Technology and Advanced Engineering, 2*(10), 228-230.

Goldkuhl, G. (2002, April). Anchoring scientific abstractions–ontological and linguistic determination following socio-instrumental pragmatism. *Proceedings of European Conference on Research Methods in Business, Reading.* https://www.researchgate.net/profile/Goeran_Goldkuhl/publication/268371288_Anchoring_scientific_abstractions_-_ontological_and_linguistic_determination_following_socio-instrumental_pragmatism/links/5592907208aed7453d46199f.pdf

Gregor, S., & Hevner, A. R. (2013). Positioning and presenting design science research for maximum impact. *MIS Quarterly, 37*(2), 337-355. https://doi.org/10.25300/MISQ/2013/37.2.01

Haigh, M. (2010). Software quality, non-functional software requirements and IT-business alignment. *Software Quality Journal, 18*(3), 361-385. https://doi.org/10.1007/s11219-010-9098-3

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. *MIS Quarterly, 28*(1), 75-105. https://doi.org/10.2307/25148625

Independent Software. (2016). *Introduction to PSR-1. The PHP standard recommendation.* http://www.independent-software.com/introduction-to-php-standard-recommendation-psr-1.html

Irani, Z. (2010). Investment evaluation within project management: An information systems perspective. *The Journal of the Operational Research Society, 61*(6), 917-928. https://doi.org/10.1057/jors.2010.10

ISO-9126. (1991). *Software product evaluation – quality characteristics and guidelines for their use.*

Jee, C. (2018, February 16). *Top software failures - the worst software glitches in recent history.* https://www.computerworlduk.com/galleries/infrastructure/top-software-failures-recent-history-3599618/

Johnson, N. (2018). How developer skills can keep pace with tech change. *Salesforce.* https://www.salesforce.com/blog/2018/03/citizen-development-anna-rodriguez.html

Jones, C. (2015). Wastage: The impact of poor quality on software economics. *Software Quality Professional*, *18*(1), 23–32. http://asq.org/pub/sqp/

Kaur, A., Kaur, K., & Pathak, K. (2014, October). A proposed new model for maintainability index of open source software. *Proceedings of 3rd International Conference on Reliability, Infocom Technologies and Optimization* (pp. 1-6). IEEE. https://doi.org/10.1109/ICRITO.2014.7014758

Kitchenham, B., & Pfleeger, S. L. (2002). Principles of survey research: part 5: Populations and samples. *ACM SIGSOFT Software Engineering Notes, 27*(5), 17-20. https://doi.org/10.1145/571681.571686

Lake, M. (2010, September 9). *Epic failures: 11 infamous software bugs.* https://www.computerworld.com/article/2515483/enterprise-applications/epic-failures--11-infamous-software-bugs.html?page=5

Lepine, J. (2015). *Maintainability index.* http://www.phpmetrics.org/documentation/how-to-understand-metrics.html#mi

Lockhart, J. (2016). *PHP: The right way*. http://www.phptherightway.com/

Lu, Y., Kuo, C., & Huang, Y. (2011). WebBio, a web-based management and analysis system for patient data of biological products in hospital. *Journal of Medical Systems, 35*(4), 579-84. https://doi.org/10.1007/s10916-009-9394-2

Malhotra, R., & Chug, A. (2016). Software maintainability: Systematic literature review and current trends. *International Journal of Software Engineering and Knowledge Engineering, 26*(08), 1221-1253. https://doi.org/10.1142/S0218194016500431

Mazaika, K. (n.d.). *Is the end of code really coming?* http://blog.thefirehoseproject.com/posts/end-of-code/

McKendrick, J. (2015, June 11). *Under pressure: Enterprises want better software, delivered faster and cheaper.* https://www.zdnet.com/article/under-pressure-enterprises-want-ever-more-sophsiticated-software-delivered-faster/

Misra, S. C. (2005). Modeling design/coding factors that drive maintainability of software systems. *Software Quality Journal, 13*(3), 297-320. https://doi.org/10.1007/s11219-005-1754-7

Moha, N. (2007, October). Detection and correction of design defects in object-oriented designs. *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion,* 949-950. https://doi.org/10.1145/1297846.1297960

Netcraft. (2016). *PHP just grows & grows.* https://news.netcraft.com/archives/2013/01/31/php-just-grows-grows.html

PHP Framework Interop Group. (2018). *PHP standards recommendations.* https://www.php-fig.org/psr/

PHP Group. (2018a). *PHP_CodeSniffer.* https://pear.php.net/package/PHP_CodeSniffer/

PHP Group. (2018b). *PHP: History of PHP – manual.* http://php.net/manual/en/history.php

PHP Group. (2018c). *Manual :: Coding standards.* https://pear.php.net/manual/en/standards.php

PHP Group. (2018d). *Manual :: The RecursiveArrayIterator class.* http://php.net/manual/en/class.recursivearrayiterator.php

PHP Group. (2018e). *Manual :: realpath.* http://php.net/manual/en/function.realpath.php

Ricca, F., & Tonella, P. (2005). Web testing: A roadmap for the empirical research. *Proceedings of the 2005 Seventh IEEE International Symposium on Web Site Evolution (WSE'05)*, 63-70. https://doi.org/10.1109/WSE.2005.23

Rodríguez, P., Haghighatkhah, A., Lwakatare, L. E., Teppola, S., Suomalainen, T., Eskeli, J., Karvonen, T., Kuvaja, P., Verner, J. M., & Oivo, M. (2017). Continuous deployment of software intensive products and services: A systematic mapping study. *Journal of Systems and Software*, *123*, 263-291. https://doi.org/10.1016/j.jss.2015.12.015

Singh, G., Singh, D., & Singh, V. (2011). A study of software metrics. *International Journal of Computational Engineering & Management, 11*, 22-27. http://ijcem.org/papers12011/12011_14.pdf

Siok, M. F. (2008). *Empirical study of software productivity and quality* (Doctoral dissertation). Available from ProQuest Dissertations & Theses Global (Order No. 3337487).

Spinellis, D., Gousios, G., Karakoidas, V., Louridas, P., Adams, P. J., Samoladas, I., & Stamelos, I. (2009). Evaluating the quality of open source software. *Electronic Notes in Theoretical Computer Science*, *233*, 5-28. https://doi.org/10.1016/j.entcs.2009.02.058

Stephens, R. (2015). *Beginning software engineering*. John Wiley & Sons.

Thomas, J. (2017, July 27). *Coding standards – Are they necessary?* http://mil-embedded.com/guest-blogs/coding-standards-are-they-necessary/

Turland, M. (2018). *PHP Class Splitter*. https://github.com/elazar/php-class-splitter

Vashist, S., & Gupta, A. (2014). A review on web security and its applications. *International Journal of Advanced Research in Computer Science, 5*(7). http://www.ijarcs.info/index.php/Ijarcs

Venable, J., Pries-Heje, J., & Baskerville, R. (2012). A comprehensive framework for evaluation in design science research. *Proceedings of the 7th International Conference on Design Science Research in Information Systems: Advances in Theory and Practice*, *Las Vegas, Nevada,* 423-438. https://doi.org/10.1007/978-3-642-29863-9_31

W3Techs (2018). *Historical yearly trends in the usage of server-side programming languages*. https://w3techs.com/technologies/history_overview/programming_language/ms/y

Wang, X., Conboy, K., & Cawley, O. (2012). "Leagile" software development: An experience report analysis of the application of lean approaches in agile software development. *Journal of Systems and Software, 85*(6), 1287-1299. https://doi.org/10.1016/j.jss.2012.01.061

Welker, K. D. (2001). The software maintainability index revisited. *CrossTalk,* 14, 18-21. http://staff.unak.is/andy/MScMaintenance0809/Lectures/Add/MIRevisited2001.pdf

Xu, B. W., & Chen, Z. Q. (2001). Dependence analysis for recursive Java programs. *ACM SIGPLAN Notices*, *36*(12): 70-76. https://doi.org/10.1145/583960.583969

APPENDIX A. TABLES

Table A1. Demographics

Program	Size	Updated	Source
addressbookv9.0.0.1	2.4	2017-10-14	https://sourceforge.net/projects/php-address-book
AShopGPL600	13.6	2018-02-08	https://sourceforge.net/projects/ashop
easyauditbundle-2018-07-04	0.9	2018-09-05	https://www.phpclasses.org/browse/package/10841/download/zip.html
iaddressbook-3.1	1.6	2018-01-04	https://sourceforge.net/projects/iaddressbook
mibew-3.2.0	3.8	2018-08-20	https://sourceforge.net/projects/mibew
moodle-auth_oidc-master	0.12	2018-05-24	https://github.com/Microsoft/moodle-auth_oidc
mrbs-1.7.1	1.5	2018-10-01	https://sourceforge.net/projects/mrbs
PBBoard_v3.0.2	5.9	2017-08-06	https://sourceforge.net/projects/pbboard
phd_2_12	0.4	2018-07-14	https://sourceforge.net/projects/phd
php_ci-expense_manager-script	3.1	2017-03-24	https://sourceforge.net/projects/php-expense-manager
phpcollab-v2.6.3	8.4	2018-08-25	https://sourceforge.net/projects/phpcollab
phplist-3.3.3	13.8	2018-08-24	https://sourceforge.net/projects/phplist
qdPM_9.1	10	2018-08-21	https://sourceforge.net/projects/qdpm
t3bot-2018-09-01	0.6	2018-09-04	https://www.phpclasses.org/browse/package/10889/download/zip.html
zabbix-3.4.12	17.5	2018-09-29	https://sourceforge.net/projects/zabbix

Note. The Size column represents the size of the downloaded zip file of the program, in megabytes.

Table A2. Maintainability indexes

Program	P-CC	Module (class)	M-CC	MI
addressbookv9.0.0.1	28.91	Sync	258	32.4
AShopGPL600	25.9	SimplePie_Misc	1002	18.85
easyauditbundle-2018-07-04	4.15	SubscriberPass	19	67.6
iaddressbook-3.1	34.6	Person	138	12.55
mibew-3.2.0	10.78	ThreadProcessor	71	50.54
moodle-auth_oidc-master	13.74	base	73	44.83
mrbs-1.7.1	24.07	Mail_mimePart	160	38.86
PBBoard_v3.0.2	42.31	PowerBBFunctions	479	23.98
phd_2_12	96	PHPMailer	254	39.7
php_ci-expense_manager-script	31.86	CI_Email	221	40.02
phpcollab-v2.6.3	28.27	Cpdf	519	32.14
phplist-3.3.3	18.54	kcfinder\browser	306	8.19
qdPM_9.1	8.94	Spreadsheet_Excel_Reader	227	23.94
t3bot-2018-09-01	6.37	ReviewCommand	22	65.08
zabbix-3.4.12	36.47	CAction	581	27.37

Note. P-CC represents the average cyclomatic complexity of the modules in the program. M-CC represents the cyclomatic complexity of the module in the Module column. MI is the maintainability index of the module in the Module column. A module corresponds to an object class definition.

Table A3. Maintainability indexes pre- and post-refactoring

Program	Module (class)	Original	Post-sniffer	Post-refactor
addressbookv9.0.0.1	Sync	32.4	32.58	35.24
AShopGPL600	SimplePie_Misc	18.85	18.85	28.45
easyauditbundle-2018-07-04	SubscriberPass	67.6	67.6	67.6[a]
iaddressbook-3.1	Person	12.55	12.55	12.98
mibew-3.2.0	ThreadProcessor	50.54	50.54	50.54[a]
moodle-auth_oidc-master	Base	44.83	45.36	45.85
mrbs-1.7.1	Mail_mimePart	38.86	39.18	40.19
PBBoard_v3.0.2	PowerBBFunctions	23.98	23.99	24.23
phd_2_12	PHPMailer	39.7	39.72	39.72[b]
php_ci-expense_manager-script	CI_Email	40.02	40.02	40.02[a]
phpcollab-v2.6.3	Cpdf	32.14	32.28	32.28[a]
phplist-3.3.3	kcfinder\browser	8.19	8.19	8.19[ac]
qdPM_9.1	Spreadsheet_Excel_Reader	23.94	23.96	23.96[ac]
t3bot-2018-09-01	ReviewCommand	65.08	65.08	65.08[ac]
zabbix-3.4.12	CAction	27.37	27.37	27.72

Note: [a]No refactoring changes could be identified that would improve the MI. [b]No improvement to the MI occurred after refactoring. [c]The only changes that could improve refactoring would be adding comments to the code.

Table A4. Summary of findings

Module (class)	LOC	Theme	Maintainability Index		
			Original	Post-sniffer	Modified-sniffer
Sync	3,449	1	32.4	32.58	32.58
SimplePie_Misc	1,196	2	18.85	18.85	28.3
SubscriberPass	193	4	67.6	67.6	67.6
Person	728	3	12.55	12.55	12.55
ThreadProcessor	815	4	50.54	50.54	50.54
base	471	1	44.83	45.36	45.36
Mail_mimePart	1,167	2	38.86	39.18	39.69
PowerBBFunctions	3,180	1	23.98	23.99	23.99
PHPMailer	2,269	4	39.7	39.72	39.72
CI_Email	2,294	4	40.02	40.02	40.02
Cpdf	3,050	4	32.14	32.28	32.28
kcfinder\browser	905	4	8.19	8.19	8.19
Spreadsheet_Excel_Reader	1,424	4	23.94	23.96	23.96
ReviewCommand	210	4	65.08	65.08	65.08
CAction	3,363	1	27.37	27.37	27.37

Note: LOC represents the number of lines of code in the module (i.e., class).

APPENDIX B

License Details

This Agreement between Grant Trudel ("You") and Elsevier ("Elsevier") consists of your license details and the terms and conditions provided by Elsevier and Copyright Clearance Center.

Print Copy

License Number	4473031194951
License date	Nov 20, 2018
Licensed Content Publisher	Elsevier
Licensed Content Publication	Electronic Notes in Theoretical Computer Science
Licensed Content Title	Evaluating the Quality of Open Source Software
Licensed Content Author	Diomidis Spinellis,Georgios Gousios,Vassilios Karakoidas,Panagiotis Louridas,Paul J. Adams,Ioannis Samoladas,Ioannis Stamelos
Licensed Content Date	Mar 27, 2009
Licensed Content Volume	233
Licensed Content Issue	n/a
Licensed Content Pages	24
Type of Use	reuse in a thesis/dissertation
Portion	figures/tables/illustrations
Number of figures/tables/illustrations	2
Format	both print and electronic
Are you the author of this Elsevier article?	No
Will you be translating?	No
Original figure numbers	Figures 1 & 10.
Title of your thesis/dissertation	EXPLORING AN EXISTING TOOL'S ARCHITECTURAL CHANGES TO IMPROVE PHP PROGRAM MAINTAINABILITY
Expected completion date	Dec 2018
Estimated size (number of pages)	120
Requestor Location	Grant Trudel 140/16 Surbiton Court Carindale, Foreign Country 4152 Australia Attn: Grant Trudel
Publisher Tax ID	GB 494 6272 12

APPENDIX C. CODE LISTINGS

```
/**
 * runProcess Execute the refactor process which firstly splits inc.php files into separate class files,
 *          then executes a refactor function which goes through each class, searching for large functions
 *          to split into separate class files. Finding any, it creates a new class with that function and
 *          modifies the original code to call the new class function.
 *
 * @param string $path  The path to run this process against
 *
 */
public static function runProcess($path)
{
   // If the path is a php file, just refactor it and exit
   if (is_file($path))
   {
      if (strpos($path,'.php') !== false)
      {
         Refactor::refactor($path);
      }
      return;
   }
   // Recursively get all the paths in an array
   $paths = Refactor::getPaths($path);
   // Split any inc.php files into separate php files
   // splitHappened is true if any of the files had a split
   $splitHappened = false;
   // splitThisTime is true if in this iteration, a split occurred
   $splitThisTime = false;
   foreach ($paths as $path1)
   {
      // We are only interested in inc.php files
      if ((strpos($path1,'inc.php') !== false) && is_file($path1))
      {
         Refactor::classSplitterFile($path1, $splitThisTime);
         $splitHappened = $splitHappened || $splitThisTime;
      }
   }
   // If the directories have changed, get them again
   if ($splitHappened) $paths = Refactor::getPaths($path);
   // Perform the refactoring process
   foreach ($paths as $path1)
   {
      if (strpos($path1,'.php') !== false && is_file($path1))
      {
         Refactor::refactor($path1);
      }
   }
} // end function runProcess
```

Figure C1. The runProcess function

```
  /**
   * getPaths   Get the paths of a directory recursively
   * @param string $dir     The directory
   * @return array $return  An array of all the paths in the directory
   */
  public static function getPaths($dir)
  {
     $path = realpath($dir);
     $return = array();

     $objects = new \RecursiveIteratorIterator(new \RecursiveDirectoryIterator($path), \RecursiveIterator
torIterator::SELF_FIRST);
     foreach($objects as $name => $object)
     {
        $return[] = $name;
     }
     return $return;
  } // end function getPaths
```

Figure C2. The getPaths function

```
/**
 * classSplitterFile Take a file containing multiple classes and create
 *                   separate class files, one class per file
 *
 * @param string $file           The input file (code assumes the file
 *                               has 'inc.php' in it)
 * @param boolean $splitThisTime  True if split occurred on this call,
 *                               else false
 */
public static function classSplitterFile($file, &$splitThisTime)
{
   $splitThisTime = false;
   // If input is not a file, exit
   if (!is_file($file)) return;
   // If we found a class then set buffer to true
   $buffer = false;

   // Only split into multiple class files if there are more than 1
   // classes in the file
   if (Refactor::countClasses($file) > 1)
   {
      $splitThisTime = true;
      $tokens = token_get_all(file_get_contents($file));
      // $nonclasstokens is a string containing non-class tokens
      $nonclasstokens = '';
      // $docarr is an array of doc domments in the form /** ... */
      $docarr = array();
      // $docs is an array of doc comments whose key is the function
      // name
      // and contents is a string of doc comments for the function
      $docs = array();
      // boolean to tell when we have found a class token
      $classfound = false;
      // the current comment line
      $commline = '';
      // the current class line
      $classline = '';
      // the last comment line
      $lastcommline = 0;
      while ($token = next($tokens))
      {
         if (!$classfound && $token[0] != T_CLASS)
            $nonclasstokens .= is_string($token) ? $token : $token[1];
         // Store the comment lines in array docarr
         if ($token[0] == T_DOC_COMMENT)
         {
            $tokenvalue = is_string($token) ? $token : $token[1];
            $doccomments = explode(PHP_EOL, $tokenvalue);
            $docarr[] = $tokenvalue;
            $commline = $token[2]+count($doccomments);
            if (!$classfound) $lastcommline = count($doccomments)+1;
         }
         if ($token[0] == T_CLASS)
         {
            $classfound = true;
            $buffer = true;
            $name = null;
```

```
        $code = '';
        $braces = 1;
        $classline = $token[2];
        do
        {
          $code .= is_string($token) ? $token : $token[1];
          if (is_array($token)
            && $token[0] == T_STRING
            && empty($name))
          {
            $name = $token[1];
          }
        }
        while (!(is_string($token) &&
            $token === '{') &&
            !(is_array($token) &&
            $token[1] == '{') &&
            $token = next($tokens));
        // Only include doc note if the line positions match
        // else we have the comment of some other class or
        // function
        if ( $classline == $commline )
        {
          $docs[$name] = end($docarr);
        }
      }
}
// Remove any comment code belonging to the last class
// Remember class0 contains non-class code only
$codes = explode(PHP_EOL,$nonclasstokens);
$codes = array_splice($codes, 0, -$lastcommline);
$nonclasstokens = implode(PHP_EOL,$codes);
// buffer is true when we find a class
$buffer = false;
// The string containing the list of class file names
$classnames = '';
$tokens = token_get_all(file_get_contents($file));
while ($token = next($tokens))
{
  if ($token[0] == T_CLASS)
  {
    $buffer = true;
    $name = null;
    $code = '';
    $braces = 1;
    do
    {
      $code .= is_string($token) ? $token : $token[1];
      if (is_array($token)
        && $token[0] == T_STRING
        && empty($name))
      {
        // store the name of the class
        $name = $token[1];
      }
    }
    while (!(is_string($token) &&
```

```
            $token === '{') &&
            !(is_array($token) &&
            $token[1] == '{') &&
            $token = next($tokens));
    } elseif ($buffer)
    {
       if (is_array($token))
       {
          $token = $token[1];
       }

       $code .= $token;

       if ($token == '{')
       {
          $braces++;
       } elseif ($token == '}')
       {
          $braces--;
          if ($braces == 0)
          {
             // Create the directory if it does not exist yet
             // if file is x.y.inc.php then dir is x_y
             // else if file is x.inc.php then dir is x
             $arr0 = explode('/', $file);
             $arr1 = explode('.', end($arr0));
             $dir = '';
             foreach($arr1 as $arr)
             {
                if ($arr == 'inc')
                {
                   break;
                }
                $dir .= $arr . '_';
             }
             $dir0 = substr($dir, 0, -1);
             $dir = dirname($file) . '/' . $dir0;
             if (!is_dir($dir))
             {
                mkdir($dir);
             }
             $buffer = false;
             // filex is the directory plus the name of the
             // class plus .php
             $filex = $dir . '/' . $name . '.php';
             // $doc1 contains the doc comments for the
             // function if they exist
             // otherwise an empty string
             $doc1 = isset($docs[$name]) ? $docs[$name] : '';
             // code will consist of any doc comments plus the
             // class code
             $code = '<?php'
                . PHP_EOL
                . $doc1
                . PHP_EOL
                . $code;
             // write the new class file
```

```
                    file_put_contents($filex, $code);
                    // add the include to the classnames which will
                    // be
                    // used to replace class code with includes in
                    // the original file
                    $classnames .= 'include "' . $dir0 . '/'
                            . $name . '.php";' . PHP_EOL;
                }
            }
        }
    }
    $file1 = fopen($file, 'w');
    if ($file1 === false)
    {
        echo 'fopen failed on file ' . $file . PHP_EOL;
        die();
    }
    // add the string of class includes to the end of the original
    // file
    // thereby replacing the class code with includes
    fwrite($file1, '<?php' . $nonclasstokens . PHP_EOL
          . $classnames);
    fclose($file1);
  }
} // end function classSplitterFile
```

Figure C3. The classSplitterFile function

```
/**
 * countClasses   Count the number of classes in a file
 * @param  string $file       The input file
 * @return int    $countclass  The number of classes in the file
 */
public static function countClasses($file)
{
  $countclass = 0;
  if (!is_file($file)) return $countclass;
  $tokens = token_get_all(file_get_contents($file));
  while ($token = next($tokens))
  {
    if ($token[0] == T_CLASS)
    {
      $countclass++;
    }
  }
  return $countclass;
} // end function countClasses
```

Figure C4. The countClasses function

```
/**
 * refactor            Count the number of lines in each function for a
 *                     class file and if over 100
 *                     then call a function to replace the code with a
 *                     function call to a new class
 * @param string $file   The file to be refactored
 *
 */
public static function refactor($file)
{
   $return = array();
   // If not a file then exit
   if (!is_file($file)) return $return;
   // If not more than one class in the file then exit
   if (Refactor::countClasses($file) < 2) return;
   // Get all the tokens
   $tokens = token_get_all(file_get_contents($file));
   $orig = file_get_contents($file);
   $tokens = token_get_all($orig);
   $linecount = 0;
   $prevnum = 0;
   $buffer = false;
   while ($token = next($tokens))
   {
      if ($token[0] == T_FUNCTION)
      {
         $buffer = true;
         $name = null;
         $braces = 1;
         $code = '';
         do
         {
            $code .= is_array($token) ? $token[1] : $token;
            if (is_array($token)
               && $token[0] == T_STRING
               && empty($name))
            {
               // name is the function name
               $name = $token[1];
            }
         } while (!(is_string($token) &&
                $token === '{') &&
                !(is_array($token) &&
                $token[1] == '{') &&
                $token = next($tokens));
      } elseif ($buffer && $name != '__construct')
      {
         $code .= is_array($token) ? $token[1] : $token;
         if (is_array($token))
         {
            $token = $token[1];
         }
         // If there are any class variables used in the function then
         // ignore it
         if (strpos($token, '$this') !== false)
         {
            $buffer = false;
```

```
                continue;
            }
            // Counting the curly braces enables us to tell when we've
            // hit the end of the function
            if ($token == '{')
            {
                $braces++;
            } elseif ($token == '}')
            {
                $braces--;
                // If the braces count is zero then we are at the end of
                // the function
                if ($braces == 0)
                {
                    $buffer = false;
                    // The easiest method to count the lines is to
                    // write the contents to a file and then do a count
                    // on the file function
                    $filenew = $file . '_' . $name . '.php';
                    file_put_contents($filenew, $code);
                    // The file function reads the file into an array
                    $linecount = count(file($filenew));
                    // Delete the temporary file
                    unlink($filenew);
                    // If linecount is more than 100 then do replacement
                    // code function
                    if ($linecount > 100)
                    {
                        Refactor::replaceCode($file, $name);
                    }
                }
            }
        }
    }
    return;
} // end function refactor
```

Figure C5. The refactor function

```
/**
 * replaceCode   Replace code in file with a call to a new class and
 *               function
 * @param string $file           The file to be read in containing the
 *                               class and function for replacement
 * @param string $function_name   The name of the function to be replaced
 */
public static function replaceCode($file, $function_name)
{
   $dest = rtrim($file, '/');
   // Store the original file contents in a string
   $orig = file_get_contents($file);
   // Get all tokens
   $tokens = token_get_all($orig);
   $buffer = false;
   $savedcode = '';
   $savedfnparams = '';
   $savedfncode = '';
   $savedname = '';
   $var = '';
   while ($token = next($tokens))
   {
      if ($token[0] == T_FUNCTION)
      {
         $buffer = true;
         // name contains the function name
         $name = null;
         // code contains the function name, the function parameters,
         // and the function code
         $code = '';
         // fnparams contains the function parameters
         $fnparams = '';
         // fncode contains the function code
         $fncode = '';
         // braces contains the number of left curly braces found
         // helping us determine when we reach the end of the function
         $braces = 1;
         do
         {
            if (!empty($name))
               $fnparams .= is_string($token) ? $token : $token[1];
            $code .= is_string($token) ? $token : $token[1];
            if (is_array($token)
               && $token[0] == T_STRING
               && empty($name))
            {
               $name = $token[1];
            }
         } while (!(is_string($token) &&
               $token === '{') &&
               !(is_array($token) &&
               $token[1] == '{') &&
               $token = next($tokens));
      } elseif ($buffer && $name == $function_name)
      {
         if (is_array($token))
         {
```

```
            $token = $token[1];
        }
        $fncode .= is_string($token) ? $token : $token[1];

        $code .= $token;
        if ($token == '{')
        {
            $braces++;
        } elseif ($token == '}')
        {
            $braces--;
            if ($braces == 0)
            {
                // We have reached the end of the function
                // so store the data

                // Reset buffer boolean to false to find the next
                // function
                $buffer = false;
                // Save the code and function name
                $savedcode = $code;
                $savedname = $name;
                $savedfnparams = str_replace('{', '', $fnparams);
                $savedfncode = $fncode;
            }
        }
    }
}
// After the loop see if we found the function and then execute
// replace logic
if ($savedname == $function_name)
{
    // Create the new file with the function
    // This is the new class
    $classname = ucfirst($savedname);
    // Namespace will be the last dir name
    $dest = dirname($dest);
    $fila = explode('/', $dest);
    $namespace = array_pop($fila);
    // This is the new file
    $filenew = $dest . '/' . $classname . '.class.php';
    // Make sure it is a static function
    $pos = strpos($savedcode, 'function');
    if ($pos !== false)
    {
        $savedcode = substr_replace($savedcode,
            'public static function', $pos, strlen('function'));
    }
    // Here we create the code to put into the new file
    $code = '<?php'
        . PHP_EOL
        . 'namespace '
        . $namespace
        .';'
        . PHP_EOL
        . 'class '
        . $classname
```

```
            . PHP_EOL
            . '{'
            . PHP_EOL
            . $savedcode
            . PHP_EOL
            . '}';
        // Write out the contents
        file_put_contents($filenew, $code);
        // Now replace the code in the original file with the function
        // call to the new class
        $newfnparams = PHP_EOL
            . 'include \''
            . $classname
            . '.class.php\';'
            . PHP_EOL;
        // Put include at the start of original file
        $orig = str_replace('<?php', '<?php' . PHP_EOL . $newfnparams .
            PHP_EOL, $orig);
        // Insert semicolon at end of function parameters
        $savedfnparams = str_replace(')', ');', $savedfnparams);
        // Put in return function call followed by final curly brace
        $newfncode = PHP_EOL
            . 'return '
            . $namespace
            . '\\'
            . $classname
            . '::'
            . $function_name
            . $savedfnparams
            . '}';
        // Locate and perform replace to replace original function code
        // with new code (i.e. function call)
        $pos = strpos($orig, $savedfncode);
        if ($pos !== false)
        {
            $orig = substr_replace($orig, $newfncode, $pos,
                strlen($savedfncode));
        }
        // Write out the file
        file_put_contents($file, $orig);
    }
} // end function replaceCode
```

Figure C6. The replaceCode function

APPENDIX D

Greg Sherwood 26 July 2018 at 9:36 am

Re: Permission to use software in study

To: grant.trudel@student.ctuonline.edu

Hi Grant,

Yes, you can modify PHPCS in any way you need. This licence allows for this without requiring permission, distribution, or contribution back.

But thanks for asking anyway :)

Greg
On Thu, 26 Jul 2018 at 09:19, Grant Trudel <grant.trudel@student.ctuonline.edu> wrote:

> [This message has been brought to you via pear.php.net.]
>
> Greetings Mr Sherwood,
>
> Hope this email finds you well.
>
> My name is Grant Trudel and I am a doctoral student at Colorado Technical University. I am doing a study on the quality of PHP programs and I was wondering if I could please have your permission to use and modify (apply a small number of changes) to your code sniffer application for my research?
>
> Thank you and I hope to hear from you soon!
> -Grant

--
Greg Sherwood
Chief of Engineering
gsherwood@squiz.com.au

Squiz Labs Pty. Ltd. Level 2, 60 Macquarie Street Parramatta 2150
P +61 2 9045 2800 W www.squizlabs.com

AUTHORS

Dr Grant Trudel is an adjunct professor at Indiana Wesleyan University in the US as well as an active practitioner in industry based in Australia (at livepro). Besides programming, Grant has a passion for travel, hiking, golfing, tennis, and spending time with his family. He also enjoys teaching and mentoring younger programmers, having created and taught IT courses for various universities at associates, bachelors, masters, and doctoral levels. Grant's research interests include the software development life cycle, IT project management, software quality, and software design.

Dr Samuel Sambasivam is Chair and Professor of Computer Science Data Analytics at Woodbury University, Burbank, CA. He is Chair Emeritus and Professor Emeritus of Computer Science at Azusa Pacific University. His research interests include Cybersecurity, Big Data Analytics, Optimization Methods, Expert Systems, Client/Server Applications, Database Systems, and Genetic Algorithms. He served as a Distinguished Visiting Professor of Computer Science at the United States Air Force Academy in Colorado Springs, Colorado for two years. He has conducted extensive research, written for publications, and delivered presentations in Computer Science, data structures, and Mathematics. He is a voting senior member of the ACM.