2012

MEASURING AND QUANTIFYING WEB APPLICATION DESIGN

Craig A. McNinch
The University of Montana

MEASURING AND QUANTIFYING WEB APPLICATION DESIGN

By

CRAIG ALAN MCNINCH

B.S. Computer Science, The University of Montana, Missoula, Montana, 2006

Thesis

presented in partial fulfillment of the requirements
for the degree of

Master of Science
In Computer Science

The University of Montana
Missoula, MT

Official Graduation Date July 2012

McNinch, Craig, M.S., Summer 2012 Computer Science

MEASURING AND QUANTIFYING WEB APPLICATION DESIGN

Chairperson: Dr. Joel Henry

Abstract:

The World Wide Web makes it easier for organizations and people to share information across great distances at a minimal cost. In addition, governments and businesses are beginning to offer services on the World Wide Web to their respective populations and customers. Unlike traditional desktop based applications where the programming language used are most likely Object Oriented (OO) based languages such as C++ or Java, most web applications are built upon more lightweight scripting languages. These languages typically don't support all the OO features that more traditional languages support. In addition, some web applications are dependent on external web services or applications. These differences make it difficult to use traditional measuring techniques such as Quality Measurements in Object Oriented Design to quantify the complexity of a web application or its quality.

This paper will propose a set of measurements derived from traditional Object Oriented metrics such as QMOOD, and attempt to use them on two Content Management Systems; Drupal and WordPress. These measurements will try to quantify the size and complexity of the two content management systems and make comparisons between them.

Table of Contents

List of Figures:

List of Equations:

List of Tables:

1.0 Introduction

For several decades, software development revolved around object oriented (OO) based applications. With OO based design, software engineers were able to abstract and isolate sections of their code into “objects”. These objects allowed software engineers a degree of flexibility in regards to the design, implementation, and testing of their software. With the popularity of OO based languages, several tools and methods have been developed to help measure the quality of their software such as QMOOD (Bansiva, 2002). While these tools are useful for OOP based applications the usage of them breaks down when trying to use them on applications that do not conform strictly to OO standards.

This thesis will consider several such OO based measurements and attempt to use them on two web based Content Management Systems (CMS). To do this, several files from each CMS will be selected for the measurements. After the measurements have been taken, the results will be analyzed and observations will be drawn and compared.

1.1 Thesis Organization

The organization of this thesis is as follows…

- Chapter 2 will cover research that has been conducted in the past regarding measuring quality in application design and the challenges presented by today’s technologies.
- Chapter 3 introduces the Drupal and WordPress Content Management Systems and describes how each CMS is organized and built.
- Chapter 4 discusses the measurements and tools used

- Chapter 5 will list our results and any immediate observations of the measurements.
- Chapter 6 presents our analysis of the measurements and attempt to draw comparisons between WordPress and Drupal.
- Chapter 7 will contain our conclusions.

2.0 Literature Review

2.1 Traditional Measurements

Quality in application design is a critical area to measure since security, scalability, and maintainability of the application depends on it. An application with a higher measurement in design quality is generally cheaper to maintain compared to one with a lower measurement. Several papers have been published proposing such metrics for Object Oriented (OO) based applications. Chidamber and Kemerer proposed a set of measurements to measure quality in OO design (Chidamber, 1994). In that paper, they proposed the following measurements.

Weighted Methods Per Class (WMC)

WMC is a metric that measures the number of methods within the class and has a weight assigned to each method. For the weight, the McCabe's complexity measurement is generally used to determine the complexity of the method (McCabe, 1976) . The range of the measurement is 0 to N where N is a positive integer. A measurement of 0 indicates that the class contains no methods and thus is the least complex while larger values of N indicates a more complex class. This measurement will be used.

Depth of Inheritance Tree (DIT)

DIT is a metric that measures the length of the inheritance for a given class. The range of the measurements is 0 to N where N is a positive integer. A measurement of 0 indicates that the class has no inheritance thus is the least complex while larger values of N indicates a more complex class due to a deeper inheritance tree. This measurement will not be used since the measurement requires explicit class declarations.

Number of Children (NOC)

NOC is a metric that counts the number of direct children that the class has. The range of the measurement is 0 to N where N is a positive integer. A measurement of 0 indicates that the class has no children while larger values of N indicate a higher number of child classes that depends on the measured class. This measurement will not be used due to the same reasons why DIT is being omitted.

Coupling Between Object Classes (CBO)

CBO is a metric that counts the number of non-inherited connections between objects. The equation to calculate the CBO is:

Equation 1:

CBO = Number of Links / Number of Classes

The range of the measurement is from 0 to N where N is a positive number. A measurement of 0 indicates that the class has no couplings thus it is an isolated class while greater values of N indicates a class that has many dependencies. This measurement will not be used, but instead the afferent and efferent coupling measurements will be used instead.

Response for a Class (RFC)

RFC is a metric that counts the number of local methods as well as remote methods outside the class that the local methods within the class call. The range of the measurement is 0 to N where N is a positive integer. A measurement of 0 indicates a class with no local methods while greater values of N indicate a class with more local methods or local methods that access a greater number of remote methods. This measurement will be used in this paper.

Lack of Cohesion of Methods (LCOM)

LCOM is a metric that measures the commonality of the methods within a specific class. The equation to calculate the LCOM is:

Equation 2:

LCOM = P – Q, if P > Q
LCOM = 0 otherwise

Where P is the number of disjoint variables accessed by methods in the class and Q is the number of common variables accessed by methods in the class.

The range of the measurement is 0 to N where 0 indicates a cohesive class while larger values of N indicate a less cohesive class.

This measurement received a lot of criticism since it was possible to have two completely different classes with the LCOM measurement of zero. Several alternative measurements for LCOM have been proposed. Henderson-Sellers, Constantine &

Graham proposed an alternative LCOM measurement called LCOM2 (Aivosto, 2011) (Henderson-Sellers B., 1996). The definition for LCOM2 is as follows:

Equation 3:

Let M be the number of methods in a class
Let A be the number of attributes in a class
Let MA be the number of methods that accesses A
Let sum(MA) be the sum of all MA's in the class

LCOM = 1 – sum(MA) / (M * A)

The range of LCOM2 is from 0 to 1 where a measurement of 0 is undefined and a measurement of N between 0 and 1 is the percentage of methods that do not access a specific attribute. Higher values of N indicate a less cohesive class. LCOM2 will be used in this paper.

Additionally several other measurements have been proposed to measure OO design. In the report, *OO Design Quality Metrics*, Martin mentions the use of afferent couplings (AC), efferent couplings (EC), and instability (I) within classes (Martin, 1994).

AC = Number of remote classes that depends on the selected class.

EC = Number of remote classes that the selected class depends on.

I = A number between 0 and 1 which indicates how vulnerable the selected class is to change.

This paper will use the measurements proposed by Martin and details about each measurement will be discussed in chapter 4.

Finally, in the paper, *A Hierarchical Model for Object-Oriented Design Quality*, Bansyia and Davis proposed a model called QMOOD that evaluates the quality of application design under the following attributes:

- Reusability – Can the software be applied to new problems?
- Flexibility – Can changes be incorporated to the software?

- Understandability – Can the design be easily learned?
- Functionality – Can certain features be accessed in a controlled manner?
- Extendability – Can new additions be added with minimal effort?
- Effectiveness – Does the software achieve desired functionality?

While these attributes can be easily applied to most OO based designs, applying them to less than ideal OO designs could pose a challenge. However, a portion of measurements used in QMOOD are based off Chidamber and Kemerer's measurements.

2.2 Challenges with Web Based Applications

In recent years, the web has made it easier for organizations and people to share and disseminate information across great distances at minimal cost (USLegal, 2012). As more organizations utilize the web for their daily operations, the amount of data and content posted to the web increases dramatically and becomes difficult to manage. From basic blogs managed by individuals to elaborate business-to-business (B2B) applications hosted by large enterprises, the web is becoming a major conduit to conduct business and to promote social interaction (Monaco, 2012). Unlike traditional OO based applications where the design of the application is conveniently laid out as a series of objects, web based applications are typically a mix of static content, pages with embedded code, and classes. This makes it difficult to apply OO based measurements to web applications.

Several studies have been made in trying to apply measurements from OO based designs to web based applications. One study by Alessandro Marchetto looked into applying OO based measurements to web based applications (Marchetto, 2004). In the study, Marchetto proposed a suite of measurements which were derived off traditional OO measurements such as the ones proposed by Chidamber and Kemerer. In order to apply these measurements to web based applications, Marchetto applied a method to isolate code fragments and treated them as modules. From there, he applied the modified metrics to obtain his results. This paper will look into using a similar approach and treat files within a content management system as a class.

2.3 PHP and Tools

PHP is one of the most heavily used languages for web applications. PHP was originally developed by Rasmus Lerdorf in 1994 as a series of scripts to make it easier for him to manage his website (The PHP Group, 2012). PHP is a weakly typed, server-side scripting language that can be embedded in HTML to produce dynamic content. Between 2000 and 2010, there have been two major releases of the language. The first was PHP 4.0, released in 2000. PHP 4 was very popular at the time and many content management systems used it within their architecture. PHP 4 supported some Object Oriented (OO) declarations but lacked certain, explicit declarations. In 2004, PHP 5.0 was released to the public and introduced more OO features. Some of these features that PHP 5.0 included were private and protected variable declarations, abstract classes, final classes, and interfaces. In addition, setters and getters have been standardized. Additionally, the error handling in PHP 5 is significantly better than in previous versions.

Several tools have been developed for PHP that have made it easier to collect measurements from PHP based applications. PHP Depend developed by Manuel Pichler is one such tool that collects measurements across a wide range of metrics (Pichler, 2011). More information for PHP Depend can be found in chapter 4. A second tool called PHP Unit is a unit testing framework for PHP that supports several OO based measurements (Sebastian, 2011). This paper will not include PHP Unit since it is sensitive to explicit class declarations and thus makes it difficult to collect the necessary measurements. Additionally PHP Depend covers the same measurements used by PHP Unit.

3.0 The Content Management Systems

With an increasing volume of information being presented on the web, the need for tools to manage this information becomes ever more important. Content Management Systems (CMS) are one such tool that helps to make it easier for businesses and organizations to manage this information (DocForge, 2011).

A CMS is an application, typically hosted on a web server that is responsible for maintaining information on a web page or web application. Most content management systems have the following attributes:

- Security - Manage permissions to administrators, publishers, and contributors.
- Workflow - Ensure that multiple people can edit content on the CMS in parallel.
- Templating - Ability to apply visual style to content being edited.
- Modularity – Ability to extend features by adding modules.
- Media Management - Aid in the ease of storage of web assets.
- Rich Text Editing - Aid in the ease of use in standard editing of web pages. Users, without knowledge of web languages, should be able to update content on the CMS.
- Scalability – Ability to use different databases, hosting environments.
- Content Distribution – Ability to share content via APIs or other syndication methods.

In General, a CMS helps to abstract away the technical knowledge required to manage content on the web.

There are many types of Content Management Systems currently in use. They range from enterprise level, proprietary systems to free, open source systems. Two popular CMS product's that this paper will focus on is Drupal and WordPress. Drupal is a fully developed, module-based CMS that is widely used across a wide range of sites. These websites could range from complex E-Commerce sites that host complex web applications to simple blogs. WordPress, on the other hand, is a simpler CMS system that mostly focuses on blogs and sites that deal with content publication, but is also capable on managing more complex websites and web applications. Both, Drupal and WordPress uses PHP and is generally hosted in a Linux, Apache, MySQL, and PHP (LAMP) environment.

When Drupal and WordPress was created, PHP 4 was the version in use. Due to limitations with PHP 4, the developers on both Drupal and WordPress took a procedural approach in the construction of the Content Management Systems. Later, PHP 5 was released which included more object oriented programming features.

For our comparison, WordPress and Drupal were chosen since they are two of the most widely used Content Management Systems on the web (W3Techs, 2012). A lot of businesses rely on them when developing and maintaining websites, so it is beneficial if a comparison between WordPress and Drupal could be made. Another CMS called Joomla Was considered, and it was ranked as the second largest CMS on the web, but it was omitted due to time constraints and the file structure between Joomla and Drupal did not allow for a more direct comparison compared to Drupal and WordPress.

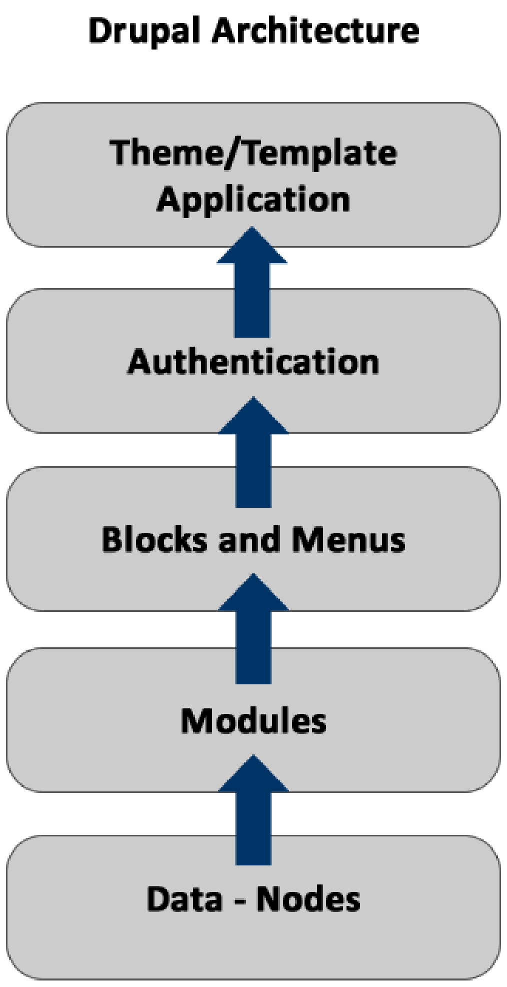
3.1 Drupal

Drupal is an open source, Content Management System that relies on modules to extend its functionality (Hunter, L. 2008). It is implemented in PHP and requires a MySQL or a PostgreSQL database. As of this writing, Drupal 6 is the current version of Drupal in use, with version 7 under development.

When Drupal was created, PHP 4.0 was in use. This made it difficult for the developers to use a completely object oriented approach to the design of the CMS, due to the limited object oriented abilities in PHP 4.0. They instead, opted for a module based system, where each module represented a class which contained “hooks” on each module for message passing. In other words, each module in the Drupal CMS will be considered a pseudo class, and the hooks would be considered event handlers to access and use that pseudo class. This helps to enforce isolation of certain Drupal core functions and encourages the use of an Application Programming Interface (API) to call Drupal modules.

Drupal works by waiting for a request through its “index.php” file. When a request is sent to Drupal, the CMS will load a pseudo bootstrapping file that loads all the necessary functions and files needed to fulfill the given request. It will then load the menu system which will determine what functionality the request requires. From there, the menu system will load the necessary node data from the database. The menu system will then call all necessary modules to process the node data. After receiving the results from the modules, the menu system will then authenticate the user submitting the request and pass the results to the theme generator. After getting the results from the theme generator, the menu system will then pass the HTML results back to the web server, which is then sent back to the client’s web browser via an HTTP response.

Figure 1: Architecture of the Drupal CMS system. Before any pages are rendered, data is retrieved via a generic data node. From there, modules can use the data for their own purposes before being passed on to the next level. On the Blocks and Menus level, Navigation items are assigned to the page before being passed through the authentication level. Once authorized, the page moves to the Theme/Template level where styling is then applied to the page.



Additionally, Drupal comes with a set of core modules. These modules contain enough functionality to start a basic website without the need of additional modules. Some of these modules are used for the following…

- authentication of users
- statistics on site usage
- management of content for blogs, forums, and polls
- applying themes to content,
- management of multiple sites
- Search Engine friendly URLs where a given URL on a Drupal site would look like http://www.domain.com/mysubpage/ instead of http://www.domain.com/node.php?ID=mysubpage
- RSS feed aggregator
- Management of user profiles such as their login, account information, and any rights that might be associated with them.

While some functionality for Drupal is in its core modules, the bulk of the work requested is handled by a set of "include" files. These include files contain the necessary functions to process the requests to the CMS, interact with the necessary modules, communicate with the database, and use the theme generator to display the results back to the user. These "include" files are the files that will be measured. Below is a list of these "include" files that the Drupal CMS uses.

Bootstrap.inc – This include file is loaded for every Drupal request. It includes all the necessary functions needed by the request. From here, other include files will be loaded.

Common.inc – Loaded by the bootstrap.inc file, contains all the common functions needed by the CMS.

Cach.inc – Handles the main caching of content on the Drupal CMS. Can be extended by using other Drupal modules.

Database.inc – General wrapper for database specific files, such as database.mysql.inc, database.mysqli.inc, or database.pgsql.inc.

File.inc – Handles the file management capabilities of Drupal. It is responsible for loading files such as theme or data files.

Xmlrpc.inc – manages the XML RPC calls. XML RPC is similar to HTTP calls, but uses XML to send the messages to the receiving application.

Unicode.inc – Manages the Unicode Character set. Unicode is an agreed standard of characters. It is generally used in international environments where ASCII, or other character sets are insufficient.

Theme.inc – The main theme engine of the Drupal CMS. Its responsibility is to take data from the layer beneath it and apply formatting to it. This will involve loading the necessary style sheets needed, and for placement of the content in defined regions.

Session.inc – Manages the session state of the user. This usually involves creating, updating and destroying user sessions.

Path.inc – Manages the paths in the Drupal CMS. It also supports path aliasing.

Menu.inc – Handles the menu system in Drupal such as navigation menus. In addition, it is responsible for the dispatching of events and calls.

Module.inc – Handles the management of modules in the Drupal CMS. Its responsibilities include the loading, enabling, disabling, and listing of modules.

Form.inc – handles the form generation, validation, and processing of forms in the Drupal CMS system.

3.2 WordPress

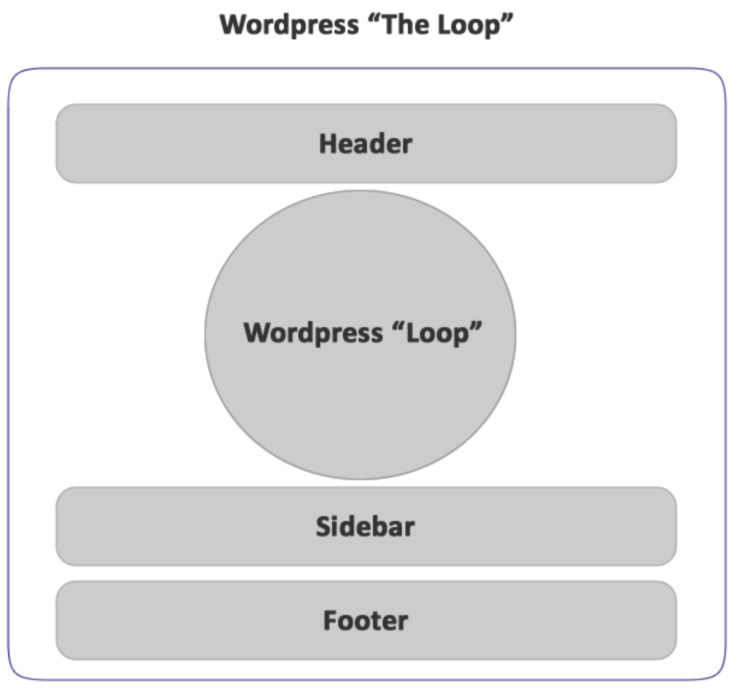
WordPress is a Content Management System that focuses on Blogs or simple websites that publishes information to the masses (WordPress.org, 2012). Unlike Drupal, WordPress doesn’t have an elaborate set of “core” modules to extend its functionality right out of the box. It, instead, relies on “plugins” to extend its functionality from the basic blog system that it was designed for. A plugin in WordPress is very similar to a module in Drupal. It is nothing more than a PHP file that accesses the WordPress API to extend its functionality. The goal in outsourcing some of its functionality to plugins is to reduce the complexity of the CMS core system and focus the core on its intended functionality which is the management and publishing of content to the blog or website.

Similar to Drupal, WordPress was implemented when PHP 4.0 was in wide use. This made it hard for the developers of WordPress to use a strictly OO based approach in developing the software. While the majority of PHP files in WordPress lack a standard class definition, there are several files that do contain classes. In addition, a level of isolation has been attempted where the code belonging to a particular function is isolated in its own PHP file and not combined with other code in the same file. As of this writing, WordPress 3.0 is the current version in use.

While WordPress is simpler than Drupal in regards to “out of the box” functionality, the underlining architecture is very similar. These similarities are: the abstraction of databases by the use of database classes, managing and applying themes to content, management of data, caching of content, Unicode support, file management, and XML Remote Procedure Call (RPC) handling. In addition, WordPress includes a common library of functions and a “bootstrapping” approach in the loading and serving of content. In addition, WordPress also uses a “hook” system to connect plugins into the core WordPress API. This is similar to Drupal’s system where modules for a particular function could interact via “hooks” to the main API.

Since WordPress is focused on managing blogs, it renders pages differently than Drupal. A big difference is that it uses a “loop” to load blog posts and comments (WordPress.org, 2012). Following the figure below, WordPress will load the header, the navigation “sidebar” and the footer. Between the header and footer, WordPress will loop through the selected list of information to be rendered on the page. Within the loop, there could be nested loops which could be used to list comments or any other information. In comparison, Drupal will not use a loop to render the page, but instead pull the data directly from the database. This is because WordPress was designed towards websites that contain multiple "entries" on a single page, such as blog entries. Drupal, on the other hand, only needs data for a specific article or data node, thus it won't use a loop to retrieve that specific piece of information.

Figure 2: A page rendered through WordPress passes through the WordPress "Loop". This occurs by rendering the header of the page first, and then loop through all entries on the page. Finally navigation items are applied to the page and the footer is then added. This is a typical process for Blogs, but can be used for individual pages since WordPress only needs to loop through the content once.



Similar to Drupal, the core of the execution in WordPress is in a set of "include" files. These files handle everything from interaction of databases to the management of comments and posts. Below is a list of important files used by WordPress to implement website functionality.

Index.php – The default page for every WordPress call. It passes the execution off to wp-blog-header.php

WP-blog-header.php – A Simple file that loads the WP-load.php file for bootstrapping and execution of the request, and then calls the template-loader.php file for styling.

WP-load.php – Similar to Drupal's bootstrapping file, this file handles the bootstrapping of modules and loads any necessary functions. This page is executed on every page call.

WP-settings.php – While WP-load.php is defined as the bootstrapping file in WordPress, this file will do the actual work in loading the necessary modules, classes, and any other procedural functions to process the request.

Bookmark.php – Include file that handles the bookmarking of blog posts and the "permalinks" associated with every post.

Cache.php – Handles the caching of pages and posts. Similar to Drupal, this include could be extended with Plugins. In addition a class called "WP Object Cache" is used to hold all cached objects in a central location.

Capabilities.php – Handles the permissions used in WordPress and manages the assignment of permissions to certain users. Several classes are also contained in this file such as WP_Roles, WP_Users, and WP_Role.

Category.php – Manages the category meta-data used in posts. Blog posts are usually categorized into categories for easy navigation and searching of posts.

Classes.php – General PHP file that includes several classes commonly used in the CMS. These classes are listed as follows…

- WP – Sets up the WordPress Environment.
- WP_Error – Container to detect any errors thrown in WordPress and encapsulates it.
- Walker – Class that Displays tree like structures.
- Walker_Page – Creates an HTML list of pages.
- Walker_pagedropdown – Creates an HTML dropdown of pages.
- Walker_category – Creates an HTML list of categories.
- Walker_categorydropdown – Creates an HTML dropdown of categories.
- WP_Ajax_Response – Returns an XML response to an AJAX request.
- WP_MatchesMapRegEx – helper class to remove the need to use eval in query strings.

Plugin.php – Similar to Drupal's module.inc file, it is responsible for loading, activating, and deactivating plugins in the WordPress CMS. It is also responsible in managing the hooks for plugins as well as the action and filter calls.

Pluggable.php – A default set of functions that handle the logged in user's information. It can be extended with third-party plugins.

Formatting.php – Handles the formatting of the outputted text. It includes several functions for formatting of Unicode characters.

Query.php – Similar to Drupal's database.inc file, this file manages the database calls to the database. Unlike Drupal, It only supports MySQL.

Theme.php – Handles the theme generation and application of theme elements to the outputting data.

User.php – Manages the authentication of the user in the CMS.

Functions.php – Similar to common.inc in Drupal, this file contains a list of commonly used functions used in the WordPress CMS.

4.0 Measurements

The goal of this paper is to extract measurements from Drupal and WordPress and attempt to draw comparisons between the two content management systems. This chapter will list the measurements used in this paper and describe our methods and tools in obtaining these measurements.

4.1 Measurements Used

In traditional OO based desktop software, there is a large set of tools and test suites that are used to measure the quality of the software. Tools such as the ones proposed by Chidamber and Kemerer as well as the QMOOD model proposed by Bansiva made it possible to measure, and to some extent, quantify OO design. For example, you can capture the number of remote method calls from a given class. The number of remote method calls could indicate the complexity of a given class since a higher count would mean that the class will have more external dependencies. The higher the number, the more sensitive the class will be to these external dependencies.

While OO based tools can easily be applied to OO based programs, trying to apply these measurements to web based scripting languages is more difficult. This is due to the lack of certain OO features in these languages. This is especially true with PHP 4.0, where several noticeable OO features such as private or protected declarations, abstract classes, constructors, and destructors,   were lacking.

While this presented a challenge, the developers of both Drupal and WordPress tried to adhere to an OO based design in their respective content management systems and tried to follow through with it during implementation. Several workarounds have been made during implementation. Instead using explicit class declarations, they tried to isolate code in individual files. An example of this would be that all the caching handled by the CMS is routed through the cache.php file and not through a class called cache. This is critical since it is possible to treat these files as classes despite lacking explicit class declarations (Bob, 2005).

In addition, they also tried to enforce abstraction and isolation between the underlining databases and code. In Drupal's case, developers even attempted to apply widely used and well accepted design patterns to their design (such as the bridge, singleton, façade, and the commonly used Model View Controller patterns).

Since the developers of WordPress and Drupal attempted to design their software with OO in mind, a good portion of the measurements used in QMOOD can be used to assess the quality of their CMS product. In addition, a good portion of QMOOD that deals with the size of the application can be applied directly without any modifications.

The measurements listed below include a series size metrics that simply determine the size of a given attribute. In addition a series of measurements derived from chapter 2 will be used as well. Below are their definitions.

Lines of Code (LOC)

This is the most basic size based measurement. It simply measures the lines of code in a given class or file. This includes everything, from Server-Side Lines of code (SLOC), Comment Lines of Code, (CCLOC), and Executable Lines of Code (ELOC).

Executable Lines of Code (ELOC)

This is a basic size measurement that will measure the number of lines of executable code in a given file. Executable lines of PHP code will be measured. This measurement will not include lines of commented code.

Commented Lines of Code (CLOC)

This is a basic measurement that only counts the lines of comments in a given class or file. This paper took it a step further and also counted the number of comments inside each function or method. This will help determine if the comments are generally descriptive for the functions or methods, or if the executable code within the function or method is properly commented.

Response for Class (RFC)

This measurement captures how many methods are called when an operation within a class is invoked. The more methods needed, the higher the complexity. In web applications, a good portion of the files in the application will most likely be files instead of classes.

Lack of Cohesion in methods (LCOM2)

This metric tries to determine how closely related the methods are in a class. In other words, if a class contains multiple methods, do these methods use a common set of attributes? If a class contains methods that share a common set of attributes, then that class is considered highly cohesive. A class that contains methods that do not share the same set of attributes is considered highly incohesive. Criticism for this measurement was discussed in chapter 2 and an alternative version of LCOM called LCOM2 was used instead. The equation for LCOM2 is located under Equation 3: in chapter 2.

Number of Local Methods (NLM)

This is a count of all local methods in a class or file. It is a basic metric but does give an indication if a class or file might have too few or too many methods or functions. If the count is low, it might indicate that the methods could go into a different, more relevant class. If the count is too high, it might indicate that the methods in the class could be split off into a new class.

Number of Remote Methods (NRM)

This is a count of all remote method calls in a class or file. This metric is just like the NLM metric except that it counts the number of method calls to objects OUTSIDE the file or class.

Number of Attributes (NOA)

Similar to NLM but we count the number of attributes in each class. Obviously, if there’s a large number of attributes in the class, it might indicate a higher level of complexity. A small number of attributes might indicate that the class is nothing more than an abstract class or interface. For our web applications, since you cannot explicitly declare an attribute private, they had to use global variables within the file.

McCabe Cyclomatic Complexity Model: (CCN)

This is a classic measurement of the complexity of a given source code. This measurement captures the number of distinct paths could be traversed through a code segment. While there is a formal definition of this measurement (McCabe, 1976), a common count of this complexity metric could be obtained by counting the number of conditional statements and add 1 to the total. The equation to calculate CCN goes as follows:

Equation 4:

CCN = D + 1
Where D is the number of decision statements in a given code fragment.

Counting Weighted Methods (CWM)

This measurement is just like the NOM measurement with the exception that we add weights to the count. The weights will be derived from the McCabe Complexity metric of the given method. In other words, the weights on each method are initially set to 1. The weights are then updated to reflect the McCabe's complexity score for the given method.

Afferent Coupling (AC)

Afferent Coupling is basically a measurement of how many classes or entities that depends on a given class. A high count indicates that the class might have many responsibilities and any changes to the class might have a high impact on the entire application.

Efferent Coupling (EC)

Similar to Afferent Coupling except this metric counts the number of dependencies going out of a given class. A high EC indicates that the class or entity is highly dependent on other classes or entities, and any changes to the application would carry a higher risk in breaking the class.

Instability (I)

From the afferent and efferent measurements you can derive a new measurement called "Instability". Instability is a measurement to determine how resilient a class is to change. The formula to determine the instability of a class is very simple.

Equation 5:

$$I = EC / (AC + EC)$$

AC = the total number of afferent couplings
EC = the total number of efferent couplings

The range for I is from 0 to 1 where 0 means that the class is completely stable and 1 meaning that the class is completely instable.

4.2 Methods and Tools used

In order speed up the gathering of our measurements, several tools were used. One of these tools is PHP Depend. PHP Depend is a quality assessment tool that analyzes static PHP code, and then applies several QMOOD based measurements on the analyzed code. Several measurements captured in PHP Depend are listed below.

- LOC – Lines of Code
- ELOC – Lines of Executable Code
- CLOC – Lines of Commented Code

- NOM - Number of methods
- CCN – Cyclometric Complexity
- CCN2 – Cyclometric Complexity with Conditional statements included.

With the exception of CCN, the measurements listed in 4.1 were used. For the CCN measurement, an alternative complexity measurement was used called CCN2. Unlike CCN, CCN2 would consider the Boolean operators in a conditional statement and treat them as unique paths. In other words, the statement (x > 1 OR X < -1) would count as one path in CCN, but it would count as two paths in CCN2.

PHP Depend is more flexible with PHP projects that utilize the OO features in PHP 5. It can evaluate entire projects and generate useful charts and graphs as well as XML data for the measurements listed in 4.1. Since Drupal and WordPress were built before PHP 5 was released, PHP Depend was not able to automate these measurements. Additionally, PHP Depend was capable of making Afferent and Efferent Coupling measurements as well, but since Drupal and WordPress didn't use the "class" construct in most of their files, PHP Depend could not automate this measurement. Despite these limitations, PHP Depend saved us a lot of time on our size measurements as well as our Cyclometric Complexity (CCN2) measurements. Finally to verify the accuracy of PHP Depend, several PHP code snippets were analyzed by the tool and the results were compared against results from a manual walkthrough.

While PHP Depend saved us a lot of time in gathering most of our measurements, several measurements were more difficult to obtain, and unfortunately, tools to automate the gathering of these measurements are non-existent for the job at hand. The number of remote method calls (NRM), and the number of efferent and afferent couplings (EC and AC respectively) in a given module took more time than other measurements gathered by PHP Depend.   For the number of remote methods calls and the efferent couplings of a given module, a code walkthrough on our selected files had to be conducted. This was time consuming and possibly less accurate due to possible human error. As for the Afferent Coupling measurement, Adobe Dreamweaver was used to find and list all calls to the given method. Dreamweaver is a common web editor for websites and it has the ability to search through the entire project for a specific method and return a list of results (Adobe, 2012).

5.0 Measurement Results

This chapter will cover the measurement results that were gathered for this paper. The first part of this chapter will cover the measurements regarding the size of the selected modules such as the number of lines of code, the number of methods, and several other related metrics. The second part of this chapter will cover measurements regarding the complexity of the modules such as the dependencies between the modules, and the complexity of our measured code.

5.1 Size Measurements:

The tables in this section list results that deal primarily with the size of the modules that are being measured. These measurements are as follows.

- The lines of code (LOC),
- The number of lines of commented code (CLOC)
- The lines of executable code (ELOC)

- The number of methods or functions in a given file (NLM).

These size measurements are useful in determining other measurements such as the number of defects per line of code. Additionally this helps to normalize other measurements. An example of this is if someone looks at a given method in a class and noticed that the cyclometric complexity count in the method is very high compared to the other methods in the class. They might wrongly assume that the method is unnecessarily complex, while in truth, the method has a higher LOC value compared to the other methods in the class. This is one reason why it is important to gather these measurements regarding the size of the classes and files.

Table 1: Our results for the size measurements from the WordPress content management system. The results show the lines of code (LOC), the commented lines of code (CLOC), the executable lines of code (ELOC), the number of local methods (NLM), the number of attributes (NOA), and the number of global and constant variables counted in the selected files. The totals for each metric are listed at the bottom of this table. Global variables are defined by the PHP "global" declaration while constants are defined by the "define" declaration.

File	LOC	CLOC	ELOC	NLM	NOA	Globals	constants
index.php	18	13	2	0	0	0	0
wp-blog-header.php	20	5	5	0	0	0	0
wp-load.php	54	22	17	0	0	0	1
wp-config.php	76	48	15	0	0	0	12
atomlib.php	288	11	219	13	18	0	0
author-template.php	343	148	153	12	0	4	0
bookmark-template.php	255	117	109	2	0	0	0
bookmark.php	377	116	218	7	0	2	0
cache.php	167	113	38	10	0	0	0
Cache.php - WP Object Cache	268	154	84	10	4	0	0
canonical.php	346	83	222	2	0	4	0
capabilities.php - WP_Roles	216	124	74	9	5	2	0
capabilities.php - WP_Role	103	64	29	4	2	1	0
capabilities.php - WP_User	384	226	124	13	11	2	0
capabilities.php	322	98	192	6	0	1	0
category-template.php	977	394	474	14	0	3	0
category.php	368	196	125	14	0	0	0
classes.php - WP	492	185	246	10	9	10	0
classes - WP_Error	172	95	59	8	2	0	0
classes.php - Walker	383	147	183	9	3	0	0
classes.php - Walker_Page	97	43	43	4	2	0	0
classes.php - walker_pagedropdown	97	43	43	1	2	0	0
classes.php - walker_category	139	45	74	4	2	0	0
classes.php - walker_categoryDropDown	43	20	20	1	2	0	0
classes.php - WP_Ajax_Response	129	47	70	3	1	0	0
classes.php - WP_MatchesMapRegEx	84	54	23	4	4	0	0
plugin.php	705	420	221	20	0	4	0
pluggable.php	1792	626	933	44	0	9	0
formatting.php	2850	1153	1481	73	0	7	0
query.php	751	376	264	37	0	12	0
theme.php	1386	579	636	54	0	18	0
user.php	737	286	351	17	0	10	0
functions.php	3641	1455	1810	115	0	16	0
compat.php	163	16	117	12	0	1	0
TOTAL:	**18075**	**7434**	**8635**	**532**	**67**	**106**	**13**

Table 2: Our results for the size measurements from the Drupal content management system. The results show the lines of code (LOC), the commented lines of code (CLOC), the executable lines of code (ELOC), the number of local methods (NLM), the number of attributes (NOA), and the number of global and constant variables counted in the selected files. The totals for each metric are listed at the bottom of this table. Global variables are defined by the PHP "global" declaration while constants are defined by the "define" declaration.

File	LOC	CLOC	ELOC	NLM	NOA	Globals	constants
index.php	39	10	21	0	0	0	0
Bootstrap.inc	1028	462	450	31	0	12	18
Common.inc	2380	1120	1073	74	0	6	3
cache.inc	169	90	69	3	0	1	0
database.inc	320	148	149	7	0	6	1
database.mysqli.inc	430	197	192	20	0	2	0
database.mysql.inc	448	206	198	20	0	2	0
file.inc	1105	281	739	19	0	1	7
database.pgsql.inc	438	206	187	21	0	3	0
install.pgsql.inc	135	24	92	2	0	0	0
xmlrpcs.inc	319	77	226	11	0	0	0
xmlrpc.inc	488	115	351	21	0	0	0
unicode.inc	509	182	288	17	0	1	3
theme.inc	1125	403	590	35	0	5	3
tablesort.inc	195	88	94	7	0	0	0
session.inc	174	74	75	10	0	1	0
path.inc	208	101	89	8	0	0	0
pager.inc	423	218	165	10	0	3	0
module.inc	423	191	197	14	0	0	0
menu.inc	1401	492	747	37	0	3	0
locale.inc	1662	302	1264	40	0	1	0
install.mysqli.inc	137	23	95	2	0	0	0
install.mysql.inc	142	25	97	2	0	0	0
install.inc	745	218	457	22	0	8	14
image.inc	347	101	204	15	0	0	0
form.inc	1620	592	881	55	0	4	0
TOTAL:	**16410**	**5946**	**8990**	**503**	**0**	**59**	**49**

Table 3: Size measurements for the WordPress CMS broken down to percentages. Averages for the %CLOC, %ELOC, and the average LOC/NLM are weighted averages based off the lines of code (LOC)

Filename	LOC	CLOC	% CLOC	ELOC	% ELOC	NLM	avg LOC/NLM
index.php	18	13	72.22%	2	11.11%	0	NA
wp-blog-header.php	20	5	25.00%	5	25.00%	0	NA
wp-load.php	54	22	40.74%	17	31.48%	0	NA
wp-config.php	76	48	63.16%	15	19.74%	0	NA
atomlib.php	288	11	3.82%	219	76.04%	13	15.23
author-template.php	343	148	43.15%	153	44.61%	12	12.58
bookmark-template.php	255	117	45.88%	109	42.75%	2	53.50
bookmark.php	377	116	30.77%	218	57.82%	7	30.85
cache.php	167	113	67.66%	38	22.75%	10	3.60
Cache.php - WP Object Cache	268	154	57.46%	84	31.34%	10	7.80
canonical.php	346	83	23.99%	222	64.16%	2	109.50
capabilities.php - WP_Roles	216	124	57.41%	74	34.26%	9	7.33
capabilities.php - WP_Role	103	64	62.14%	29	28.16%	4	6.25
capabilities.php - WP_User	384	226	58.85%	124	32.29%	13	8.60
capabilities.php	322	98	30.43%	192	59.63%	6	31.66
category-template.php	977	394	40.33%	474	48.52%	14	20.57
category.php	368	196	53.26%	125	33.97%	14	8.78
classes.php - WP	492	185	37.60%	246	50.00%	10	23.50
classes - WP_Error	172	95	55.23%	59	34.30%	8	6.87
classes.php - Walker	383	147	38.38%	183	47.78%	9	19.77
classes.php - Walker_Page	97	43	44.33%	43	44.33%	4	9.75
classes.php - walker_pagedropdown	97	43	44.33%	43	44.33%	1	10.00
classes.php - walker_category	139	45	32.37%	74	53.24%	4	17.50
classes.php - walker_categoryDropDown	43	20	46.51%	20	46.51%	1	16.00
classes.php - WP_Ajax_Response	129	47	36.43%	70	54.26%	3	22.33
classes.php - WP_MatchesMapRegEx	84	54	64.29%	23	27.38%	4	4.25
plugin.php	705	420	59.57%	221	31.35%	20	10.95
pluggable.php	1792	626	34.93%	933	52.06%	44	19.20
formatting.php	2850	1153	40.46%	1481	51.96%	73	20.26
query.php	751	376	50.07%	264	35.15%	37	54.23
theme.php	1386	579	41.77%	636	45.89%	54	11.72
user.php	737	286	38.81%	351	47.63%	17	20.47
functions.php	3641	1455	39.96%	1810	49.71%	115	15.73
compat.php	163	16	9.82%	117	71.78%	12	9.75
Averages:	**536.6**	**221.2**	**41.23%**	**255.1**	**47.55%**	**15.6**	**20.45**

Table 4: Size measurements for the Drupal CMS broken down to percentages. Averages for the %CLOC, %ELOC, and the average LOC/NLM are weighted averages based off the lines of code (LOC)

Filename	LOC	CLOC	% CLOC	ELOC	% ELOC	NLM	avg LOC/NLM
index.php	39	10	25.64%	21	53.85%	0	NA
Bootstrap.inc	1028	462	44.94%	450	43.77%	31	13.90
Common.inc	2380	1120	47.06%	1073	45.08%	74	14.44
cache.inc	169	90	53.25%	69	40.83%	3	22.66
database.inc	320	148	46.25%	149	46.56%	7	21.00
database.mysqli.inc	430	197	45.81%	192	44.65%	20	9.55
database.mysql.inc	448	206	45.98%	198	44.20%	20	9.85
file.inc	1105	281	25.43%	739	66.88%	19	38.47
database.pgsql.inc	438	206	47.03%	187	42.69%	21	8.90
install.pgsql.inc	135	24	17.78%	92	68.15%	2	46.00
xmlrpcs.inc	319	77	24.14%	226	70.85%	11	23.18
xmlrpc.inc	488	115	23.57%	351	71.93%	21	16.66
unicode.inc	509	182	35.76%	288	56.58%	17	16.70
theme.inc	1125	403	35.82%	590	52.44%	35	16.71
tablesort.inc	195	88	45.13%	94	48.21%	7	13.28
session.inc	174	74	42.53%	75	43.10%	10	7.40
path.inc	208	101	48.56%	89	42.79%	8	11.00
pager.inc	423	218	51.54%	165	39.01%	10	16.40
module.inc	423	191	45.15%	197	46.57%	14	14.00
menu.inc	1401	492	35.12%	747	53.32%	37	19.54
locale.inc	1662	302	18.17%	1264	76.05%	40	31.57
install.mysqli.inc	137	23	16.79%	95	69.34%	2	46.50
install.mysql.inc	142	25	17.61%	97	68.31%	2	47.50
install.inc	745	218	29.26%	457	61.34%	22	20.09
image.inc	347	101	29.11%	204	58.79%	15	13.53
form.inc	1620	592	36.54%	881	54.38%	55	16.29
Averages:	**631.15**	**228.692**	**36.23%**	**345.8**	**54.78%**	**19.35**	**19.62**

The files selected in the tables above are some of the core files for Drupal and WordPress. They are responsible for a wide range of responsibilities such as authentication, caching, database operations, and display of content to the user.

There are several things to note in the tables listed above. In table 2 you will notice that there are no attributes defined in Drupal. This is due to the Drupal team deciding to omit the use of classes in their core PHP files. Secondly, constants for both WordPress and Drupal are defined by using the PHP “define” statement. Thirdly, more lines of WordPress code have been analyzed than Drupal code. This is due in part that several PHP files in WordPress contain several classes in each file while a file in Drupal constitute as one class. Additionally the column called “globals” is the number of global variables defined by the PHP “global” statement.

For the commented lines of code (CLOC) Drupal has an average near 36% while WordPress has an average near 41%. From this observation it appears that both applications have a good amount of CLOC in their classes.

The next item is the percentage of executable lines of code (ELOC). Close to 55% of code in the files measured in Drupal are executable lines of code versus roughly 47% in WordPress. One thing that you should keep in mind is that the higher the percentage of ELOC is the lower the percentage of CLOC.

Our third measurement is the average number of ELOC for each method in a class. Similar to CLOC, there is no optimal number of LOC for a given function. Too much LOC in a given function indicates that the function might be too complex and should be broken up while too little LOC indicates that the function might be too trivial.

In determining the average number of ELOC for a given function in a class the following equation is used.

Equation 6:

$$1/n\sum_{i=1}^{n} ELOCi$$

Where *n* is the number of functions in a given class and *ELOCi* is the number of lines of executable code in a given function.

Drupal had an average of 19.62 lines of executable code per function. WordPress got an average of 20.45 lines of executable code per function.

5.2 Complexity Measurements:

The next set of measurements primarily deals with the complexity of the classes and files that were analyzed. The following metrics were used in our complexity measurements.

- The number of remote methods called (NRM)
- The response for a class (RFC)
- The lack of cohesion in methods (LCOM2)
- The number of afferent couplings (AC)
- The number of efferent couplings (EC)
- The cyclometric complexity (CCN2)
- Instability (I)

Table 5 lists our complexity metrics for the WordPress Content Management System while the Table 6 lists our complexity metrics for the Drupal Content

Management System. Finally table 7 and table 8 lists the averages in complexity for WordPress and Drupal respectively.

Table 5: Complexity measurements for WordPress listing the Response For Class (RFC), Lack of Cohesion Of Methods (LCOM2), Number of Remote Methods (NRM), Number of Local Methods (NLM), Afferent Coupling (AC), and Efferent Coupling (EC). The last row contains the totals for the measurements across the files analyzed.

File	RFC	NRM	NLM	CCN	AC	EC
index.php	0	0	0	1	0	1
wp-blog-header.php	0	0	0	2	0	2
wp-load.php	0	0	0	6	0	4
wp-config.php	0	0	0	2	0	1
atomlib.php	15	2	13	61	1	2
author-template.php	25	13	12	51	13	8
bookmark-template.php	10	8	2	28	4	6
bookmark.php	23	16	7	64	8	6
cache.php	17	7	10	10	20	1
Cache.php - WP Object Cache	11	1	10	26	1	1
canonical.php	41	39	2	144	0	9
capabilities.php - WP_Roles	12	3	9	19	3	2
capabilities.php - WP_Role	8	4	4	7	3	2
capabilities.php - WP_User	20	7	13	33	11	4
capabilities.php	17	11	6	58	88	6
category-template.php	35	21	14	88	23	13
category.php	26	12	14	39	22	6
classes.php - WP	36	26	10	95	6	9
classes - WP_Error	10	0	8	21	37	0
classes.php - Walker	9	0	9	67	2	0
classes.php - Walker_Page	10	6	4	14	2	4
classes.php - walker_pagedropdown	2	1	1	2	0	1
classes.php - walker_category	9	5	4	28	0	4
classes.php - walker_categoryDropDown	2	1	1	4	0	1
classes.php - WP_Ajax_Response	8	5	3	17	1	3
classes.php - WP_MatchesMapRegEx	8	4	4	5	0	0
plugin.php	23	3	20	68	153	2
pluggable.php	112	68	44	244	97	20
formatting.php	86	13	73	296	155	7
query.php	53	16	37	82	4	6
theme.php	80	26	54	190	18	13
user.php	59	42	17	115	32	12
functions.php	177	62	115	507	156	18
compat.php	18	6	12	36	16	3
Totals:	**962**	**428**	**532**	**2419**	**876**	**169**

Table 6: Complexity measurements for Drupal listing the Response For Class (RFC), Lack of Cohesion Of Methods (LCOM2), Number of Remote Methods (NRM), Number of Local Methods (NLM), Afferent Coupling (AC), and Efferent Coupling (EC). The last row contains the totals for the measurements across the files analyzed.

File	RFC	NLM	NRM	CCN	AC	EC
index.php	0	0	0	6	0	1
Bootstrap.inc	50	31	19	130	83	10
Common.inc	105	74	31	334	146	14
cache.inc	10	3	7	18	4	3
database.inc	16	7	9	43	40	6
database.mysqli.inc	33	20	13	42	37	5
database.mysql.inc	33	20	13	45	37	6
file.inc	32	19	13	119	7	4
database.pgsql.inc	36	21	15	47	37	6
install.pgsql.inc	4	2	2	13	0	2
xmlrpcs.inc	18	11	7	53	1	2
xmlrpc.inc	24	21	3	83	1	2
unicode.inc	19	17	2	83	15	2
theme.inc	73	35	38	156	49	12
tablesort.inc	13	7	6	35	9	4
session.inc	17	10	7	27	1	2
path.inc	14	8	6	29	42	4
pager.inc	17	10	7	48	17	3
module.inc	24	14	10	60	32	6
menu.inc	57	37	20	245	12	8
locale.inc	66	40	26	235	0	12
install.mysqli.inc	4	2	2	14	0	2
install.mysql.inc	4	2	2	14	0	2
install.inc	54	22	32	144	53	12
image.inc	20	15	5	47	2	4
form.inc	116	55	61	308	37	6
Totals	**859**	**503**	**356**	**2378**	**662**	**140**

Table 7: Averages for the Number of Remote Methods (NRM), Afferent Coupling (AC), Efferent Coupling (EC), and Executable Lines of Code (ELOC) across files in WordPress. The weighted average for Instability (I) is based on the number of executable code (ELOC), while the weighted average for the Lack of Cohesion of Methods (LCOM2) is based off the Number of Local Methods (NLM)

File	NRM	AC	EC	CCN2	Instability	LCOM2	ELOC
index.php	0	0	1	1	NA	NA	2
wp-blog-header.php	0	0	2	2	NA	NA	5
wp-load.php	0	0	4	6	NA	NA	17
wp-config.php	0	0	1	2	NA	NA	15
atomlib.php	2	1	2	61	0.67	1.00	219
author-template.php	13	13	8	51	0.38	0.85	153
bookmark-template.php	8	4	6	28	0.60	1.00	109
bookmark.php	16	8	6	64	0.43	0.79	218
cache.php	7	20	1	10	0.05	1.00	38
Cache.php - WP Object Cache	1	1	1	26	0.50	0.75	84
canonical.php	39	0	9	144	1.00	0.38	222
capabilities.php - WP_Roles	3	3	2	19	0.40	0.59	74
capabilities.php - WP_Role	4	3	2	7	0.40	0.17	29
capabilities.php - WP_User	7	11	4	33	0.27	0.78	124
capabilities.php	11	88	6	58	0.06	0.50	192
category-template.php	21	23	13	88	0.36	0.76	474
category.php	12	22	6	39	0.21	1.00	125
classes.php - WP	26	6	9	95	0.60	0.86	246
classes - WP_Error	0	37	0	21	0.00	NA	59
classes.php - Walker	0	2	0	67	0.00	0.78	183
classes.php - Walker_Page	6	2	4	14	0.67	1.00	43
classes.php - walker_pagedropdown	1	0	1	2	1.00	1.00	43
classes.php - walker_category	5	0	4	28	1.00	1.00	74
classes.php - walker_categoryDropDown	1	0	1	4	1.00	1.00	20
classes.php - WP_Ajax_Response	5	1	3	17	0.75	0.33	70
classes.php - WP_MatchesMapRegEx	4	0	0	5	0.00	0.63	23
plugin.php	3	153	2	68	0.01	0.73	221
pluggable.php	68	97	20	244	0.17	0.96	933
formatting.php	13	155	7	296	0.04	0.98	1481
query.php	16	4	6	82	0.60	0.90	264
theme.php	26	18	13	190	0.42	0.98	636
user.php	42	32	12	115	0.27	0.88	351
functions.php	62	156	18	507	0.10	0.97	1810
compat.php	6	16	3	36	0.16	0.83	117
Averages	**12.59**	**25.76**	**5.21**	**71.47**	**0.26**	**0.89**	**255.12**

Table 8: Averages for the Number of Remote Methods (NRM), Afferent Coupling (AC), Efferent Coupling (EC), and Executable Lines of Code (ELOC) across files in Drupal. The weighted average for Instability (I) is based on the number of executable code (ELOC), while the weighted average for the Lack of Cohesion of Methods (LCOM2) is based off the Number of Local Methods (NLM)

File	NRM	AC	EC	CCN2	Instability	LCOM2	ELOC
index.php	0	0	1	6	NA	NA	21
Bootstrap.inc	19	83	10	130	0.11	0.81	450
Common.inc	31	146	14	334	0.09	0.86	1073
cache.inc	7	4	3	18	0.43	0.33	69
database.inc	9	40	6	43	0.13	0.98	149
database.mysqli.inc	13	37	5	42	0.12	0.83	192
database.mysql.inc	13	37	6	45	0.14	0.83	198
file.inc	13	7	4	119	0.36	0.95	739
database.pgsql.inc	15	37	6	47	0.14	0.92	187
install.pgsql.inc	2	0	2	13	1.00	1.00	92
xmlrpcs.inc	7	1	2	53	0.67	1.00	226
xmlrpc.inc	3	1	2	83	0.67	1.00	351
unicode.inc	2	15	2	83	0.12	0.71	288
theme.inc	38	49	12	156	0.20	0.93	590
tablesort.inc	6	9	4	35	0.31	1.00	94
session.inc	7	1	2	27	0.67	0.80	75
path.inc	6	42	4	29	0.09	1.00	89
pager.inc	7	17	3	48	0.15	0.60	165
module.inc	10	32	6	60	0.16	1.00	197
menu.inc	20	12	8	245	0.40	0.90	747
locale.inc	26	0	12	235	1.00	0.98	1264
install.mysqli.inc	2	0	2	14	1.00	1.00	95
install.mysql.inc	2	0	2	14	1.00	1.00	97
install.inc	32	53	12	144	0.18	0.94	457
image.inc	5	2	4	47	0.67	1.00	204
form.inc	61	37	6	308	0.14	0.95	881
Averages:	**13.69**	**25.46**	**5.38**	**91.46**	**0.38**	**0.91**	**345.77**

From tables 7 and 8, the average number of remote method calls (NRM) between Drupal and WordPress are very close having 13.69 calls on average for Drupal and 12.59 calls on average for WordPress. Using the number of remote method called from a given class and the number of local methods in a given class, you can determine the "Response for Class" of the class. Drupal has an average of 33.04 RFC calls for a given class and WordPress has an average of 28.29 RFC calls for a given class.

For our Cyclometric Complexity Measurement (CCN2), Drupal has an average CCN2 of 91.46 for a given class while WordPress has an average CCN2 of 71.47 for a given class. We obtained these results by using PHP Depend to obtain a CCN2 value for the class and then obtain an average from all the classes measured. From these results, it seems that Drupal, on average, has a higher cyclometric complexity than WordPress.

For our afferent (AC) and efferent calls (EC), Drupal has an average of 25.46 afferent calls per class and an average of 5.38 efferent calls per class. WordPress has an average of 25.6 afferent calls per class and 5.21 efferent calls per class. As you can see, the number of afferent and efferent calls in Drupal and WordPress are very close.

For Instability (I), Drupal has .38 for the average instability for each class while WordPress has .21 for the average instability for each class. Please note that these results reflect the files and classes that are listed in the tables above.

For our Lack of cohesion measurement (LCOM2) Drupal has a weighted average of .91 while WordPress has a weighted average of .89. The higher the measurement, the less cohesive the methods in the file or class are.

Finally, Looking at the tables above you will notice that for some values for LCOM2 or Instability, there is an “NA” listed. This means that we could not determine the value for the selected file or class. Also, an LCOM2 measurement with a “NA” indicates that the file does not have any classes or methods defined. In other words, the PHP file is not designed to be a class or something resembling a class. While this is problematic it is not uncommon for web applications to have files with no methods defined since a good portion of the pages are usually nothing more than a script with static content.

6.0 Analysis

The previous chapter listed several tables containing our measured results for WordPress and Drupal. This chapter will look into these results more closely and try to draw observations from our measurements as well as making comparisons between Drupal and WordPress with our results.

6.1 Size Metrics:

Chapter 5 presented several size measurements for WordPress and Drupal. One measurement listed was the percentage of commented lines of code (CLOC). Drupal had an average CLOC of 36% across all files while WordPress had an average CLOC of 41%. There are no definite guidelines to the ratio of commented lines of code versus total lines of code, but it is ideal to have comments describing each class or method as well as in non-trivial areas of code. It appears that both WordPress and Drupal has sufficient amount of commenting in the code. This is useful since both products are open-source which allows them to be modified for specific purposes. Modifying an application with minimal commenting will pose a challenge since it typically takes longer to understand the code being used.

Looking at the executable lines of code (ELOC), Drupal had an average ELOC of 55% compared to 47% in WordPress. There is no definite guide on what percentage of code has to be executable. However, notice that totaling both the CLOC and the ELOC in Drupal or WordPress does not yield a result of 100%. The missing percentage could be explained by empty lines of code in the files or lines of non-PHP code. Such code could be HTML, Javascript, or plain text.

One area of importance is the number of lines of code (LOC) in a given method. A method with a high LOC value might indicate an overly complex method which might benefit a refactor. Drupal had on average 19.62 lines of code compared to 20.49 lines of code for WordPress. These measurements are so close together that they could be considered equal. However, looking at the results in table 2 you will see that the canonical.php file in WordPress has an average of 109.5 LOC per method. This is considerably higher than Drupal's largest value of 47.50 LOC per method which is located in the install.mysql.inc file. Additionally, we can look at the distribution of the average LOC per method for both WordPress and Drupal. Looking at figure 3 you can see that a majority of the methods in WordPress has a LOC lower than 40. Looking at figure 4 all the methods for Drupal has a LOC value between 0 and 50. This is good since you rarely will encounter a method with a large LOC value. Methods with a smaller LOC value are easier to understand and maintain.

Figure 3: Histogram showing the average lines of code per method in WordPress. The bars represent the number of methods that have a measured LOC within the given range.

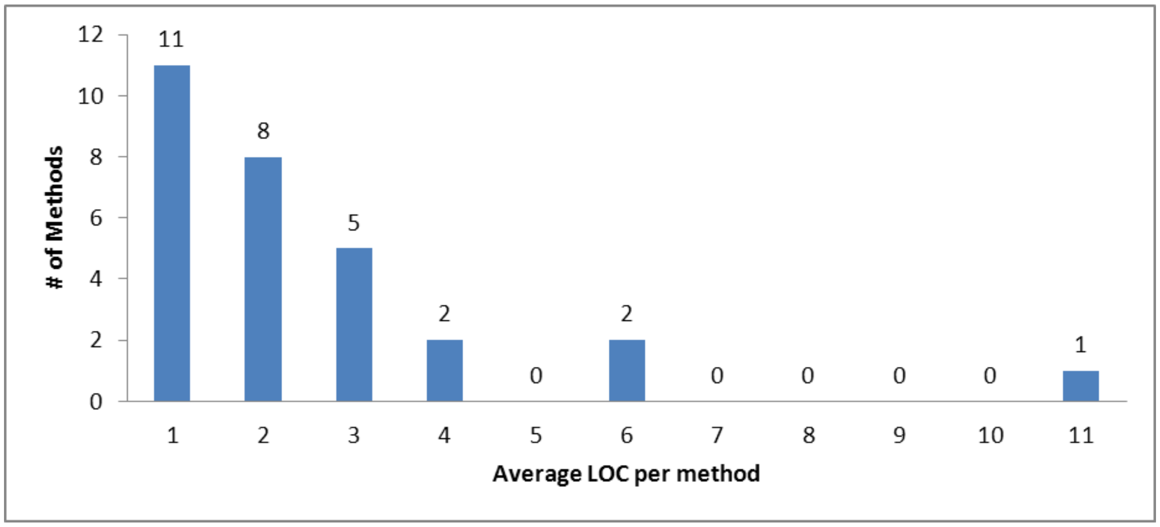
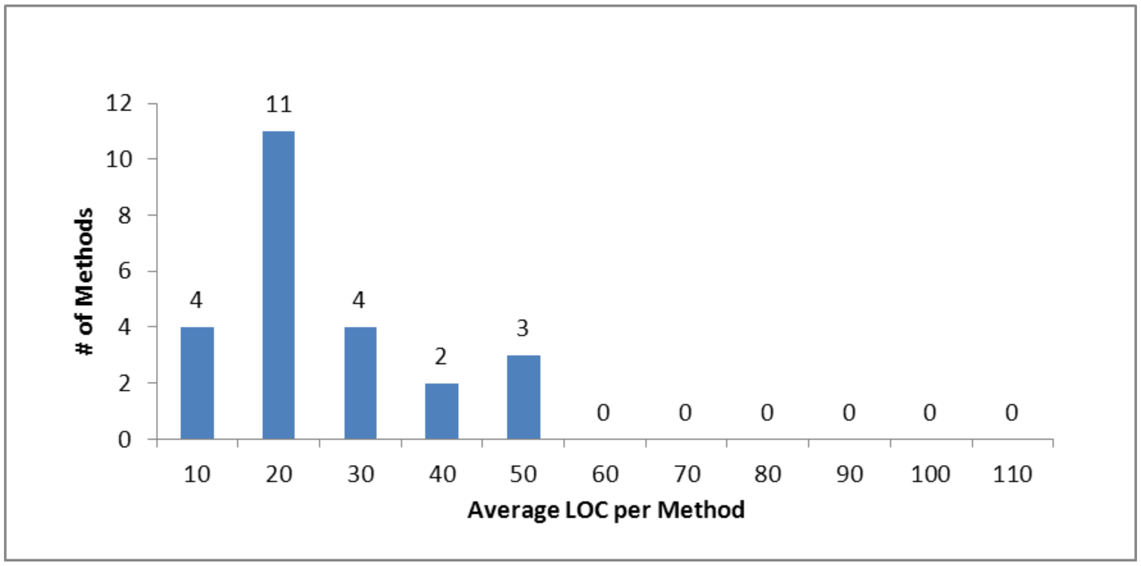


Figure 4: Histogram showing the average lines of code per method in Drupal. The bars represent the number of methods that have a measured LOC within the given range.



6.2 Complexity Metrics:

There is no good way in determining what is nominal for the Number of Remote Methods (NRM) and Response for Class (RFC) measurements. Generally, the higher the measurement for both NRM and RFC, the more complex the class is. This is due to the higher number of dependencies the class has with other classes within the application. At first glance, it appears that WordPress has a lower value for the NRM and RFC measurements and thus an average class in WordPress is considered “less complex” than a class in Drupal. While this might be the case, the measurements will need to be analyzed in more detail.

In obtaining the NRM measurement, a code walkthrough had to be performed on the selected class. The total remote methods calls were then listed for each local method within the class. For the results in the previous paragraph, all the remote method calls were simply added in a given class and averaged out across all selected classes. The problem with this approach is that some classes will contain more local methods than other classes hence a higher NRM count. A better approach is to get the average NRM across all the local methods within the class and average out the averages. This resulted in .77 remote method calls per local method in Drupal versus 1.48 remote method calls per local method in WordPress. This contradicts the earlier results of 13.69 and 12.59 respectively and indicates that WordPress has more external dependencies on a given local method compared to Drupal hence its complexity will be higher than Drupal.

As for the RFC measurement, a similar fallacy applies since the RFC measurement is based on a worst case scenario where a given method call in a class could possible invoke all local methods within the class and all remote methods called by the class. Simply put…

Equation 7:

$$RFC = NRM + NLM$$

The fallacy in this logic is that it is unlikely that there will be a method call that will invoke all the local and remote methods within the class. This creates an illusion that a class with more local methods will be more complex than a class with fewer local methods. There is a more accurate way to determine the RFC of a class by tracing through the method call stacks within a class and using the maximum result. The problem with this approach is that it is a time consuming method and will be difficult to apply in real world situations. On the other hand most sources simply determine the RFC by looking at the absolute worst-case scenario for the RFC which is simply applying a ceiling to the possible number of local and remote method calls. This is the same method that was used and while it may not be as accurate, it helps to get a general idea on which classes might be more complex than others.

The next measurement will be the “Cyclometric Complexity Model” (CCN2) measurement. This measurement measures the number of unique paths a given code segment could follow. This is important for testing purposes as well as to determine if a particular code segment might be too complex and should be refined.

While these results are helpful, they don’t take into account the size of the classes being measured. In other words, a larger class will most likely have a larger CCN2 value than a smaller class due to its size. In order to obtain a more accurate result, the number of lines of executable code (ELOC) should be factored into this measurement. This is possible by using the following formula…

Equation 8:

$$1/n\sum_{i=1}^{n} CCN2i/ELOCi$$

n = the total number of classes analyzed,

CCN2i = our Cyclometric Complexity result in the selected class and

ELOCi = the number of lines of executable code in the selected class.

Results from using this formula will range from zero and one. A value of zero means that there are no unique paths through the code. This is impossible since by default, there is at least one path through a selected code segment. On the other hand, a result of one means that every line of executable code is a unique path. This is considered a worst-case scenario.

This method resulted in a CCN2 of .26 per line of executable code for Drupal versus .30 per line of executable code for WordPress. This contradicts our earlier results and it appears that WordPress is more complex than Drupal.

The next set of measurements will be the Efferent and Afferent Couplings (EC and AC respectively). Unlike CCN2 which measures the complexity within a class, the EC and AC metrics measures the complexity of the class in terms of its dependencies and dependents of the class. In short, the measurement looks at how complex the relationships are from the selected class to other classes within the application. Unlike previous measurements, these measurements will be taken on the class level and not at the method level. This is because the goal of these measurements is to determine how susceptible our selected classes are when other classes in the application are altered.

Something to be noted here is that for these measurements, the average number of afferent couplings is typically higher than the number of efferent couplings. This is because any class outside our selection of files in Drupal or WordPress can call our selected class. Since there are more classes outside the file sampling for both Drupal and WordPress, the number of AC calls will be higher than EC calls.

As mentioned in chapter 2, Instability (I) is a measurement to determine how resilient a class is to change. For the weighted average in each class, Drupal got .38 while WordPress got .26 for instability. So, a question regarding this metric is what should be considered a nominal value? At first glance, one would consider that an average result closer to zero would be ideal since most classes will be considered more "stable". If that is the case, WordPress will be considered more stable and thus changes to WordPress would be easier to implement than Drupal. The problem with this reasoning is that for every stable class that exists, there has to be a class that depends on it therefore not all classes will be considered stable (Martin, 1994). We can have completely stable classes, with no dependents, but the class will become useless since nothing will be using the class.

A more realistic situation is that you will have a set of classes that will be considered stable and a set of classes that are considered instable. In an essence, the goal is to make our instability measurement for a selected class approach either zero or one, and away from the middle. Since we got an average across all measured classes of .38 and .26 for

Drupal and WordPress respectively, it seems that they might be trying to do that. But, we can go a step further and generate histograms from our instability measurements.

Figure 5: Histogram showing the distribution of classes in WordPress with a specific Instability. Instability values closer to 1 indicates an unstable class while values closer to 0 indicates a stable class. Notice the spikes at .1 and 1 while a majority of the measurements fall between .2 and .8.

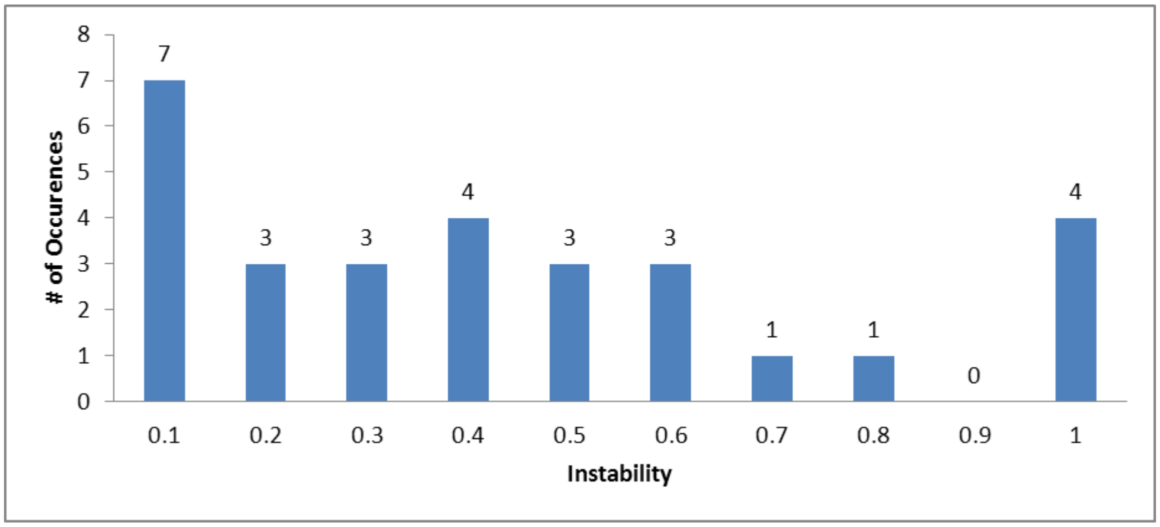
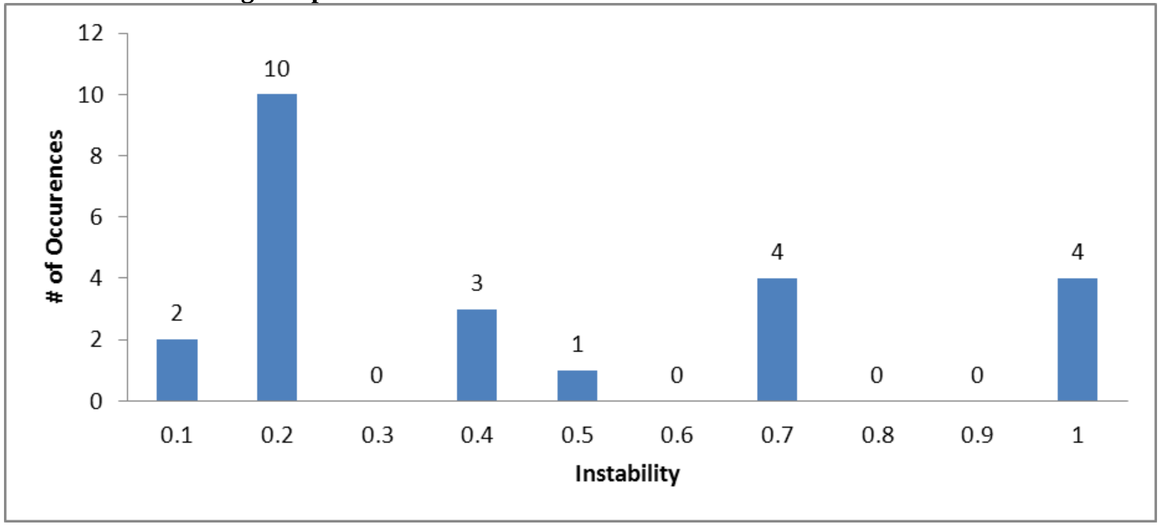


Figure 6: Histogram showing the distribution of classes in Drupal with a specific Instability. Instability values closer to 1 indicates an unstable class while values closer to 0 indicates a stable class. Note that there are several classes in Drupal that have Instability value higher than .7 but there are also classes with a very low instability value. This distribution indicates that some of these non-stable classes might depend on the more stable classes.



Ideally the histograms should show a double curve or a distinctive “V” shape instead of a single bell curve. Looking at the graph for Drupal you can see that there might be a weak double curve forming with .2 and .7 being the peaks. It is very difficult to tell, and in order to get a more accurate answer, more classes will need to be included in the measurements. As for WordPress, you can see spikes at .1 and 1, with a slight curve peaking at .4.

The values of 1 in WordPress and Drupal are most likely due to files being nothing more than installers. These files have no dependencies but will depend on other classes or files. With that being factored in, it appears that WordPress might not have a much defined set of stable and instable classes within the CMS. In other words, it seems that Drupal might have taken some effort to separate their stable, core classes away from their more instable, dependent classes. In Drupal, a few of the files in the .1 to .2 range are database.inc (.13), common.inc (.09), and unicode.php (.12). These files are considered critical to the CMS since database.inc handles database operations, common.inc holds a common set of methods used in Drupal, and unicode.php handles the encoding of information in the CMS. Less stable files in Drupal are xmlrpc.inc (.67), and session.inc (.67). These files typically handle syndication and session management which in turn might depend on database and caching. Finally, a surprising finding is that cache.inc in Drupal has an instability measurement of .43. This is considered high for a class that has the potential to have numerous dependencies. An explanation could be that cahce.inc might depend on database, or even session management but more information is needed.

For WordPress, it appears that a majority of the classes have close to the same number of efferent couplings as afferent couplings and thus an instability measurement closer to .5. Files with lower instability measurements are cache.php (.05), capabilities.php (.06) and plugin.php (.01). cache.php in WordPress appears considerably more stable than cache.inc in Drupal, but there is a class called WP_Object_Cache in WordPress which has an instability measurement of .5. This is close to the same measurement as the cache.inc file used in Drupal. Capabilities.php handles authentication and role assignment while plugin.php handles plugin management. These classes are considered critical to WordPress with numerous dependents. With the exception of files with a measurement value of 1, the following files or classes have a higher instability measurement. Atomlib.php (.67), classes.php (.6), WP_Ajax_response (.75). AtomLib.php handles syndication, classes.php contains several utility methods, and WP_Ajax_Response, handles AJAX calls. These files would most likely need access to underlying database, file, or caching functionality therefore they are highly dependent on other files or classes in the CMS.

Another common OO measurement that exists is the “abstractness” of a class (A). It is simply the number of abstract declared classes over the total number of classes within the file selection. This measurement was not included in this paper since PHP 4 did not support the abstract keyword. PHP 5, on the other hand, does support the “abstract” keyword, so this measurement will work for PHP 5 based web applications, but since Drupal and WordPress was built around the time of PHP4, it was get used.

The final measurement is the Lack of Cohesion of Methods (LCOM2) measurement. WordPress has a weighted average of .89 while Drupal had a weighted average of .91 for the LCOM. The weighted average takes into consideration the number of local methods that each class has. Looking at the values, one could argue that since WordPress did explicitly declare classes in some of their files compared to Drupal that they may have took some effort in ensuring that the methods in their classes are closely related. The problem is that developers for Drupal also took into consideration isolating code into files and making sure that closely related methods are located in the appropriate files. Finally, a measurement of .89 and .91 for WordPress and Drupal is considered too close to draw comparisons since the measurements were taken in less than ideal circumstances. In particular, the lack of explicit class and attribute declarations in Drupal.

7.0 Conclusion:

The resulting measurements from Drupal and WordPress are interesting since many of the results in their measurements are close. So the question is, what content management system is better?   From our analysis in chapter 6, it appears that Drupal has a slight edge regarding keeping code size to a minimum in its methods. In Figure 3: Histogram showing the average lines of code per method in WordPress. The bars represent the number of methods that have a measured LOC within the given range. you can see a few outliers for WordPress compared to Figure 4 in Drupal. Additionally for the Instability, there is an argument that Drupal took some effort in stabilizing classes or files that are critical to the CMS such as the database class. Figure 5 shows this separation to a degree compared to WordPress where a considerable number of files and classes fall towards the middle in Figure 6.

One item that should be pointed out was that WordPress did on several occasion use explicit class declarations compared to Drupal which did not. An argument could be made that an application that uses explicit object oriented statements would result in a higher quality product. The measurement results in this paper say otherwise. While WordPress did not use class declarations throughout the CMS, the measurements between Drupal and WordPress were pretty comparable. Looking at the averages in Table 3 and Table 4 as well as the average complexity measurements in Table 7 and Table 8 you can see that a lot of averages between Drupal and WordPress were close. There are differences in the measurements, but nothing that really stood out as alarming.

One reason for this is that the Drupal development team took a considerable amount of time on planning and designing the CMS. They took the lack of OO features into consideration and omitted explicit OO calls (Bob, 2005). WordPress did use explicit class definitions, but they did not use it throughout. This does not mean that they took a more ad-hoc approach in their design of the CMS. What it means is that if WordPress did use class declarations throughout the CMS, the measurement results could be different. For now, the results in this paper are close

So which content management system is better? It depends on what you want to do with it. If you need to make a modification in a certain area such as caching, or database operations, you should look at the results in chapter 5 more closely. In particular, look at the files or classes that you want to make the modifications. If you're trying to make a business decision on what CMS to use, it could be either. As mentioned earlier, the measured results between Drupal and WordPress are too close to call. This is not bad since decision makers can rule out this attribute in their decision making process and focus on other areas such as cost, hosting, or support.

Object oriented design and programming has been around for some time and several tools and measurements have been proposed and used in measuring quality in OO design. With the rise of web based applications, it makes it difficult to use these measurements in less than ideal OO designs. However, with a little modification to the measurements such as treating files as classes, you can use these measurements to a degree. There is still a lot of work to do in this area, and while there are tools that could measure quality in PHP based applications such as PHP Unit of PHP Depend, they are still too rigid to be used on numerous web applications.

Finally, the World Wide Web is a fast moving environment and it tends to lean towards the “bleeding edge” of application development. With this fast paced environment, tools and methods will become obsolete in a short time. It is then ideal to see if these tools and methodologies could be agile enough to keep pace with the changes that are yet to come.

Bibliography

Adobe. (2012). *Adobe Dreamweaver*. Retrieved August 1, 2012, from Adobe: http://www.adobe.com/dreamweaver

Aivosto. (2011). *Cohesion Metrics*. Retrieved May 20, 2011, from aivosto.com: http://www.aivosto.com/project/help/pm-oo-cohesion.html

B., S. (2011). *PHP Unit*. Retrieved 06 01, 2011, from PHP Unit: http://www.phpunit.de

Bansiva, J. D. (2002, January). A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEE Transactions on Software Engineering, 28*(1), 4-17.

Chidamber, S. K. (1994, June). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering, 20*(6), 476-493.

DocForge. (2011, August). *Content Management Systems*. Retrieved July 15, 2012, from DocForge: http://docforge.com/wiki/Content_management_system

Ghosheh, E. B. (2008). Design Metrics for Web Application Maintainability Measurement. *AICCSA '08 Proceedings of the 2008 IEEE/ACS International Conference on Computer Systems and Applications.* . Washington DC: ACM.

Group, T. P. (2011). *History of PHP and Related Projects*. Retrieved May 3, 2011, from php.net: http://us2.php.net/history

Henderson-Sellers B., C. L. (1996). Coupling and Cohesion (Towards a Valid Metrics Suite for Object-Oriented Analysis and Design). In *Object-Oriented Systems* (pp. 143-158).

Hunter, L. (2008). *The Drupal Overview*. Retrieved May 1, 2011, from Drupal.com: http://www.drupal.org/node/265726

Li, W. H. (1993). *Object Oriented Metrics Which Predict Maintainability*. Department of Computer Science, Virginia Polytech Institute and State University.

Lincke, R. L. (2007). *Compendium of Software Quality Standards and Metrics - Version 1.0*. Retrieved May 2, 2011, from Arisa.se: http://www.arisa.se/compendium

Manual, P. (2012). *Software Metrics Supported by PHP Depend*. Retrieved July 1, 2012, from pdepend.org: http://www.pdepend.org/documentation/software-metrics/index.html

Marchetto, A. (2004). A Concerns-based Metrics Suite for Web Applications. *INFOCOMP journal of computer science, 4*(3).

Martin, R. (1994). *OO Design Quality Metrics.*

McCabe, T. (1976, July). A Complexity Measurement. *IEEE Transactions on Software Engineering*, *2*(4), 308-320.

Monaco, A. (2012, June). *A View Inside the Cloud*. Retrieved August 1, 2012, from IEEE: http://w3techs.com/technologies/cross/content_management/ranking

Palmer, J. (2002, June). Website Usability, Design, and Performance Metrics. *Information Systems Research*, *13*(2).

Pichler, M. (2011). *PHP Depend*. Retrieved May 5, 2011, from PHP Depend: http://www.pdepend.org

USLegal. (2012). *Digital Distribution Law & Legal Definition*. Retrieved August 1, 2012, from USLegal: definitions.usLegal.com/d/digital-distribution

W3Techs. (2012, July). *Usage of Content Management Systems Broken Down by Ranking*. Retrieved July 28, 2012, from W3Techs: http://w3techs.com/technologies/cross/content_management/ranking

WordPress.org. (2010). *About WordPress*. Retrieved May 1, 2011, from WordPress.org: http://WordPress.org/about

WordPress.org. (2010). *The Loop*. Retrieved May 1, 2011, from WordPress.org: http://codex.WordPress.org/The_Loop