CWX CODEWORKS 2010

No hotel food. No travel. Just great PHP. **No hotel food.** No t
No hotel food. No travel. Just great PHP. No hotel food. No travel. J
No hotel food. **No travel.** Just great PHP. No hotel food. No travel. J
No hotel food. No travel. Just great PHP. No hotel food. No travel. J
No hotel food. No travel. **Just great PHP.** No hotel food. No t

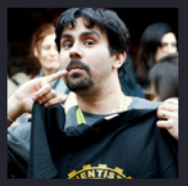
SEATTLE ▪ PORTLAND ▪ AUSTIN ▪ BALTIMORE ▪ ORLANDO

SPEAKERS


Marco Tabini


Cal Evans


Ryan Stewart


Keith Casey

PRESENTED BY

php architect
www.phparch.com

Adobe

http://CODEWORKS.PHPARCH.COM

REGISTER NOW

Refactoring in the Real World

by D. Keith Casey, Jr.

We all talk about how to best structure our projects and how to write 'clean code', but how do we clean it up? How do we find problems in a large, legacy codebase and clean them up responsibly?

Background on web2project

While most people haven't heard of Web2project, many have heard of our parent, dotProject. In late 2007, dotProject faced a crisis. We had over 200,000 lines of code spanning over 7 years of development all the way back to PHP 3. That legacy was reflected in poor object models, a lack of coding standards and a mixed up, pseudo-MVC, structure. The codebase was difficult to debug and even harder to extend. Installing some modules required a skill for creative copy/pasting and generally, a high

REQUIREMENTS

PHP: *5.2+*

Other Software:

- Phing - http://phing.info/trac/
- phploc - http://github.com/sebastianbergmann/phploc
- phpcpd - http://github.com/sebastianbergmann/phpcpd
- phpmd - http://github.com/manuelpichler/phpmd

Related URLs:

- http://blueparabola.com/blog/subversion-commit-hooks-php
- http://github.com/caseysoftware/web2project/blob/master/unit_tests/build.xml

tolerance for pain. As a result, half of the dotProject team wished to start over the "right way" and set out on that path. The other half wanted to refactor the current codebase and formed the Web2project team. I was recruited into the latter and have served on that team ever since.

Starting Point

When we set out to refactor the system, we knew there was a huge effort in front of us. There were coding standard problems, hundreds of bugs, major performance issues, HTML validation problems, cross-browser JavaScript problems and errors in how permissions and security worked. Since we couldn't tackle everything at once, we decided to focus on performance and stability first and foremost. It is not important how many features we have nor how XHTML compliant the system is if it's not reliable.

On the performance front, we added some simple debugging and found that nearly 50% of page load time was spent retrieving our database objects, nearly 40% was permissions checking and only the remaining 10% was spent processing the data and rendering it for the user. Our first step was exploring MySQL's `Explain` command and the `Slow Query Log`. As a result, we found there to be a total lack of database indexation and numerous, redundant joins. By adding indexes to the proper fields, we immediately eliminated 95% of the database time.

The permissions checking was quite a bit more difficult and required a team member, Pedro Azevedo, to build a permissions caching layer that only rebuilds permissions when they change; instead of on every request. As a result, we eliminated nearly 98% of the time required. Now that our pages were loading in less than 1/10th of the original time, we could focus on stability.

For stability and general debugging, we had to take a completely different approach. We knew that no one thing would solve 50% of our problems. Therefore, we started by exploring the tools available; at the time, there weren't many. CodeSniffer was - and continues to be - useful for checking our code against coding standards, like PEAR, but it just identifies sloppiness, not necessarily problems. Therefore, we started a completely manual process to detect, review, and refactor duplicate-but-slightly-different code. To be honest, we didn't have a clue, and we didn't know where to find one.

While the manual process worked, it was time-consuming, prone to errors, and slightly less painful than stubbing your toe at 2am while you're half asleep. **Scratch that.** It was more painful and resulted in similar expletives. The bug reports for the v1.0 Release Candidate demonstrate this. Regardless, we started by looking at areas with similar functionality and refactoring where we could. Not surprisingly, the most common actions - like retrieving a list of active projects or getting a list of the users' assigned tasks, were implemented a half dozen different ways. Each way was slightly different than the one before in terms of query structure and respecting system-wide permissions. The worst part is that most of them were just a little wrong. By focusing on these areas first, we were able to trim a few thousand lines of code and eliminate numerous issues and behavior oddities on different screens. It was a starting point and allowed us to make sure ACLs were respected and queries were well-structured, but it was difficult and slow going.

Applying Tools to the Process

We needed a new strategy that would actually work long term. Luckily about that time, we added a new team member, Trevor Morse, who brought another perspective into the group. He dedicated himself to building a solid `Unit Test` suite. Since the Companies class had been problematic but still relatively straightforward with few external dependencies, he started there. In this case, the key was starting with one and only one class to get things rolling. Once he had a basic framework for running the tests, others joined in writing Unit tests for areas in which they were having trouble. In a span of 60 days, we went from zero to nearly ninety Unit Tests; however, we ran into problems with executing them easily.

In order to run the tests easily and consistently, I created the Phing build script visible in Listing 1. Phing is like Ant or `make` but specifically for and extensible with PHP. Using Phing, you can wire together a set of tasks to run in sequence to perform tasks such as Subversion checkouts, deleting directories, running unit tests, or even creating zip/tar.gz files. The key feature for us was its ability to recursively explore a directory structure, retrieve all the unit tests, and execute them. In a matter of minutes, we had a script that would run all of our

tests as often as we needed with a single command. As an added benefit, many IDEs like Eclipse PDT and NetBeans can execute build scripts automatically as needed.

Next, we used one of the single most underrated tools in the PHP world: **php lint**. Lint simply validates PHP files and throws an **E_Fatal** if there are syntax errors. The best part is that it's built into PHP itself, so anyone can run it with:

```
php -l filename.php
```

Ideally, this would be part of our Subversion pre-commit hook to prevent bad code from even making it into the repository, but we took what we could get. More importantly, Phing implements lint natively as seen below:

```
<target name="lint">
   <phplint haltonfailure="true">
      <fileset dir="..">
         <include name="**/*.php"/>
      </fileset>
   </phplint>
</target>
```

Whilst we don't have any E_Fatal errors in web2project, this makes sure we keep it that way.

Next, we needed to the tackle code quality itself. Luckily, in the few months between when we started the main refactoring effort and looking for better methods, three tools came to light. The first was **phploc** which simply measures the "size" of a PHP project. Size is calculated through simple metrics like file and directory count and total lines of code in addition to overall Cyclomatic Complexity and the average number of lines of code per/class or per/method. To run it, from the phploc directory, type:

```
php phploc ../web2project/
```

In general, we look at these metrics on a project level, but by specifying a module or class in the command, we can analyze a smaller set of files. This is where the we can draw some interesting conclusions.

At a project level, the average class is about 800 lines with an average method size of about 80 lines and Cyclomatic Complexity of about 8. Overall, these numbers should come down, but they're not a bad starting point. Unfortunately, when we look at a few specific classes, the story is completely different. The Tasks module has nearly 8000 lines with an average method size of about 120 lines and Cyclomatic Complexity of about 13. On the other hand, the Project Designer module has one quarter of the code and a Cyclomatic Complexity of over 60. For context, a Cyclomatic Complexity over 11 is considered "highly complex". Now we have a starting series of points on which to focus our efforts.

Next, we put **phpcpd** to work. Not surprisingly, **phpcpd** or the "PHP Copy/Paste Detector" does exactly what its name implies and identifies duplicated code within a project. By typing: **php phpcpd ../web2project/** from the **phpcpd** directory, we get back a list of files with specific line numbers where duplicate snippets occur. Before the cleanup effort began, we had over 200,000 lines of code with nearly 8% duplication, for a total of 16,000 duplicate lines. Once again, we have specific places to focus our refactorings. Following the bulk of our cleanup effort, the results were amazing. We had removed nearly 15,000 lines of code and reached a 2% duplication. Unfortunately, the project as a whole isn't the important part. Applying the same detection to specific areas of a project is more informative. Further, when duplicate lines are found, it is useful to explore the area around the samples to determine if there are near-duplicate lines surrounding it. In that case, it may make the most sense to refactor those areas and increasing the duplication before refactoring it away.

While using **phpcpd** regularly, that pattern has emerged time and time again. Obviously, as we eliminate duplicate code, the percentage goes down. More importantly, as we clean up code to be more

LISTING 1

```
1. <?xml version="1.0"?>
2. <project name="web2project Unit Tests" basedir="." default="report">
3.     <target name="report">
4.         <phpunit codecoverage="true" haltonfailure="true"
haltonerror="true">
5.             <formatter type="plain" usefile="false"/>
6.             <batchtest>
7.                 <fileset dir="modules">
8.                     <include name="*.test.php"/>
9.                 </fileset>
10.            </batchtest>
11.
12.            <formatter  type="xml" todir="reports" outfile="logfile.xml"/>
13.        </phpunit>
14.        <phpunitreport  infile="reports/logfile.xml"
15.            styledir="/usr/share/php/data/phing/etc/"
16.            todir="reports"
17.            format="noframes" />
18.    </target>
19. </project>
```

similar to eventually refactor away, our percentage increases. Therefore, throughout development, we see this percentage drift up as a big refactoring approaches and drops immediately thereafter. At a project level, this drift is hidden by the sheer amount of code but within individual modules, it is a visible and dramatic sign of changes required.

The most important tool that reached maturity during this effort was **phpmd** or the "PHP Mess Detector". While it is the most powerful tool, it tends to be the most complicated. By running:

```
php phpmd.php ../web2project html codesize,design,naming,unusedcode > ../reports.html
```

We get a simple table of all the issues caught by the rulesets referenced. Once again, this isn't a tool that solves problems, it simply directs us to where the biggest problems are. The most common problems in code generally seem to be Cyclomatic Complexity, NPath Complexity, and methods that are too long. The single most useful ruleset of web2project was `unusedcode`. This highlighted an entire library that was included but not actually used. Removing it eliminated over **20,000 lines of code**. For sanity's sake, NPath Complexity should not exceed **200** while methods should generally stay under **100** lines. Not surprisingly, web2project suffered from many of these problems and more. *One particular legacy function* - **projects_list_data** - uses and modifies numerous globals instead of parameters which gives it a complexity of over **140 million**. **No, I'm not kidding.**

By using each of these three tools in conjunction to detect and triage problems by size or complexity, we can apply standard refactoring techniques. This has allowed us to eliminate tens of thousands of lines of code while making the system less complex. More importantly, through refactoring these duplicate snippets, we've closed literally dozens of bugs and eliminated subtle behavior differences of different parts of the system. But more than anything, the most important part has been the improved stability in the development process itself.

Finally, once all of these pieces were in place and the development process was smoother, the next logical step was to simplify the packaging and distribution. While it is trivial to generate zip and tar.gz files manually, automating it via Phing ensures it is performed exactly the same every single time. A simple version of the packaging process is included in Listing 2. It simply exports from Subversion, applies PHP Lint for one last sanity check, and then compresses the directory for later distribution. In almost no time, we have a stable, repeatable, build process that eliminates most of the places where stupid mistakes are made. One less area for problems is one less area to debug. While it was technically complete, we discovered that a more robust process was possible.

In the next step, we take the packaging process quite a bit further. By using **phing package** and giving a specific tag or trunk, Phing handles the rest. It applies the lint check, creates a complete changelog based on Subversion's log capability, and performs some simple JavaScript and CSS minification to reduce the overall size of the application. Assuming all of those steps are successful, the zip and tar.gz files are created. Although this did absolutely nothing for code quality, the 2.5 minutes required is well worth it as the files come out exactly as planned each and every single time. More importantly, this process allows us to create a release at almost any time and have some confidence in the actual system. *The full Phing script for this step is included in the 'Related URLs' section*.

LISTING 2

```
<target name="package">
   <svnexport
      svnpath="/usr/bin/svn"
      repositoryurl="https://web2project.svn.sourceforge.net/svnroot/web2project/trunk/"
      todir="web2project"/>

   <delete dir="web2project/unit_tests/" />

   <phplint haltonfailure="true">
      <fileset dir="web2project">
         <include name="**/*.php" />
         <exclude name="lib/" />
      </fileset>
   </phplint>

   <tar destfile="web2project.tar.gz" compression="gzip">
      <fileset dir="web2project/">
         <include name="**/**" />
         <exclude name="unit_tests/" />
      </fileset>
   </tar>
   <zip destfile="web2project.zip">
      <fileset dir="web2project/">
         <include name="**/**" />
         <exclude name="unit_tests/" />
      </fileset>
   </zip>

   <delete dir="web2project/" />
</target>
```

Results

Not surprisingly, the effort started slowly, gained momentum as we refactored, and has slowed again now that the most visible and easiest problems are resolved. Regardless, after two years of being an independent project, we had closed over 150 issues, added 40 features, eliminated nearly 65,000 lines of code, reducing our duplication from 8% to 2.1%. *This is the equivalent of closing an issue every five days, adding a new feature every other week, and deleting 89 lines of code every day...for two years.* In addition, in the last 16 months, we've added nearly 400 unit tests which works out to a new test every 32 hours.

One of the side benefits has been completely unexpected. With a class hierarchy that is better structured both logically and file-wise, editors such as Eclipse PDT and NetBeans can interpret the files properly and support code completion and features like "Go to Declaration". Previously, these functions would function sporadically at best. This has made debugging and refactoring become a task measured in minutes instead of hours or days.

And finally, now that we practice coding standards and use class structures conventions instead of just talking about them, we can raise our expectations for the community. The ability to hold everyone to the same standards is freeing and can simplify many potentially contentious disagreements. In some cases, we've had to clean up a module to show the process and give clear examples, but the community in general is starting to practice the same. After all, no one wants to be the guy writing the sloppy code.

Conclusion

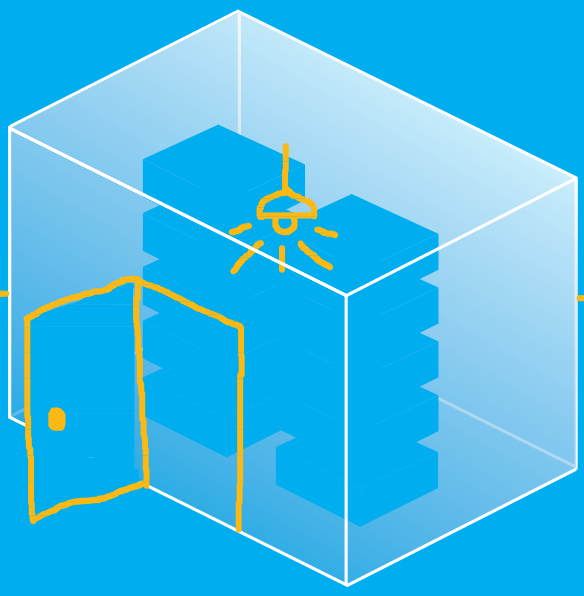
Overall, the effort has been large, sometimes difficult, and has slowed our development of *new* features. That said, the code is faster, cleaner, more logical, and there's less code, period. Since the cleanup, though, new features, functionality and entirely new modules have been easier and faster to build. Most importantly, the extension modules can reuse more of the core system logic which means less code to maintain and the sometimes-complex ACL system is respected. While I would do it all over again, I would start using these tools earlier to help prioritize our own efforts better. The important part to remember is that none of these are one-time use tools. At minimum, unit tests should be run before and after each commit, though ideally, they could be part of a continuous integration system. The rest of the tools don't need to be used continuously, but should be run regularly. I collect the output of each tool around the first of each month and when we make a release. So far, it has been an excellent way to track our progress and determine which areas need attention.

One of the things not discussed here is the value of code reviews. Random people throughout the PHP community have assisted in one way or another, but specific people have gone above and beyond all expectations. People such as Sebastian Bergmann (author of many of the tools above), Arne Blankerts, Stefan Priebsch, and Matthew Turland have contributed specific and tangible recommendations, many of which we've taken in the last year.

To pay the bills, **D. KEITH CASEY, JR.** works with Blue Parabola, LLC and has developed large-scale PHP-based systems for organizations ranging from news channels to small non-profits. In his spare time, he is a core contributor to web2project, blogs regularly at http://CaseySoftware.com/blog and is completely fascinated by monkeys.

NEXCESS

beyond hosting.



HOSTING THAT'S DOWNRIGHT

DEDICATED.

Not robots, we're staffed by actual devoted human beings. At NEXCESS, we've focused on one thing since day one: superior hosting solutions. Today, our focus hasn't changed, but our products, our services and our facilities have. We are committed to providing reliable, scalable, dedicated hosting services to clients of all shapes and sizes. Come find out how we can take your hosting to the NEX level.

RELIABLE | DEDICATED | TANGIBLE | SCALABLE | WWW.NEXCESS.NET