

Large Scale PHP

Jonathan Oxer <jon@oxer.com.au>
Open Source Developers Conference
Melbourne, Australia
December 3rd, 2004

While it was originally only intended as a templating tool PHP is now being used for some very large-scale projects.

A “large-scale” project means different things to different people. It could mean a very small amount of scripting in a simple webpage that happens to be accessed by hundreds of thousands of users every day. At the other end of the scale it could be an enormous project involving half a million lines of code, but only accessed by three people. But most commonly it's a combination of both, with a respectably large codebase being used by a medium to large number of people.

Many large-scale PHP projects involve a number of developers collaborating on a common codebase. While this is a common scenario for developers working in other languages such as C, PHP has only recently started to be used for large-scale projects and so PHP developers often have less experience working as part of a development team and using accepted software engineering practices.

This tutorial examines some of the software engineering practices relating to managing or working within a team on a medium to large project, and covers topics such as automatically generating internal documentation, managing a codebase using revision control systems such as Subversion, using a bug tracking system, and tying development tools together to create a semi-automated development environment.

Deploying A Bug Tracking System

Most major software projects use a BTS (bug tracking system) to assign, prioritise and report on tasks and bugs. Even small projects could benefit greatly from a BTS and once software developers start using one to manage their day to day tasks it can be hard to imagine working without one. In its simplest form a BTS is really just a list of jobs that have to be done: a glorified To Do list. However, a BTS can be much more than that and most systems offer a huge range of features that will streamline the development process.

Well-known bug tracking systems such as Bugzilla and Debbugs are very powerful but they can be difficult to install and are daunting for even large development teams to deploy. In this presentation I will focus on Flyspray (<http://flyspray.rocks.cc/>), a BTS written in PHP and using either MySQL or PostgreSQL for storage.

Flyspray was originally developed for the Psi project as an easy-to-use BTS for internal use, but it has since attracted a lot of attention as a good general purpose system and is now in use by thousands of people around the world for a wide variety of projects. It has been created with the stated design goals of usability, simplicity and elegance but without compromising features. As a result it can provide basic BTS functions with minimal pain but can grow with a project as its requirements become more complex. Flyspray offers a number of advanced features including email and Jabber notifications, task reminders, definable user groups, task categories with category owners, multi-project support, customisable interface themes, multiple language packs, event history, file attachments, status reports, and hooks for external systems such as auto-closing bug reports on Subversion commits.

An introduction to Flyspray is available online as a PDF user guide titled *Killing Bugs With Flyspray*, which you can download from <http://jon.oxer.com.au/flyspray/>

Source Code Management Systems

Probably the single most useful tool for a software development team (ranking second only to a good text editor in my opinion, and not by much!) is a source code management system (SCM).

When using an SCM all members of a software development team keep their own local working copy of the project source code. A developer can make a change to their local working copy and have it propagated out to all the other developers, and likewise have every other developer's changes merged semi-automatically back into the local working copy. Conceptually it can be considered to be multi-directional synchronisation, allowing multiple developers to work on the same project (or even the same file) at the same time without stepping on each other's toes.

While source code management systems provide a framework to merge and store project source code, that's just the start of it: they also allow developers to move backwards and forwards through the development tree chronologically and examine every change that has been made to the code (revisions), and move sideways through the tree to examine parallel alternative versions of the code (branches).

SCMs can be broadly divided in a couple of ways: either centralized or decentralized, and either changeset oriented or patch oriented.

Centralized vs Decentralized

Most source code management systems are centralized and are therefore based around the concept of a single "repository" which is the master copy of the source code, and multiple "working copies" which are where changes are actually made. The repository generally resides on an Internet-connected server somewhere, while the working copies reside in a location convenient for each developer to work on such as on their workstation or a development server. Changes made to a working copy by one developer are merged up to the repository, and then merged down to all the other working copies.

Decentralized systems are a much newer approach, and they basically work by meshing a bunch of working copies together without any one copy of the source code being the master copy. Changes made to a working copy by one developer are propagated sideways directly to other working copies. That allows for some interesting advantages like being able to perform SCM operations even when offline, but also leads to some interesting problems like being able to define the "definitive" version of a particular source tree. Advocates of this approach would say of course that any working copy could be nominated as being the "definitive" version, but for most developers it doesn't yet feel like a natural way to work.

Changeset vs Patch Oriented

Source code management systems can also be roughly divided into being either changeset oriented or patch oriented. Being changeset oriented means that the SCM keeps a master copy and changes are applied to it sequentially and cumulatively. A change made by one user and applied to the repository results in the repository version being incremented, and a subsequent change made increments it again. Therefore the code in the repository is the sum of all previously applied changes, one after another. For example, with a changeset oriented SCM starting with version 1 of the code, applying patch A will take it to revision 2 and applying patch B will take it to revision 3.

When an SCM is patch oriented it treats changes in a much more loosely-coupled way. For example, a changeset oriented SCM that applies changes A and B to take the code to version 3 provides no way to see the code with only patch B applied and not patch A. However, a patch oriented SCM would allow you to do exactly that: a developer can choose to view a source tree with only a discrete subset of changes applied if they want to.

Once again, patch-oriented systems are less well known by developers and so they don't feel like such a natural way to work. And as with decentralized SCMs, it can be harder to specify what the “definitive” version of the source code is.

Choosing An SCM

As a major generalisation, traditional SCMs are centralised and changeset-oriented while newer SCMs are decentralized and patch-oriented.

Of course any statement like that is bound to insult a large proportion of SCM developers!

The most traditional SCM of all is probably CVS, or Concurrent Versions System. It is centralised and changeset oriented, and is almost certainly the most widely deployed SCM of all time. However, it has a few limitations and is starting to show its age. For example, it can't handle file renames or copies, doesn't support file meta-data, commits are not truly atomic, and it doesn't handle binary files properly.

Even with those shortcomings it still uses a logical and well understood operational model, so most experienced developers are very familiar with the process of working with it. It makes sense then for a modern SCM to basically follow the CVS operational design, but to re-implement it from scratch in a way that allows those shortcomings to be overcome. In fact that's exactly what the Subversion project is all about, and as a result Subversion is rapidly rising to the fore as the most popular up-and-coming SCM after CVS.

My personal favorite SCM at this time is Subversion, but there are a number of other very interesting alternatives that are also worth looking at including Arch which is patch-oriented. To complicate things there are also layering projects like SVK, which is a project to write a system that uses SVN for the back-end storage engine but wraps it inside an extra layer that decentralizes the overall architecture. Basically it's like Subversion but without the requirement for a central master repository.

There is a neat little overview of various SCMs at the Better SCM Initiative site at <http://better-scm.berlios.de/>. Some other interesting starting points include:

- Subversion: <http://subversion.tigris.org>
- Arch: <http://gnuarch.org>
- SVK: <http://svk.elixus.org>

Internal Documentation

To most developers “documentation” is a four letter word, but once a software project grows beyond a few thousand lines of code good documentation can be a big timesaver in a number of ways. I'm not talking about external documentation (user manuals, etc) but internal documentation: code comments, API docs, architectural docs, database schemas, etc.

Most programmers take the attitude that internal documentation is unnecessary for two reasons. Firstly because once code has been written and proved to work the job is done, and it seems like a waste of time to then go back and write docs when there are new and interesting problems to solve. This is usually reinforced by management who care more about getting things done than about doing them properly with a view to saving future effort. Secondly because there is an elitist attitude that the code *is* the documentation, and if anyone is going to go through my code then by God they should be able to read it like a book or they don't have any business being there and they should leave things to the professionals [insert sound of chest-beating here].

There are benefits though, and once you get into the habit of it and set yourself up with some tools

you'll find that generating internal docs is easier than you might think.

Firstly the obvious benefit is as a memory-refresher for yourself and your team-mates as you revise or extend existing code. No matter how much chest beating they do just about any programmer has had the experience of looking back at old code and wondering what the hell they were thinking.

Secondly, internal docs can greatly assist when bringing new people into a development team. We've all heard the statement that adding developers to a late project will only make the project later. The reason is that the new hires need a lot of help to get themselves oriented, and the people providing that help are the experienced developers who should be spending their time developing. The end result is that less work gets done overall than before the extra programmers joined the team. While nothing will make that effect go away entirely, internal documentation can greatly reduce it.

The minimum-effort first step is to start using a system to automatically generate documentation based on code comments. The concept was popularised by Javadoc, which allows Java programmers to write comments within the code and then run a program that extracts them and builds documentation automatically. The end result is that you don't need to spend hours writing up docs in your word processor: just write code as usual but make sure you add appropriate comments, and periodically re-run a program that creates the actual docs. If you're feeling particularly lazy-like-a-fox you can set up a cron job to do it for you.

Not surprisingly the PHP implementations of Javadoc have come to be called PHPDoc, and there are a number of them available including:

- PHPDoctor (<http://phpdoctor.sourceforge.net/>)
- phpDocumentor (<http://www.phpdoc.org/>)
- phpdocgen (<http://galland.stephane.free.fr/arakhne.org/phpdocgen/>)

Writing comments in a format suitable for Javadoc itself is very well documented at <http://java.sun.com/j2se/javadoc/writingdoccomments/>. The various PHPDoc implementations try to follow the Javadoc standard as much as possible so it's worthwhile referring to the Javadoc docs to understand the format to follow.

Basically what you need to do is put a “DocBlock” just before any construct, class, function or method. A DocBlock looks something like this:

```
/**
 * This is my groovy function to multiply two numbers
 * @param integer $width The width of the wall
 * @param integer $height The height of the wall
 * @return integer $area The total area of the wall
 */
```

There are lots of tags you can put in a DocBlock, but the important ones are the initial description of what the chunk of code is meant to do, parameters that are passed to it, and what it returns.

Once appropriate comments have been placed in the code you can run a PHPDoc program which walks through the source code, finds the comments, and generates documentation in various forms including HTML, XML, and PDF. These docs can then be made available to your developers internally by placing them on an internal server or distributed directly within the source tree by your SCM.

Tying Systems Together

All these systems are great on their own, but things get really interesting when you start chaining them together to create a streamlined development environment.

The first thing to look at in detail are hooks available within your chosen SCM. For example, Subversion provides hook scripts such as “pre-commit” and “post-commit”, which are called before and after a change is committed to the repository. These scripts are passed useful information such as the revision number, the developer who made the change, the changelog entry, etc. You can then use these scripts to call other things you define. SCM hooks are the glue that bind an automated development environment together, since most events need to be triggered when changes are made to the SCM repository.

Automatic Change Emails

Subversion comes with an example script which can be called from the post-commit hook to send an email every time a commit is performed. The message can contain the changelog entry, a list of files affected, the date and time, the name of the developer, and even diffs of all the changes. At Internet Vision Technologies we have this email (minus the diffs) sent to an internal mailing list that the entire development team is subscribed to, making it very easy to keep track of what changes have been made by other people. It also provides a conveniently searchable history of all files, since you can then use your email client to find all changelog emails that mention a specific file to see everything that's been done to it.

Automatic Bug Closures

Committing a change often closes a bug report, requiring the developer to then manually go to the BTS and close the item they just fixed. This can be automated using post-commit hooks as well. For example, I've written a hook script that links the Subversion SCM to the Flyspray BTS and scans the changelog of every commit looking for lines in a special format like this:

```
Closes: #123
```

It then connects to Flyspray and closes the corresponding item. Flyspray can then perform its usual actions, like sending out a notification email to users who were tracking that bug to tell them it has been closed.

Rebuilding Documentation

When a commit has been made the code has been changed, so usually the documentation needs to be rebuilt and once again SCM hooks come to the rescue. A post-commit hook could trigger a PHPDoc program to traverse the source tree and build an updated version of the docs. No need to have the docs out of date or manually rebuilt when they can be kept up to date totally automatically.

About The Author

Jonathan Oxer is founder and Technical Director of Internet Vision Technologies, an Australian web application development agency with clients around the world. He is also a Debian developer, and organised Debian Miniconf 2 in Perth in January 2003 and Debian Miniconf 3 in Adelaide in January 2004 in association with Linux Conf Australia where he also presented technical papers. He has spoken on various Internet-related topics at both corporate and government seminars and conferences around the world, and is a regular guest on radio station RRR's “Byte Into It” segment. He sits on the Advisory Group of Swinburne's Centre for Collaborative Business Innovation. His first book, *How To Build A Web Website And Stay Sane* (www.stay-sane.com), will be published soon, and his second book, *The Debian Universe* (www.debianuniverse.com), is being written live on-line. He has been retained by O'Reilly to work on *Running Debian GNU/Linux*, and his fourth and fifth books (*Web-Based Business Intelligence* and *Disaster Proofing For Small Networks*) are already underway. Jonathan can be reached at jon@ivt.com.au.