Prevent Conflicts in Distributed Agile PHP Projects

by Yegor Bugayenko

Parallel programming in a distributed team is a tricky and risky process, especially if you want your project to be successful and delivered on time. Subversion helps isolate programmers in their branches, but when they start to reintegrate into trunk, conflicts may effectively ruin hours or days of work. In this article, I will share a few best practices that help our team manage conflicts and streamline our development process.

REQUIREMENTS

PHP: *5.2+*

Other Software: *Subversion 1.5+*

Related URLs:

http://subversion.tigris.org

http://code.google.com

http://www.fazend.com

Distributed development of software over the last decade has been growing in popularity. This is mostly due to the continuous improvement of the quality of communication channels and the emergence of new methods of interaction between us over the Internet. Nowadays, brick-and-mortar offices become less effective when a software project requires the participation of engineers with different skills, experience, and price.

Distributed PHP teams can be very productive, provided they are organized and empowered by the necessary instruments. One such instrument is

control systems. In this article, I'll discuss multi-branched parallel development and the most popular types of conflicts that can lead to eventual project collapses if not planned, controlled and resolved in time. I will also suggest a number of principles and best practices that may help you avoid such conflicts and streamline the development process.

Why Do We Need Version Control?

It's difficult to imagine a software team (or even a programmer working alone) that will keep its source code without some form of version control. As M. Pilato, et al. explain in *Version Control with Subversion*, those who

Perforce, etc.) know that it helps them to:

-

logical order

-

ferent people at the same time

recommend hosting it via http://code.google.com if the project is open source, and http://fazend.com if otherwise. Once you set up a new account and your

its root. You need to "check out" the source code to your laptop, make changes locally (add new files, alter existing files, or delete obsolete files) and "commit" your changes back to the server, like this:

```
$ svn checkout svn://svn.fazend.com/myRepo/trunk myRepo
Checked out revision 1.
$ cd myRepo
$ echo "<?php echo 'Hello, world';" > index.php
$ svn add index.php
A         index.php
$ svn commit -m "My first simple application"
Adding         index.php
Transmitting file data .
Committed revision 2.
```

Once the code is committed to the server, it becomes available to other programmers from your team and is securely stored on the server. You don't need to worry about backups as it's done by the server free of charge.

Why Do We Need Branches?

Sooner or later, you experience a necessity to use

"branching patterns" that tell us when and how we may create branches; Brad Appleton, et al. have described many at http://www.cmcrossroads.com/bradapp/acme/branching. To make a long story short, we are creating a new branch when we need to isolate one development stream from another. We want to make changes to the code but leave the original version untouched and accessible. In most cases, we don't want those who use the original version to know that a new branch was started and changes made.

For example, we met a new customer for our small

This new customer asks us to tailor the application for them. We know this change won't be suitable for other customers that we already have and want to make the requested change just for this one customer - isolated, so to speak. So we will need a new branch, and we'll create one like this:

```
$ cd myRepo
$ svn copy ^/trunk ^/branches/specific-customer \
  -m "branch created"

Committed revision 3.
$ svn switch ^/branches/specific-customer
At revision 3.
$ echo "<?php echo 'Good bye, world';" > index.php
$ svn commit -m "new version implemented"
Sending         index.php
Transmitting file data .
Committed revision 4.
```

Why Do We Need (!) Conflicts?

Typically, everything works fine until one day, we decide to merge changes made in the branch with the original version (often called **trunk**
we need to do this when the changes made in

> We spent 10 full weeks with 3 programmers just to get it 80% covered by unit tests and compliant to phpcs/ phpmd rules.

the isolated branch would benefit **trunk** and other programmers, and very often, we do it on purpose, meaning we create a branch with the plan to merge it back into **trunk** ("reintegrate" the branch) once the changes made there are "good enough".

merging operations and resolving conflicts. However, no computer system can resolve "semantic conflicts" without our personal participation. Consider this example semantic conflict:

```
$ cd myRepo
$ svn switch ^/trunk
U    index.php
Updated to revision 4.
$ echo "<?php echo '<p>Good bye, world</p>';" > index.
php
$ svn commit -m "HTML formatting added"
Sending        index.php
Transmitting file data .
Committed revision 5.
$ svn merge --dry-run ^/branches/specific-customer
--- Merging r3 through r5 into '.':
C    index.php
Summary of conflicts:
  Text conflicts: 1
```

We can't reintegrate our branch to **trunk** because of the changes made to the original version of the **index.php** file.

We have to understand that conflicts are instru-

Without conflicts, we would effectively and quickly turn our source code into a giant mess. Conflicts are actually the earliest indicators of our mismanagement.

There Are Four Simple Rules

I propose a principle of management for parallel programming that minimizes conflicts and streamlines the development process. There are four simple rules which, if followed, will benefit your team and the entire project:

trunk into a fortress
trunk the personal responsibility of its guard

lost)

won)

Now, we'll discuss each one in detail.

Turn Your trunk Into a Fortress

You have to see the source code in **trunk** as being in one of two states: it's either *broken* or *solid*. To calculate its current state, you need a number of specialized software packages. The most popular ones on the market that I would recommend are:

phplint validates **php/phtml** files for syntax correctness.
jslint validates all your JavaScript files for syntax correctness.
phpunit executes certain modules of your product with the intention of breaking them. If all attempts fail - the code is solid. You will have to create PHP unit tests in

order for **phpunit** to execute them.
phpmd detects potentially messy PHP blocks, like functions that are too long, unused variables or overly complex constructs.
phpcs validates every **php/phtml** file for compliance with PHP coding standards related to code formatting, indentation, variable/class/function naming, etc.
xdebug calculates source code test coverage and detects uncovered code blocks. This tool may be used only in combination with **phpunit**.
phing executes all of these "validators" in a pre-defined order and reports a summary result. Either a problem is found and the code is broken, or it is solid.

Every time you start **phing** in **trunk**, it performs all configured validations and reports whether the code is broken or solid. Once **phing** is set up, you can start making your validators more and more powerful. You will turn your **trunk** into a fortress that will protect itself against defective code. This process will never stop, and the more time and effort you invest in the fortress, the easier your life (and life of your project) will become in the long run.

Your continuous integration server should start **phing** every time a new change is committed to **trunk**. If the code remains solid after this change, the product should automatically be deployed to the production server. If the code is broken, the continuous integration server reports the problem by email and stays, waiting for a correction. The production server does not receive the broken code. This situation is never going to happen in your projects, and a bit later, you will understand why.

The sooner you start protecting your fortress, the easier it will be. When the project gets big enough, it becomes extremely difficult to add a new validator when the fortress is already broken. You should not only protect it, but make it solid beforehand. Last year, my team received a PHP project from a previous developer, and our task was to make it maintain-

covered by unit tests and compliant to **phpcs/phpmd** rules. Not surprisingly, this effort paid off. The majority of bugs disappeared, and we passed the product to the maintenance team assured they wouldn't come back to us later with a broken product. We built a fortress which is difficult to penetrate, either occasionally or by intention.

This case demonstrates that if you start building your fortress from the first day of the project with the validators in place, you won't need to spend more time later to return your code to a solid state.

Ideally, your validation mechanisms (especially unit tests) should be more complex and bigger in size than the source code itself. In one of our projects, the total size of all unit tests is three times bigger than the size of the code itself. To say the least, this project is the most stable and maintainable among all others.

You also can invent and introduce validators that are specific to your particular project. For example, you can check your XML files for validity against XML Schemas and DTDs. You can also validate correctness -

work (http://www.phprack.com). You can control the quality and completeness of your phpDoc embedded documentation, and you can check your code with **phpcpd** toolkit that will validate your code for existence/absence of copy-paste blocks.

Make trunk a Personal Responsibility of Its Guard

Now, your source code has a mechanism (empowered by **phing**) to detect whether it is broken or solid. The

next rule dictates that you forbid access to **trunk** for everybody except one person. Let's call him or her a "guard". From now on nobody is able to commit their changes to **trunk**, but everybody can read it.

authz

file, for example:

```
[myRepo:/trunk]
* = r
guard = rw
[myRepo:/branches]
* = rw
```

From now on every change in **trunk** must comply with this simple workflow:

in the branch

the branch is ready

phing manually to confirm

that the code is not broken

trunk

Ideally, the guard should not be one of the programmers. He should accept only those changes that don't break the product or penetrate "the fortress". Programmers and other engineers responsible for functionality implementation and bug fixing are very motivated to deliver their results quickly. Most often speed is the enemy of quality. This is where the guard plays an important role. They should not allow programmers to break the product by erroneous code in their branches and should say "no" to branches that compromise the entire fortress.

It is important to motivate the guard for such a "conflicting" behavior. They will be in a difficult position in front of the entire team. It might also be a good option to have the guard located remotely, as far as possible from others.

In a bigger project, the guard and deployment manager/engineer roles can be shared. A deployment engineer in enterprise projects is responsible for the configuration of package deployment mechanisms and the development of integration scripts. This person creates a **build.xml** script for **phing** and keeps an eye on it during the entire project life cycle. They are a good candidate for the guard role.

Punish for Abandoned Branches (Attacks Lost)

When the fortress is in place with strong validators and you have a guard who is responsible for rejecting branches that break the product, it's time to establish the first rule for the team: "abandoned branches will be prosecuted". A branch is "abandoned" if it is created, commits are made, but it is never reintegrated to **trunk**. Of course, we are talking about branches that were initially planned for reintegration.

A scenario of such an abandonment may look like this: A programmer starts to implement a new feature and creates a new branch, he commits his changes to the branch and reports to the guard that the branch is ready for **trunk**, but the guard refuses to reintegrate it because the changes introduced

> Two obvious recommendations may help: reintegrate as soon as possible, and make your changes as small as possible.

break the product. The programmer makes corrections to try and solve the problem and gets back to the guard again and again. After a number of failed attempts, the programmer gives up and goes to another feature or starts a new branch for this feature.

No matter how this sad story ends, the time invested in the programming was just wasted. The changes never reach `trunk`, and this is the personal fault of the branch's author, the programmer.

To continue the analogy we are already using, the programmer attempted "an attack" on our fortress. He tried to penetrate the wall of validators and place his changes (new source code) inside it. He lost the attack, and the guard won. This was obviously a fault for the entire project: time was wasted, the feature was not implemented, and overall morale went down.

To help discourage and prevent this, try to invent a mechanism of "punishment" for such situations that are specifically tailored for your own team.

Award for Reintegrated Branches (Attacks Won)

Some "attacks" will be won by programmers and new pieces of source code will reach `trunk` and become part of the fortress. The size of the fortress will grow without losing strength. This means that the project will grow in size while staying maintainable, stable, and extendible.

Working in parallel, programmers don't make problems for each other since every change (a new feature or a bug fix) is isolated and reaches `trunk` as a transaction, meaning that it is either in or rejected.

Try to establish some motivational policy to reward those who successfully reintegrate their branches into `trunk` and close them in time. Two obvious recommendations may help: reintegrate as soon as possible, and make your changes as small as possible. The same recommendations go for the database transaction. In order to avoid deadlocks and collisions, your transactions have to be as small as possible and should happen often.

In our projects, we're trying to encourage programmers to reintegrate branches within one working day. If you start a new branch in the morning, you have to reintegrate it before the end of the day (of course, the guard will do the actual reintegration, you will just let them know that the branch is ready). Typically, changes for one branch take 2-4 working hours for one programmer.

According to our statistics, an average branch and should be split onto two (or more) smaller ones which will have to be implemented and committed it can be implemented together with another branch. In other words, we're trying to group micro features and minor bugs into aggregate branches.

Conclusion

Distributed parallel development is becoming more popular for PHP projects. However, communication problems often lead to collisions and repository conflicts, which are more severe and destructive than in collocated teams. A few recommendations on how to avoid such conflicts were discussed in this article. When used together in your project, they will protect you against major problems.

The four principles explained above may help your team, if you understand the key principle: "`trunk` is a fortress". Email me (yegor@tpc2.com) if you would like to share your own success stories of resolving conflicts in parallel programming.

YEGOR BUGAYENKO is the lead architect of the FaZend Framework and a proud holder of the ZCE, ZFCE and PMP certificates. He is also the director and co-founder of TechnoPark Corp., a custom software development company specializing in complex and distributed web applications.