Using branches in Eclipse in unexperienced XP-teams.

Miguel Ewezon
D00, Lund Institute of Technology, Sweden
d00me@efd.lth.se

February, 24 2004

Abstract This paper gives a short introduction to branches and some selected branching patterns and then discusses the experiences from using branches in Eclipse and CVS during Extreme Programming projects. The paper is supposed to clarify two things: Firstly, as the participants had no prior experiences from using branches, is it worthwhile educating them or is the overhead to large? And secondly, does an XP-project benefit from using branches — primarily for releases — or not?

Contents

1 Introduction

This paper will examine how the use of branches can simplify the development (and of course also maintenance) work in an Extreme Programming (XP)[Beck99] project. An extremely important part of the development work are the releases and hopefully this paper will show you how to simplify them. We will also see how we can increase our chances of making successful spikes.

The paper consists of two parts: The first covers some theory about branches and why and how to use them, and in the second the experiences and thoughts of the coaches and developers involved are summarized. Hopefully this will show that the overhead when using branches is insignificant compared to the positive effects (which among other should be better and less painful releases). Because the project participants had no previous experiences using branches, the first part of this paper was handed out to them, and after some introductory words followed by hands-on experience in front of the computers, they were (hopefully) ready to start with the projects.

1.1 Terms

A *branch* is often a line of development coupled to one or more changes. A *baseline* is a collection of files which is stable — frozen in time — relative to the work of single programmers. A *merge* or *propagation* is when we integrate some changes or a branch with some other code (or branch).

2 Problem

Why do we want to use branches? Well, as with every new technique that we learn, we're hoping that it will decrease our pain and increase our gain. The main reason for using branches is to make parallel development more efficient. An example of parallel development is when we are working on two releases concurrently, and another one is when we are working on a new release and at the same time are fixing bugs in the previous. But exactly *how* does it work? We can use branches to isolate changes, and to insulate developers from other's integrated changes that have yet to be integrated, built and tested. We can also use them to separate tasks/stories from each other and for numerous other situations. [ABC98]

2.1 What is a branch?

Imagine yourself the following (extremely common) scenario: You are working for a company that recently released a new version of a program. Everybody is happy and have begun implementing new stories that your customer wants. Suddenly a severe bug is discovered in the released version, and you and your partner get the assignment to find it and fix it. You quickly realize that it must be a hard nut to crack since it slipped through both the unit tests and the acceptance tests. You and your partner agrees on that you must create a stable workspace to avoid conflicts and incompatibilities between the releases. Now it is inevitable that we face the double maintenance problem[1] [Babich86], but we will rely on CVS to help us as much as possible. The solution is to create a new branch and work on that (see figure 1). Meanwhile, the other members of the team continue working on the main branch (the baseline).

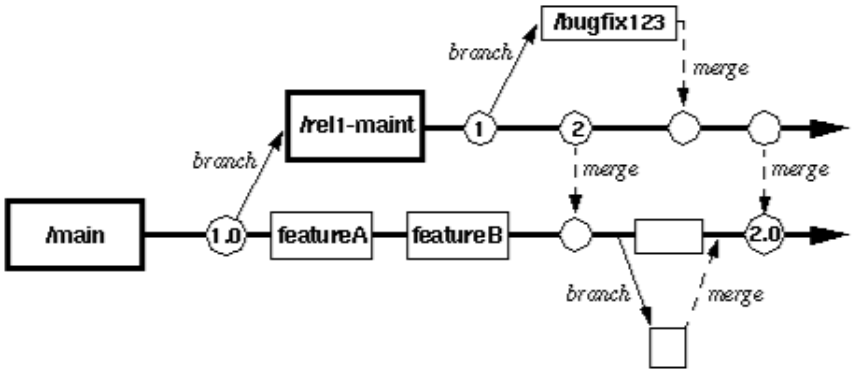

Figure 1: A sample version tree diagram for a project or system.

When you have fixed the bug, built and tested your changes there is one extremely important thing left to do: Because this change should also be implemented in the main branch (of course we don't want to have this bug in the new release) we must make sure that our changes are also made in the main branch. But, instead of just making the same changes one more time, we simply propagate (merge) our changes from the maintenance branch into the main branch. By doing this merge, we are also ending the double maintenance phase, since we now no longer have the same data on two branches.

[1]When one has the same data on two separate places, they will inevitably diverge with time.

3 Theory and analysis

3.1 The problems

When using branches, there are some common traps and pitfalls that we want to avoid. In this section, we will discuss some common problems.

3.1.1 Big-bang integrations

When should we merge? Intuitively we understand that to be able to see how well the changes on the different branches integrate with each other, we must merge often. If we wait to the end, we won't know if it will work or not. We also understand that it's better to do a couple of small merges along the way than one enormous at the end. So, it is desirable to merge very often, and thus avoiding an enormous Big-bang integration at the end (see figure 2).

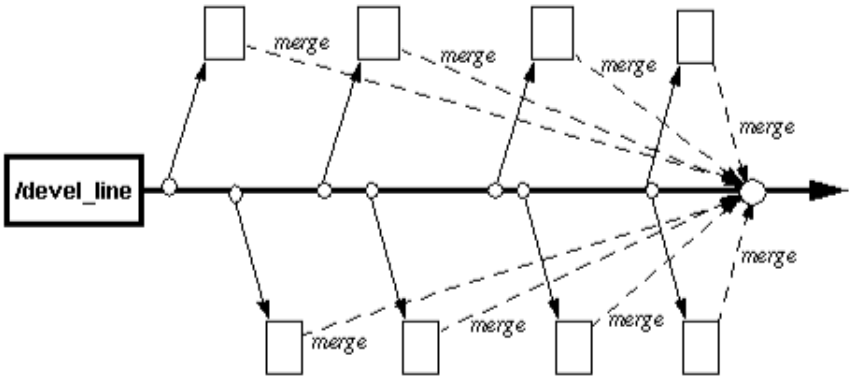

Figure 2: Big-bang integration at the end (undesirable).

3.1.2 Handling releases

How do we handle releases as efficiently as possible? Often we will need to maintain an old release at the same as we are developing a new. Only using labels and tags won't suffice.

3.1.3 Simplify concurrent changes

We want to make sure that we can handle concurrent and overlapping changes. If a project is using locking — that is, only one person at the time is allowed to work with a certain file — we tend to get long "busy wait" periods where

developers simply can't continue working because the file they need is used by someone else. How do we get rid of these stalls?

3.1.4 Version our own files

What do we do, if we are working with a large story, can't commit until the entire story is finished but we still wan't to version our changes? One alternative would be to have a private repository in our workspace, but that would probably give to much overhead and force us to do more manual work when for example merging between the two repositories.

3.1.5 Spikes

When we do spikes we often do it on (parts) of the production code. If we simply copy the current production code to the spike repository and work on it here, soon we won't know what directory to use or if we must make a new copy of the production code. At a certain time, we won't know if we are doing our spike on code that is relevant or not. And if we don't know whether our experiment is carried out on the intended code or not, our experiment isn't any good.

3.2 The solutions — Branching patterns

For each of the problems in the previous section, luckily there is a branching pattern that solves it! The first four of the following five branching patterns are from [ABC98] and the last one is from [ABE03].

3.2.1 Merge early and often (P5)

To avoid Big-bang integrations we use the pattern *Merge early and often* (see figure 3). This pattern applies to all work associated with Configuration Management (CM)-tools, not only when using branches. Don't be a Rain-man[2], make sure to update often! In an XP-project it is even more important to use this pattern, since we else will fail to follow *continuous integration*, which is a part of XP.

3.2.2 Codeline per release (C3)

To simplify releases we branch off from the baseline directly when we start working on a new release (see figure 4) and do all the development on this

[2]A person whose world is virtually blown to pieces when something in it changes. From the movie Rain main, 1988.

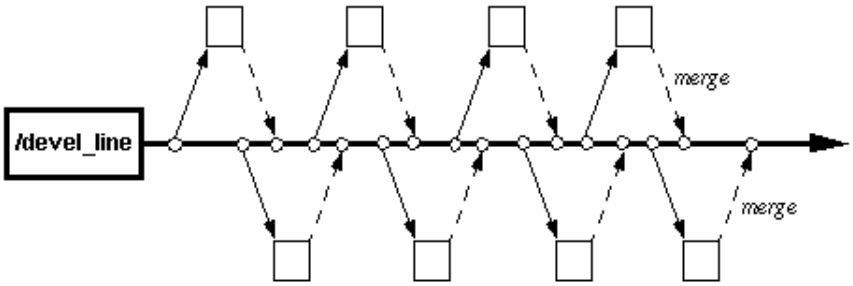

Figure 3: Merging back to the codeline after each activity-branch.

particular release on this branch (of course we may branch off also from this one). The idea is that we can have several releases which are being worked on simultaneously and still have them separated. We will use this pattern together with *Deferred Branching (P8)* (see figure 5), which implies that we should branch off as late as possible (when we decide that no new stories should be included in the coming release).

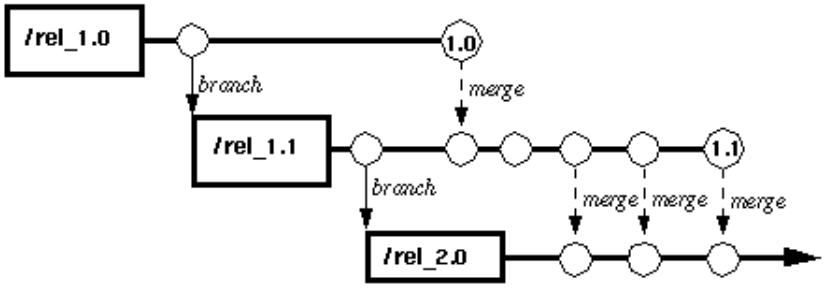

Figure 4: Codeline per Release for releases 1.0, 1.1 and 2.0.

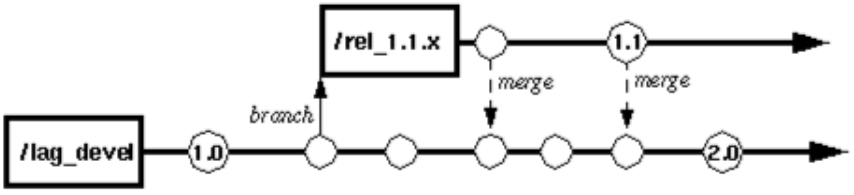

Figure 5: Deferred Branching for releases 1.0, 1.1 and 2.0.

3.2.3 Branch per task (C2)

The solution is to branch off from the main line as soon as you start working on a new task (see figure 6). If you are the only one working on the branch, you will have no problems with locking.

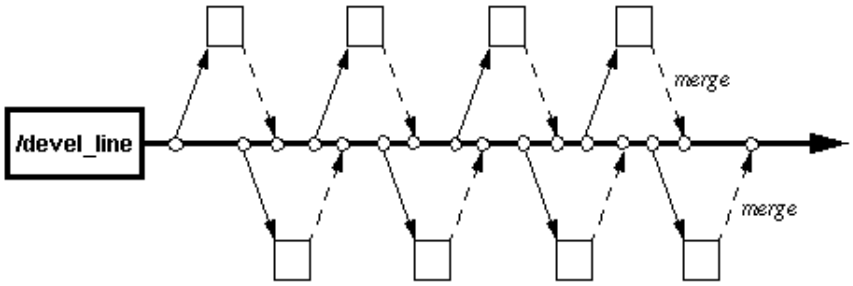

Figure 6: A separate activity-branch for each development task.

This pattern can also be used for releasing independent patches. Imagine yourself working for a company which develops a program, lets call it Doors. Also imagine that you have one (or several) customer(s) who is using a very old version of Doors on an equally old computer. One day he calls you and wants to have a fix for a bug. Certainly you say, just install our latest (enormous) bug fix and new feature package and it will clear all the bugs and give you some nice new features. But, the customer says, I don't want that. I'm perfectly satisfied with the features I have, I just want that *particular* bug fix. Sorry, can't help you there buddy, you reply. But, if you had used *Branch per task* during the development, you could have helped him and gained another happy customer. Simply let each bug fix — or for that matter each new feature (in XP, story) — be developed on its on branch. Then you just send "that branch" to the customer and he gets precisely what he asked for — no more, no less.

3.2.4 Private versioning

Branches could also be used for *private versioning*. That is when you want to be able to version your files without making them public in the repository. Well, of course they are public, but if they are on a branch called "Bobs_own_test", it will presumably be discarded by your colleagues. If your "test" goes well, you will probably merge the code into the baseline, else you will simply throw it away. Note that this isn't regarded as a spike, since private versioning is what you and your partner "experiments" with during work time. Private versioning is a special case of *Branch per task*.

3.2.5 A spike is a branch

A spike is a branch is intended to make sure that we always spike on the most recent production code. But we also get rid of the spike repository that easily gets cluttered with numerous directories. Instead of having separate repositories for production and spike code, we will use spike branches in the production repository. Whenever we need to do a spike that requires some production code to test our spike on, we make a branch in the production repository. When we are finished with our spike, we simply discard the branch. But, it is *NOT* allowed to merge spike code with production code. That way of working comes in a packet with a severe punishment...

3.3 How to use branches in Eclipse

Eclipse [Eclipse] has good support for branches. It uses three-way merge (that is, when merging it also looks on the common ancestor), while CVS uses two-way merge. Another advantage when using Eclipse is that we get a very nice graphical user interface to the repository and during merges. All this was taught with the help of [Eclipse-branching].

4 Hypothesis

This section will try to explain why we you should use the patterns described above. The *Merge early and often* should always be used, even if you are all working against the same (main) branch. Try to update as often as possible, because a couple of hours of trying to fix a big merge conflict isn't that fun.

During the earlier instances of this course there has been a lot of problems regarding the releases, especially the first one. Because of this, we want to examine if *Codeline per release* can remove some of the problems. The main advantage with this pattern is that when a release is coming up, we can say that no new stories/tasks should be implemented in this release and assign one or two pairs to use the time until the release honing the already existing code. The others will keep on implementing new stories on the new branch. This way, there will never be any uncertainties regarding neither what should be in the upcoming release nor if all stories will be ready on time.

But, branches can't solve all our problems. Don't think that branches is a hammer and everything else is a nail. But use it wisely, and you might find it very helpful.

5 Experiment

During the spiketime between the first and the second iteration I taught my group (which from now on will be referred to as team01) how to use branches. The first — nearly finished — part of this paper was handed out to the students and the day after a total of nine persons (out of fourteen in the group, twelve students and two coaches) worked through the exercise mentioned earlier. It all went very good with little problems and questions. I had intended to hold a very short lecture before the actual computer laboratory, but since the group was so small (only four pairs) I chose not to. Instead I sat down with them, answered their questions and guided them through the exercise. It took about one and a half hour for all of us to complete the exercise. The CVS-guide [Eclipse-branching] was written for Eclipse 2.1.2 and we were using Eclipse 3.0M5, so some things were different. Worth noticing is that if the students had had more experiences from Eclipse, the exercise probably would have taken only half the time.

Though five branching patterns are mentioned above, the one we really wanted to test was *Codeline per release*. So, our group was able to use branches already during iteration two.

The week after, between iteration two and three, I taught another group (team04) — in the exact same manner as explained above — how to use branches. Because I now had some experiences from teaching branches (and the group was smaller), it only took about an hour.

6 Results

The experiences from my team are that branches are very useful. In the end of the second iteration — when we determined that we would not be able to include all of the stories in the upcoming release — we branched off and two pairs continued implementing new stories on the branch while the rest of the team honed the already existing stories on the main branch. Due to an somewhat unsuccessful first release we spent iteration three fixing the bugs and errors, but a couple of pairs continued working on the branch and implementing new stories. The unsuccessfulness of the release had nothing to do with the use of branches or anything like that, instead it was when we tried to use the code for our stories together that we noticed that nothing worked! The stories worked excellent on their own and in the test cases though... So the release problems came from trying to clean up the mess and get it working.

Just before the release (1.b) we spent about an hour merging the changes

on the main branch out to the release branch. We had some merge conflicts but they were not that tedious. After the merge, the ”old” (improved) and the new stories worked nicely together. Worth to mention here is that we didn't do frequent merges between the branch and the main line, thus kind of violating one of XP's practices — the continuous integration. This was a deliberate choice, because the stories being developed on the branch were independent of the fixes being made on the main line. And furthermore, since this was the first the time that we were using branches, we didn't want to complicate it too much. On the other hand, continuous merges would most definitely have meant that the final merge would have been less simpler.

Team 04 only did one release using branches, but they felt that it worked well. Due to the deadline of this paper, the results and experiences from the two final releases have unfortunately not been included. It would have been very nice to have also those results, as they probably would have given this paper much more substance. When making new releases, the teams would have been forced to do more — and perhaps very tedious — merges between the current ”main” branch and the release branch.

I've discussed branches with some of the coaches and they all feel that branches would be useful and something that should be included. Most coaches felt that although they hadn't used branches for the releases their development had never really stalled. But that's most likely due to the small size of this project, and that the size and amount of stories were quite well adjusted to take one iteration. One coach especially mentioned that it would have been nice to be able to use branches for the spikes, since they had had some difficulties keeping track of the spikes.

Our use of branches has been very straightforward and easy all the way. It is only when it is time to merge that it gets a little complicated, but the conclusions are that for every hour of working with a merge, the team has won a couple of hours of development thanks to the lack of stalls.

Questionnaires were handed out to the members of the two teams. Below is summary of the answers and comments.

1) Which release worked best according to you, the one with or the one without branches?
The students who had made one release with and one without all answered that the one with branches were better (and easier to make). Some comments were:
Branches are good and simplifies the work.
There is no commit-deadline.

2) Was it hard/difficult to use branches? Answer in a scale of 1-5, where 1 is not hard at all and 5 is very hard.
One student gave the mark 3, all the others gave it ones or twos. Some comments were:
"Eclipse handled all the difficult parts."
"Pretty easy once you've learned where to click".

3) If you received education: Was the provided time enough? Did the material work; this paper respectively the CVS-guide?
All of the students felt that the provided time was enough to learn and to understand. Some comments were:
There was enough time. It just requires that one sits down and "plays around" with it on ones own.
Totally adequate. The material was good.
The training time was enough to give the theoretical understanding but one needs to use it at least one time in reality to really being able to se the use.
. . . there was always the possibility of asking the coach for help.
. . . perhaps there should be more on branches during the first part of the course.

4) How hard and extensive was the merge between the branches? Did you experience that the (hopefully) less painful release using branches was worth the harder merge or would you have preferred not using branches? Answer in a scale of 1-5, where 1 is not hard at all and 5 is very hard.
Most of the students hadn't actually been involved in a merge, but they anticipated that it would be better to using branches. The marks were all in the lower part of the scale. Some comments were:
A merge is always painful — branches or not.
Branches provide a feeling of security.
We were lucky during our merge.

5) Do you think that all the project teams next year should receive education in how to use branches?
On this question all of the students answered a ringing yes.

6) Did you use branches to something else besides releases? If yes, when

and why?
No student had used branches to accomplish something else.

The results of the questionnaire are almost completely unambiguous. The students feel that branches are something useful — and not that hard to learn and to master. Since two hours of education were obviously enough and since all students thought that it should be a part of the course, it would probably be a very good idea to include some education on branches in the course next year. An extra lab exercise on branches would also lead to that the students knowledge of Eclipse would be improved. An earlier knowledge of branches would most likely result in the students using branches for other things that just releases.

To summarize the answers to the two main questions:
It is worth the two extra hours of education and an Extreme Programming project benefits from the use of more extensive CM, as in branches. The gain using branches is larger than the pain it takes to learn how to use them (and to master the merge conflicts). This is clear even before release 2, and since we only need education once — and learn more and more while doing — we can see that the gain increases along the way.

It would have been interesting to use branches more — mainly during a longer period of time and perhaps in more groups — to get further experiences and to be able to compare the (two) groups more. Although I didn't get as much information as I had hoped, I still feel that it's safe to say that branches are something useful.

Acknowledgements

I would like to thank Lars Bendix for his support to this paper, and the developers and coaches in the teams that took part in the experiment.

The pictures in this article is from [ABC98].

References

[ABC98] B. Appleton, S. Berczuk, R. Cabrera, and R. Orenstein, "Streamed Lines: Branching Patterns for Parallel Software Development", PLoP '98 conference.

[ABE03] U. Asklund, L. Bendix, and T. Ekman, "Configuration Management for eXtreme Programming", Department of Computer Science, Lund Institute of Technology, Sweden.

[Babich86] W.A Babich, "Software Configuration Management: Coordination for Team Productivity", Addison-Wesley Publishing Company, 1986.

[Beck99] K. Beck, "Extreme Programming Explained: Embrace Change", Addison-Wesley Publishing Company,2000.

[Eclipse] The Eclipse Java Development Tools subproject, http://www.eclipse.org.

[Eclipse-branching] Branching with Eclipse and CVS, http://www.eclipse.org/articles/Article-CVS-branching/eclipse_branch.html

[JAH01] R. Jeffries, A. Anderson, and C.Hendricksson, "Extreme Programming Installed", Addison-Wesley Publishing Company, 2001.