Konrad Lukasik 13 May 2014

Branching and Merging: Ten Pretty-Good Practices

In the course of rescuing a development from 'merge misery', it became increasingly apparent that there were a number of practices for managing branches in the Version Control System that would have reduced the pain and effort of the subsequent merge, and made the dream of continuous delivery come closer to reality. From the experience comes some well-tested ways of making branches and merges a stress-free part of application development.

DevOps, Continuous Delivery & Database Lifecycle Management
Automated Deployment

A bit of history…

“Konrad! We have to talk!” – With those words, the Programme Manager rushed into my office room. I don’t like it when Programme Managers move fast: it is usually a bad sign. “We have an issue with the release of the next module of our financial system. Its’ code refers to some core libraries, which we have significantly modified since last release. Now those amendments need to be merged with the live version of module. You will lead a team that will work on the integration task”.

I already knew about this issue and many others as well, because I'd been in the programme for more than nine months. I was keen to do the job as I had heard complaints on many occasions from other architects about the many technical aspects that had not been cared for properly, or endlessly postponed because they'd been valued less than business features. It wasn't divine intervention that led the customer to suddenly realize that those things mattered: It was simply that the deadline for the release was looming closer and the ground was starting to slide from under the customer's feet. I set about building a list of the initial backlog of work in collaboration with my colleagues. It was only the tip of the iceberg, and at that stage we had no idea of the extent of the ice lurking underneath. We had to:

- combine the changes in the database schema and the data integration logic between modules, each of which had evolved this code in different and mutually-exclusive way
- evolve the folder-structure of branches for relatively huge codebase to common format, so that the folders are named consistently and located in the same place of directory tree
- upgrade 3rd party libraries, the CMS framework and our core libraries in all modules to the same version
- write additional integration tests and, where that proves too costly, perform manual regression in order to confirm that the result of our merge is stable
- resolve deployment issues, determine where release steps had not been automated and automate them

Furthermore during the process we discovered that our deployment scripts were unmaintainable. When we tried to fix them, we entered a vicious cycle – one uncovered problem lead to another. At some point builds started to take more than forty minutes aggravating further our situation.

After a few months of unforeseen work at a frantic pace, we successfully delivered the project, largely thanks to a great team. From the history of that struggle emerged the most important lesson – a vast amount of effort and problems can be avoided by having a proper branching strategy and by a continuous integration (CI) process. Here are ten pretty-good practices, which I found useful.

Pretty-good Practices for Branching and Merging

Use the standard Source Control folder-structure correctly

In the development environment I work in, SVN is very popular version control system (VCS). I noticed that several problems were caused by developers not understanding the standard folder-structure and its' purpose. We have three main folders.

- ***Trunk***, the first one, is usually used in two ways: as a read-only reflection of 'live' or as an active development branch.
- ***Tags*** is meant to be read-only, immutable set of shortcuts to shipped releases and it should never be used for coding.
- Finally, ***Branches*** folder should contain: hotfixes, experimental branches and the intermediate results of parallel feature development.

Know the strategy used in your project

Communication will be clearer, and misunderstandings will be less frequent, if the team uses the terminology and jargon correctly and consistently. It is not just the architect who is responsible for promoting this, but the entire delivery team. For example, we have three strategies, which I believe every member of the development team should be aware of.

- "**Branch by release**", sometimes called "staircase model" is the oldest one, where for every planned release we have a separate branch.
- "**Branch by feature**", in turn, has distinct branches for different user stories (US). This occasionally comes in handy, as when we have to do a challenging task such as a cross-cutting upgrade, and we do not want to risk the stability of our code by mixing upgrade check-ins with other stuff.
- Lastly, we have the "**branch by abstraction**" approach, which has

recently had much more attention due to the “continuous delivery” (CD) buzz. This strategy is founded on the concept of a single branch for everything. Features that are new, but not yet finished, can be enabled or disabled by feature-toggles. We can use this device to release different versions of our application from single branch, by changing its configuration. It is very convenient, because it frees us from the merging task, but it is not appropriate for every project.

Try to minimize the number of branches

No matter which strategy is in use, it is always beneficial to minimize the number of branches that are required. Fewer branches means less merges, reduced conflict and diminished misunderstanding within a team, in the same way that, the fewer the layers of the architecture, the lower the cost. In the extreme case, this might be reduced to a single branch with feature toggles. However, on larger code-bases, branching allows changes to be isolated, and so helps teams to avoid disturbing each other. Also, short-lived feature branches can work well, especially when used for risky refactoring or experimentation. It is only when the branches remain in existence for too long that the likelihood of merge-conflicts mount.

Predict release dependencies

When working with a few branches of code that reflect subsequent releases, it is good to be able to predict dependencies and make decisions that will make subsequent merges as easy as possible. There are questions that need to be answered as soon as possible. Which release will go first? What will be the direction of merges and their frequency? On which branch should a particular core component be implemented so that it is most useful and less risky to everybody? If left unanswered, these become so-called “last mile” delivery problems. We have to try to foresee potential issues and plan branches / merges accordingly.

Do merges regularly

Deployments are still sporadic in the enterprise environment, even though start-ups and community projects are revolutionizing delivery techniques. It will take time to change the release strategy for the majority of companies. As a consequence, merges might happen rarely and I've already alluded to the sort of problems that this leads to. Even if the release cycle is extended, it is much better to do the merges regularly; not necessarily automatically, because you then lose the chance to review the code and get familiar with changes of other teams. I recommend weekly merges as being a good frequency to start with. Those merges should be performed by the team that is responsible for the target branch of the merge – they know best when to do it and how to resolve any conflicts.

Think about the impact of the choice of repository

Every VCS has its advantages and drawbacks. This will affect the decisions you take about the merging process, and the Source-Control strategy you choose. SVN or Git manage merging and branching better than TFS. TFS's Auto-merge feature is poorer than in competing products and occasionally the results are so bad that some my colleagues decided to not use this feature at all. Moreover, there is no possibility of synchronizing three branches – the tool will always detect some difference, even though it is only in the metadata. TFS tried to prevent misuse of merging by limiting this process to branches that are in parent-child relationship. Although this initially seems like a safer option, in reality it is restrictive. TFSs' "Baseless merge" is a way to work around this, but it is an alternative that is available only from the command-line and so leaves users with the feeling that you need many "hacks" to work with the tool. My experience comes from work with the 2010 edition, so please bear in mind that this might already be fixed.

Coordinate changes of shared components

Merges might be difficult, taking days and requiring expertise in code. It is occasionally almost impossible, especially for binary files or XML, such as the one used by SSIS's DTSX files. That is why we have to avoid them where possible by planning and coordinating the changes to elements of the application that pose a risk, especially shared elements such as the database or core libraries. Mitigation techniques can include:

- Brain-storming sessions to identify all possible implementation options

- Organizing meetings to discuss the impact of changes
- Creating core teams that are responsible for shared components

At Objectivity, one of teams that faced a particularly complex challenge used a branch graph, hanging on the wall, to discuss and plan future releases. I really liked that practice.

Develop methods to simplify merges

Undoubtedly, there will be times when merging can't be avoided. Certain methods may help to shorten the time before a merge, and minimize the number and scope of code-conflicts. It may help to structure the folders to isolate the code on which teams will work in parallel.

When continuously writing database upgrade scripts, each identified with unique version, you may come across script version conflict during merge. It will happen for example when teams, working independently on separate branches, have written two scripts upgrading to the same DB version, but containing different database changes. With more file conflicts situation is even worse as usually upgrades are not idempotent and they have to be run in planned sequence, giving you no place in between to insert conflicting scripts. To prevent, simply reserve individual version ranges for the teams. Alternatively use separate schemas or databases so that each team owns part of DB. That helps to mitigate conflicts, but is typically not sufficient in enterprise apps requiring data sharing.

When system gets large, it is likely to need to be modularized; and its components, whether shared or not, should be kept in separate branches, compiled into binary form (DLL or NuGet for example) and used in dependent projects as external libraries. This way component's code is held in single place, it does not require merging and the build is faster as the components are compiled just once. If those methods fail or are too costly, the fall-back is described above change coordination.

Promote "CI for everybody"

Every active branch should have a separate CI process to ensure its quality and to make it easy to change and maintain. Without CI and automated tests, the system becomes harder to modify duue to the prolonged feedback loop. If branches survive too long without a merge, then the merge-related bugs are discovered too late, when some of the detail about the changes within the branch has been forgotten. This rule is very intuitive, yet not always followed.

Include merging in estimation

Good agile teams use "definition of done" (DoD) to decide when particular feature can be considered as "done", so that there are fewer misunderstandings within a team or between a team and customer about progress. I found it useful to consider whether merging activity should be included into DoD: Sometimes it is irrelevant or obvious, but at times it counts a lot.