Version Control Systems, Tools and Best Practices: Case Git

Conference Paper · June 2015

CITATION
1

READS
804

4 authors, including:

Marko Mijač
University of Zagreb
11 PUBLICATIONS **33** CITATIONS

SEE PROFILE

Sustavi za verzioniranje, alati i dobra praksa: slučaj Git

Version Control Systems, Tools and Best Practices: Case Git

Igor Tepavac, Krešimir Valjevac, Stefano Kliba, Marko Mijač

SAŽETAK

Sustavi za verzioniranje već dugo predstavljaju sastavni dio razvojnog procesa i neizostavan alat kako individualnim programerima tako i programerskim timovima. Međutim, uporaba takvih sustava i izbor odgovarajućeg načina rada u početku može predstavljati izazov. Programeri i timovi često ne ulože dovoljno vremena da upoznaju mogućnosti ovih sustava, što na kraju rezultira time da ne iskoriste mogućnosti sustava za verzioniranje u punom smislu. U ovom radu pokušati ćemo doprinijeti rješavanju ovog problema identificiranjem i sistematiziranjem korisnih alata i dobrih praksi za korištenje sustava za verzioniranje. Rad će se temeljiti na Git-u, jednom od danas najpopularnijih sustava za verzioniranje.

ABSTRACT

Version Control Systems for quite some time present an integral part of development process and a must have tool for both individual developers and teams as well. However, use of version control systems and choice of proper workflow can at first be challenging. Developers and teams often do not invest enough time to get to know the possibilities of such systems, which results in these systems not being used to their full potential. In this paper we will try to mitigate this problem by identifying and systematizing useful tools and best practices in using version control systems. We will cover the case of Git – one of today's most popular version control system.

1. INTRODUCTION

Version control systems for quite some time present an integral part of development process and a must have tool for both individual developers and teams as well. Today, there is probably no serious developer or a company which doesn't use some sort of version control system on their daily basis. Indeed, these systems are relieving developers of tedious and error prone work of managing source code versions. Indeed, they bring to table a lot of features that now make us wonder how we managed to survive without them.

One of the most popular version control systems, initially developed in 2005 for versioning Linux kernel, is Git. Its popularity is owed to it being free, open-source, fast, highly flexible distributed version control system. Its flexibility is reflected in the fact that it is used by both individual developers for small scale projects and huge companies such as Google, Facebook, Microsoft, Twitter, Eclipse for large projects. However, it also means that in order to use Git, both individuals and teams may have to invest significant time to master its possibilities and to use it to its full potential. Learning a syntax of few commands is not going to guarantee you are doing version control properly. There is also multitude of written and unwritten patterns, anti-patterns, do's and don'ts, best practices, workflows, recommendations etc.

In this paper we will try to mitigate this problem by identifying some useful tools and some best practices in using Git. All this practices should be considered as rules of thumb, and applied if make sense to your particular environment and situation. Second section introduces Git version control system and its basic concepts. Then in third section we list best practices in using Git. In section four several useful tools and services for Git are presented, including client applications, service providers and other utilities. Finally in section five we discussed our findings and presented conclusion remarks.

2. GIT VERSION CONTROL SYSTEM

Git version control system allows us to easily manage changes and versions of different files. We can Git as a special database or repository containing snapshots (versions) of our files taken at different point in time. Git and other version control systems are especially useful in software development because most files we want to keep track of are source code files. Since source code files are basically textual files, version control systems can understand changes happening to particular file throughout various versions. This allows you to rather easily roll back to any recorded point in your project history, or only inspect what the project or any file looked like at some point. Although not its primary goal, this makes version control systems a great backup tool. Version control system is also a great way to track contributions of individual team members, it makes it easier to find or at least narrow down bug sources, and it promotes code review and team member communication.

One of the most important features version control systems offer is collaboration between team members. This is usually done by setting up shared central repository, by which team members share and synchronize their work. Also, in a more distributed manner, version control systems such as Git allow team members to exchange their work directly with each other, without setting a dedicated central repository. However, even in distributed workflows there are usually special team members which are proclaimed project maintainers/managers, and their repository represent official codebase. This is often done in very large teams or open source projects in order to increase security and retain control over project.

Git has quite a number of different concepts and commands. Some of these can even be used for different purposes, so using Git can sometimes be puzzling. However, most of your daily work with Git will revolve around only a few commands.

In Git every team member besides working directory (working copy) has his own local repository to which he commits changes from working directory. Local repository allows member to privately work without disturbing other team members, or to be disturb by others. Of course, since team members do work on the same project, eventually they need to share and synchronize their work. If this collaboration is done through shared central repository, each team member must pull other member's changes from central repository and, when ready, push his own changes to central repository. Following picture describes basic flow between user's working directory, local repository and central repository.

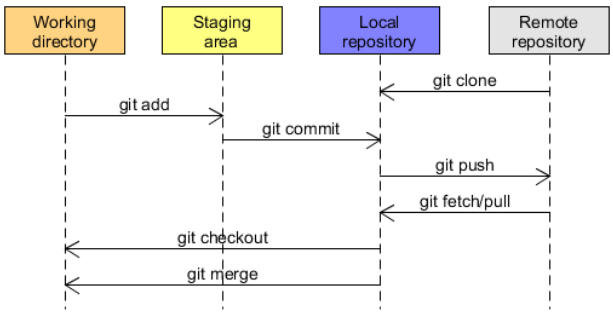

Figure 1 Basic flow in Git

There are two scenarios how we can obtain local Git repositories: we can create new local repository using *git init* command, or we can join existing project and fetch this project's repository from central server using command *git clone*. At his moment our working directory's state corresponds to the state of project in local repository.

After we make some changes in Working directory and want to save them as a new version (snapshot) of our project, we prepare these changes by *adding* them to *Staging area*, and *commit* them to Local repository. In order to make these changes available to other team member, we *push* them to Central repository. If other team members have done the same, we can *fetch* or *pull* their contribution from Central repository to our local repository.

3. BEST PRACTICES

3.1. Common Git workflows

One of the best features of Git is ability to use it in a way it suits your team or company best. Its flexibility allows one to define and use various different workflows, so it can be hard to decide which is the most appropriate one. Before inventing your own workflow there are some common workflows that we think you should consider, that can at least serve as a starting point for your own customized workflow.

A team's decision on appropriate Git workflow can depend on several factors: e.g. size and experience of team, location of team members, type of project, followed development process and methodology,

implementation technology etc. For example, a team with a lot of experience in using SVN will be more comfortable using some of the centralized workflows. For an open source project in which practically anyone can participate a distributed workflows would suit better, so a more restrictive policy for changing official codebase can be employed. Teams which follow prescribed development proces with strongly emphasized phases, can profit by specifying a workflow which mirrors their development process.

Centralized workflow

Centralized workflow assumes there is one central repository which serves as a collaboration medium, and possibly multiple local user repositories. Members commit regularly in their local repository, and when they want to synronize with others they pull other member's work from central repository and push their own work. In centralized workflow this is usually the only way to collaborate. Central repository is configured as „bare“ repository, which means that unlike members' local repositories, it doesn't have a working copy. This workflow employs very simple branching strategy. It requires only one branch (master) to exist, so members commit their work directly to master branch.

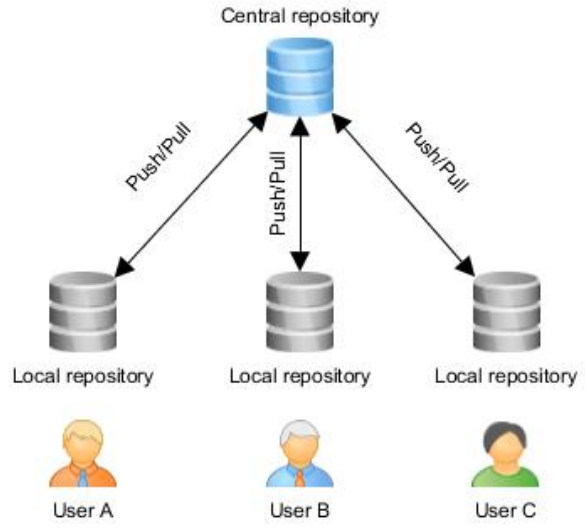

Figure 2 Centralized workflow

Feature branch workflow

Feature Branch workflow is based on Centralized workflow so members continue to collaborate through central repository. However, the difference is in branching strategy. In this workflow, instead of committing directly to master branch, members create new branch for each new feature they start to work on, hence the name. This means that the main codebase in master branch will remain clean, and will get new features when they are fully completed by merging their branches.

In this workflow all development happens in feature branches. They are usually pushed to central repository in order to back up one's work, and to collaborate and discuss with other members. Ofcourse, feature branch can also exist only locally. If we want to discuss changes made in a branch, or we need an approval to merge new feature to official codebase, we can utilize mechanism called pull requests. Such workflow is used even in Github development.

Gitflow workflow

GitFlow workflow continues to use central repository as a primary means for collaboration, and builds upon Centralized and Feature Branch workflow. However, besides master branch, it introduces one additional long-running branch – *develop* branch. Here, the master branch contains only official codebase, while the develop branch is used to branch off and integrate the new features in. No feature branch is merged directly to master branch. Rather, when completed, the feature branch is merged into develop branch, and then develop branch can bi merged into master branch.

Forking workflow (Integration-manager workflow)

Forking workflow is more of distributed nature. It does not prescribe setting up one central repository to which members individually push their changes from their local repositories. Rather, every member has two repositories, one local (private) and one server-side (public) repository. Members make their changes in their local repositories, and when they want to make these changes available to others they push it to their public repository. Only the owner of public repository can push to that repository, others can only pull.

In order to collaborate, one of the members is proclaimed as project maintainer or a manager, and his public repository as an official codebase (canonical repository). Note however, that any

member can be project maintainer and his repository official codebase. It is only a matter of convention who will have that role.

Since official codebase can only be modified by owner (project maintainer), contributions of other members are integrated using pull-requests. E.g. when one of the members finishes new feature, he makes it available by pushing it to his public repository. Then by issuing pull request, member notifies project maintainer that new feature is ready to be integrated to official codebase. The project maintainer pulls new feature from member's public repository into his local repository. After the project maintainer makes sure the new feature is correct, he can push it and make it the part of official codebase repository. Since this workflow introduces higher level of security and control over official codebase, it is often used in open-source projects.

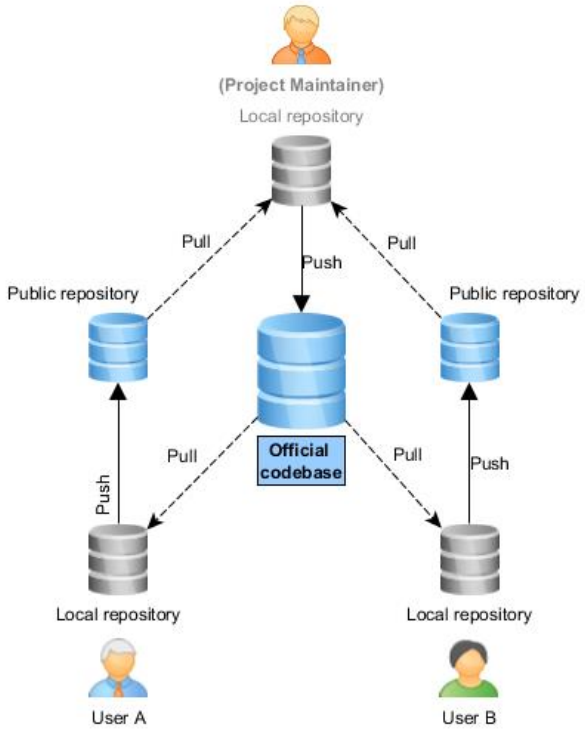

Figure 3 Distributed workflow

Dictator and Lieutenants workflow

Some projects can be so large and complex, that one project maintainer cannot be in charge of managing complete codebase. In such situations Dictator and Lieutenants workflow can be applied. Here, multiple maintainers (called Lieutanants) are assigned with some part of codebase for which they handle pull requests. Additionally all lieutenants answer to member with Dictator role, who only has access to official codebase.The example of using this workflow can be found in linux kernel development.

Development process workflow

Some teams set up their Git workflow in such way it mirrors and simulates their development process. Such workflows usually contain long-running branches which depict important phases in development process, e.g. development, testing, quality assurance, staging, production, release etc. This perhaps forces or makes easier for team members to follow prescribed methodology, and gives each role (developers, testers, integrators, stakeholders) a place to do their job, before moving on to next phase. However, such approach can besides complexity, introduce additional levels of indirection and burden to development process.

3.2. Using branches

Coding in branches is a simple practice that keeps you and your work more organized. Branches let you easily maintain your “in-progress” work separately from your completed, tested, and stable code. This is an effective way to collaborate with others, it will also allow you to automate the deployment of updates and fixes to your servers. There are two cases of coding: you’re either building new features or fixing bugs in an existing codebase.

We could imagine you launched a big Feature X. Things are working nicely together at first and you decide that everything is alright slowly moving on to your next item of business, Feature Y. You start coding and completing the Feature Y, but somewhere along the way you realize that there is a problem with Feature X and its fixing is urgent. If you were working in your default working branch of your repository, you will need to figure out how to save the work you've done so far on Feature Y, rollback the repository to the state where it was when you deployed Feature X, make your fix to the problem we mentioned earlier, do your fix and then re-introduce your work from Feature Y. This is a messy approach and you will most definitely lose some of your work. Instead, what you could do is make a feature or bug-fix branches and let the VCS do the hard work for you. Once you've finished the deployment of Feature X, you could make a branch

that is made for sole purpose of deployment of Feature Y. This allows you to work on a separate history in your features without code overwriting one another. Once you're completely sure that Feature Y is ready and working and could be released, you merge the branch for Feature Y with the working branch of Feature X. This of course means you can switch and work on one of the branches whenever you want and create more branches from any point in time, perchance a bug-fix? Making a branch for every small bug-fix might be unnecessary but following this pattern of work you could avoid making a big mistake of not making a branch for big bug-fixes, potentially leaving your working branch in a messy state.

Some of the most important practices to follow using branches that represent features or bug fixes:

- Try to avoid committing unfinished work to your repository's default working branch.
- Create a branch any time you begin non-trivial work including features and complex bug-fixes.
- Don't forget to delete feature branches once they were merged into stable branch. This will keep your repository clean.

Another reason to use version control is so that you can use your repository as the source to deploy code to your servers. Much like feature and bug-fix branches, environment branches make it easy for you to separate your in-progress code from your stable code. Using environment branches and deploying from them means you will always know exactly what code is running on your servers in each of your environments.

We've mentioned "default working branch" – but you can also think of this as your development environment branch. It's a good idea to keep this branch clean – this is easily done by using feature and bug-fix branches and only merging them back to your development branch once they are tested. In other words, at any point in time your development branch should contain only stable code, as stated above several times.

When your development environment has been updated with features and bug-fixes that are tested, you can use your VCS to do a diff between your stable (master) branch and staging branch to see what would be deployed that's not currently on staging. This is a great opportunity to look for low quality or incomplete code, debug code, and other development leftovers that shouldn't be deployed. This diff can also be helpful in writing release notes. In order to keep your environment branches in sync with the environments, it's a best practice to only execute a merge into an environment branch at the time you intend to deploy. If you complete a merge without deploying, your environment branch will be out of sync with your actual production environment.

With environment branches, you never want to commit changes directly to the branch. By only merging code from your stable branch into staging or production branches, you ensure that changes are flowing in one direction: from feature and bug-fix branches to stable and staging environments, and from stable to production. This keeps your history clean, and again, lets you be confident about knowing what code is running in which environments.

Some of best practices in environment branches are following:

- Use your repository's default working branch as your "stable" branch.
- Create a branch for each environment, including staging and production.
- Never merge into an environment branch unless you are ready to deploy to that environment.
- Perform a diff between branches before merging—this can help prevent merging something that wasn't intended, and can also help with writing release notes.
- Merges should only flow in one direction: first from feature to staging for testing; then from feature to stable once tested; then from stable to production to ship.

3.3. Committing changes

One should commit his work often. This can prevent unsaved work being lost. Comments to your commits should be meaningful and self-explanatory, because in the end, these comments represent a handbook to your project's making process. Also, commit should be atomic, meaning they contain changes related to one particular issue, task or functionality.

Proper commits ease things up for you, your team members and others, by requiring less time to spend

figuring out what actually happend in particular project. If a bug is found in the later stages of the project and it’s concluded on what commit this bug was made, good comments make things easier to understand what the purpose of that commit was. In order to isolate exact revision which introduced a bug, we can use *bisect* command.

Good commenting does not only help your team, but also helps you. If you have to rework some part of the project that you worked on months ago, good comments will save you time spent figuring out what you intended to do back then.

Commit your work often and give some meaning to your comments. This does not take much time, but helps you and your colleagues and can save your team a lot of time later on.

3.4. Resolving conflicts

Although conflicts are not something to be feared (at least in Git), resolving conflicts can require careful analysis and discussion with other team members, which in the end can be quite time consuming. So one of the strategies in dealing with conflicts is to try to actually avoid conflicts. This means that one should concentrate on much more frequent and smaller commits in order to avoid merging conflicts. Of course, that merely depends on the situation and thus we will share a few of the best gathered practices tied with solving conflicts – not counting the one mentioned in the beginning.

One of the most advised methods is to share the work you've done with your teammates. If commits are being done on a more frequent scale – conflicts will be easier to track and in the end, to resolve. Conflicts are tied with the local machine on which the work is being done, so one shouldn't stop sharing on a more frequent basis due to being afraid of breaking the project. So, commit often and resolve conflicts as soon as you see them. Having frequent commits helps with resolving conflicts because in small scale commits there won't be as many conflicts as it would be in large scale commits.

So far we have been talking about avoiding conflicts and not letting them swarm you or the team. But let's say that conflicts just can't be avoided and that you are sure that there will be some. One of the most important things is that you think about the documentation. Not only with fixing conflicts, but with overall process of pushing your code on Git. Once you've successfully resolved a specific conflict, make a note – say what exactly was fixed, how, what was changed, and what the source of the error was.

Let's say that two members of a certain team are editing the same file, the same lines of code, and at the same time. Changes will be made, obviously different, and we will face a conflict. The best possible way to resolve that conflict, without relying that much on the software itself is to see who else has been modifying the file. That way through a discussion and a bit of backtracking – conflicts can be resolved without much trouble coming from pin-pointing the issue.

Git itself already says a lot to us when we step into a conflict. But, what if we wanted to make it even better and a bit easier to read and handle? The answer lies in merging tools. Using the *git config* command, one can set up the merging tool mostly to his liking. So, what do we actually get with a tool like that? Well, it shows us the changes that were made on a particular file, but also provides us with an option to merge one of the shown files into the final version of the file that will be stored into the project tree. Depending on which tool one decides to use, the code that was merged into the final file can be edited even further. After that's done, changing the file is as easy as closing the tool which will then trigger the *git add* command and resolve the conflict.

Considering the above described scenario – let's say that you take one of the given options and whilst doing so you actually end up making the code being worse, slower or overall not as good as it was before. Luckily merging tools also come with the undo option which enables it's user to backtrack to the previous state of the affected file.

Lastly – we want to note that Git is extremely helpful and offers us a lot of options with solving conflicts and editing the code itself. It helps no matter if we are working in large teams and on large, medium or actually small projects. However we would like to stress that benefits of using Git are limited when combined with bad team coordination and overall bad project management. What does that actually mean? Well, it means that git as good as the team which uses the software is.

4. TOOLS

4.1. Service providers

For users to collaborate, depending on workflow, Git assumes either setting up a public central repository or a public repository for each user. There is a number of services which offer hosting and managing these public repositories. Most popular ones are listed in Table 3.

4.2. Client applications

Command line interface is primary way of interacting with Git, allowing access to all Git's commands and options. However, many users choose to work with Git through GUI client applications, which may provide better user experience and easier access to some functionalities (such as browsing history, merging, resolving conflicts etc). Table 1 presents some of the most popupar GUI standalone and IDE integrated client applications:

4.3. Other utilities

Table 2 presents several Git standalone and integrated utilities for Git such as: merging tools, scripting tools, user statistics generators etc.

Name	Platform	Cost	Type	Popularity
GitHub for Mac	OS X	Free	Interface tool Client	*****
GitHub for Windows	Win	Free	Interface tool Client	*****
Gitbox	OS X	Free / 14.99$	Interface tool Client	***
GitX-dev	OS X	Free	Interface tool Client	****
Git Extensions	Win	Free	Windows explorer integration	****
Tower	OS X	Trial / 59$	Git Client	*****
SourceTree	OS X, Win	Free	Interface tool Client	***
git-cola	All	Free	Interface tool Client	**
SmartGit	All	Free (non commercial) / 79$	Interface tool Client	****
GitEye	All	Free	Integrated interface tool	***
gitg	Linux	Free	Gnome Shell Integration	**
TortoiseGit	Win	Free	Windows Shell Extension	****
IDEs supporting Git	Microsfot Visual studio, Android studio, Eclipse, Netbeans, IntelliJ, Ninja-IDE			

Table 1 Git client applications

Name	Type	Platform	Cost
Meldl	Merge tool	All	Free
JavaGit	Library	All	Free
Diffmerge	Merge tool	All	Free
Gitolite	Authorization Layer	All	Free
Git Hooks	Scripting tool	All	Free
Git merge tool	Merge tool	All	Free
Gitinspector	User statistics	All	Free
Diffuse	Text file comparison	Win	Free
EGit	Team provider interface	All	Free
Flashbake	Commit message generator	Linux, OS X	Free

Table 2 Other useful tools for Git

Provider	Free Plan			Pricing Plans			Code Review	Time Tracking
	Private Repos	Public Repos	Users	Price ($/month)	Private Repos	Users		
Assembla	1	1	Unlimited	$24 - $199	Unlimited	12 - 100	No	Yes
Bitbucket	Unlimited	Unlimited	5	$10 - $200	Unlimited	10 - Unlimited	Yes	Yes
CloudForge	Unlimited (trial)	Unlimited (trial)	Unlimited (trial)	$2 per-user - $10 per-user	Unlimited	Unlimited	No	Yes
Codebase	1	1	2	$8 - $65	3 - 40	10 - Unlimited	No	Yes
GitEnterprise	Unlimited	Unlimited	10	$30 - $10 per-user/year	Unlimited	25 - Unlimited	No	Yes
GitHub	None	Unlimited	Unlimited	$7 - $200	5 - 125	Unlimited	Yes	Yes
GitLab	None	None	None	$20 - $249	Unlimited	20 - 100	Yes	Yes
Unfuddle	Unlimited (trial)	Unlimited (trial)	10 (trial)	$19 - $249	Unlimited	10 - 50	No	Yes
Visual Studio Online	Unlimited	Unlimited	5	$20 per-user - $60 per-user	Unlimited	Unlimited	Yes	Yes

Table 3 Popular service providers

5. CONCLUSION

Git's flexibility allows vast number of different workflows and ways to use it. You can customize it to suit your needs. However this flexibility and customizability comes with price in a form of Git's complexity. Indeed, in order to explore Git's full potential one needs to invest significant time and effort. Unfortunately, a large number of developers pressed with their everyday activities and project deadlines do not have enough time to set aside. Nevertheless, we strongly believe that no serious individual developer and especially developer teams should perform their every day jobs without version control system such as Git. For this reason, in this paper, we reported several best practices and useful tools which provide help in using Git.

One of the most important things, which determins overall strategy in using Git, is chosen workflow. We listed several common workflows which can be applied as they are, or they can provide a solid base for developing customized workflows. Also, we indicated a few recommended practices in using branches, which are one of the most advocated Git's features and inseparable part of Git workflows.

In order to collaborate with others and contribute to a project, user wraps his work in a form of commits. Because commits represent a fundamental part of versioning process and influence the quality of this process, we provided a several useful guidelines.

If two team members simultaneously make changes in the same part of the same file, a conflict will occur. In Git, conflicts are nothing to be afraid off, and are quite common. However, if they are too frequent, one should question wheather project tasks are properly distributed to team members. Nevertheless, we bring few recommendation when dealing with conflicts.

Quite a number of tools and services exist which can help you in using Git. They range from GUI client applications which tend to replace Git command line, to various utility applications for Git statistics, scripting, merging and conflict resolution. Also, since a centralized Git workflows are quite popular, we listed several services which provide Git hosting.

During the writting of this paper a few interesting topics for further exploration and research came up:

- Further systematization of best practices in using version control systems,
- Formalization of these practices in a form of patterns,
- Impact of various factors on choosing proper version control system workflow,
- Supporting various software development methodologies with version control system.

REFERENCES

[1] S. Chacon and B. Straub, *Pro Git: [everything you need to know about Git]*, 2. ed. New York, NY: Apress, 2014.
[2] R. Hodson, “Ry’s Git Tutorial,” *RyPress*, Dec-2012. [Online]. Available: http://rypress.com/tutorials/git/index.
[3] “Git Workflows That Work | End Point Blog.” .
[4] T. Günther, *Learn Version Control with Git*. 2015.
[5] “Git Tutorials and Training,” *Atlassian Git Tutorial*, May-2015. [Online]. Available: https://www.atlassian.com/git/tutorials/. [Accessed: 24-May-2015].
[6] “Git Best Practices: Workflow Guidelines,” *Lullabot*. [Online]. Available: https://www.lullabot.com/blog/article/git-best-practices-workflow-guidelines. [Accessed: 24-May-2015].
[7] “A successful Git branching model,” *nvie.com*. [Online]. Available: http://nvie.com/posts/a-successful-git-branching-model/. [Accessed: 24-May-2015].
[8] S. Robertson, “Commit Often, Perfect Later, Publish Once: Git Best Practices,” 2012. [Online]. Available: https://sethrobertson.github.io/GitBestPractices/.
[9] M. Mijač, I. Švogor, and B. Tomaš, *Odabrana poglavlja programskog inženjerstva*. Varaždin: FOI, 2014.
[10] E. Pidoux, *Git Best Practices Guide*. Packt Publishing, 2014.
[11] B. Binkovitz, “Branching, Merging, Commits, and Tagging: Basics and Best Practices.”

INFORMATION ON AUTHORS:

Igor Tepavac
igotepava@foi.hr
Faculty of Organization and Informatics

Igor Tepavac is currently a second year undergraduate student of Information Systems at the Faculty of Organization and Informatics in Varaždin. He is interested in software development for desktop, web and mobile platforms. Also, he is consistently learning about new development technologies which are upgrading his knowledge and experience.

Krešimir Valjevac
kvaljeva@foi.hr
Faculty of Organization and Informatics

Krešimir Valjevac is a third year undergraduate student of Information Systems studying at Faculty of Organization and Informatics. He is consistently learning new technologies and his main point of interests are Mobile, Web and desktop development. He has participated on SEETech hackathon in 2015. where he attained the sixth place. He has been working as a volunteer for the past 7 years for a German company named Gameforge where he developed his soft skills.

Stefano Kliba
skliba@foi.hr
Faculty of Organization and Informatics

Stefano Kliba is a regular third year college student at Faculty of Organization and Informatics, University of Zagreb, in Varaždin. He is working towards my bachelor's degree in Information systems. He's primary interest is in mobile development, particulary Android. He has participated in several competitions, such as Hamag Bicro's Hackathon where he and his team were placed 6th out of 20 competitors, App Start Contest which is currently in progress. Currently working on several mobile development projects. He is also a member of Laboratory for mobile technologies (MT Lab) at Faculty of Organization and Informatics.

Marko Mijač
mmijac@foi.hr

Faculty of Organization and Informatics
Pavlinska 2
42000 Varaždin
tel: +385 42 390 853
fax: +385 42 213 413

Marko Mijač, as of July 2011 works at the Faculty of Organization and Informatics in Varaždin as a project KI Expert 2012, project MEDINFO associate, and as a teaching assistant on courses related to software engineering and information systems. Prior to his current employment he worked as a developer of intranet production planning system at Boxmark Leather d.o.o. He is interested in web, desktop and mobile applications development in various platforms and technologies.