Improvement of a software development workflow

Théo Chamley

Master of Science Thesis
Stockholm, Sweden 2013

TRITA-ICT-EX-2013:84
Civilingenjör thesis

Improvement of a software development workflow

by Théo Chamley – chamley@kth.se
Adviser: Johan Montelius, Associate Professor at the ICT School of KTH – johanmon@kth.se
Supervisor: Niels Giorno, System Engineer at ANSSI – niels.giorno@ssi.gouv.fr

May 9, 2013
Abstract

Organizing the workflow in a software development project is not an easy task. How is the work distributed? How is a coding policy enforced? How can one be sure of the quality of the code produced? Those are non-trivial questions whose answers usually lie in the use of several development tools and practices. Code reviewing and analysis as well as automatic building are, for instance, meant to improve the overall quality of a software project’s code. However, if those tools are not properly configured and if the workflow has not been thoroughly thought, they can be completely counter-productive. Based on the installation of a complete software development chain in a governmental institution (the french ANSSI), this document aims at giving a solution to some of the problems that can arise during such an installation.
## Contents

### Abstract

1 Introduction

1.1 What policies can be used in a code-reviewing tool? ........................................ 1
1.2 Which workflow can a company impose on its developers? ................................. 2
1.3 How can a continuous integration tool be used with a code-reviewing application? ................................................................. 3
1.4 How can a bug tracker be used as the center of the development chain and which one should be chosen? .................................................. 3

2 Background

2.1 Definitions ............................................................................................................. 4
2.2 Choice of the Source Code Management tool ...................................................... 5
2.3 A short introduction to Git ..................................................................................... 5

3 What policies can be used in a code-reviewing tool?

3.1 When is the review done? ....................................................................................... 6
   3.1.1 Code-reviewing before the upload to the central repository ......................... 7
   3.1.2 Changes are in a “waiting” state on the central repository pending approval .......... 8
   3.1.3 Changes are reviewed after being merged into the central repository ............... 10
3.2 A little digression: the Linux kernel or “Darwinian” code reviewing ...................... 10
3.3 Choice of a code-reviewing strategy ..................................................................... 11

4 Which workflow can a company impose on its developers?

4.1 A single central repository vs. a repository per team .......................................... 12
4.2 Git Flow .............................................................................................................. 13
4.3 Our own adaptation of Git flow .......................................................................... 15
4.4 With Gerrit .......................................................................................................... 18
   4.4.1 Fast-forward only ............................................................................. 18
   4.4.2 Merge if necessary ............................................................................ 18
   4.4.3 Always merge ................................................................................... 19
   4.4.4 Cherry-pick ..................................................................................... 19
   4.4.5 Conclusion about Gerrit policies ......................................................... 20

5 How can a continuous integration tool be used with a code-reviewing application?

5.1 The possibilities offered by a continuous integration application ....................... 20
5.2 Which information generated by the continuous integration tool is relevant for a code-reviewing application? ................................................. 21
5.3 How can a continuous integration and a code-reviewing application communicate? ....................................................................................... 22
6 How can a bug tracker be used as the center of the development chain and which one should be chosen? 23

6.1 Purpose and general functionality of a bug tracker . . . . . . . . . . . . . 24

6.2 Information hub . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
   6.2.1 Centralizing information . . . . . . . . . . . . . . . . . . . . . . 25
   6.2.2 Redistributing information . . . . . . . . . . . . . . . . . . . . 25

6.3 Comparison of several solutions . . . . . . . . . . . . . . . . . . . . . . 26
   6.3.1 Bugzilla . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
   6.3.2 Redmine . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
   6.3.3 ChiliProject . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
   6.3.4 Google Code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
   6.3.5 Mantis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
   6.3.6 Trac . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
   6.3.7 Summary of the presented bug tracking solutions . . . . . . . . 28

7 Conclusion 29

References 31

List of Figures 31

List of Tables 32
1 Introduction

Developing softwares and telecom solutions at a professional level requires a team job, interaction between developers and exchange of information. In an organization the main concern of which is the security of information systems, communication of information—regardless of its importance—is always a threat. In those conditions, organizing and improving the developers’ work without compromising the security of the systems is a challenge.

In a typical software development chain, several tools can be used. The most important ones are probably the bug tracker, the source code management (SCM software), the code reviewing application, the continuous integration (CI) application and finally the code analysis tool (the terms will be defined in part 2.1). Because of such a variety of elements, a development chain can be configured in an almost infinite number of ways.

The purpose of this document is to explore different possibilities in the deployment of this development chain, and to find the best configuration possible for the organization in which it is used. In particular, we are going to give a precise answer to the following set of questions about the elements of this chain:

- What policies can be used in a code-reviewing tool? What are their advantages and drawbacks? (see section 1.1)

- Which workflow can a company impose on its developers for their interactions with a central source code repository in order to improve the quality of their work and the integrity of the repository? (see section 1.2)

- How can a continuous integration tool be used in conjunction with a code reviewing application? (see section 1.3)

- A bug tracker can obviously serve as the center of the development chain, that is to be used to distribute work between developers and to track their progress. How can a bug tracker be used for this goal, and which bug tracker can be chosen? (see section 1.4)

The answers to those questions will be based on theoretical studies of policies and tools, as well as the experience of the deployment of a development chain at the ANSSI, a governmental agency. As such, the answers will sometimes be hard to generalize to tools different than the ones used at the ANSSI.

Most of the time, we will prefer give the reader the tools to come up with his own solution, adapted to his needs, than give a definit answer.

1.1 What policies can be used in a code-reviewing tool?

This question deals with the different possible uses of a code-reviewing tool. Code-reviewing is a complex activity, highly influenced by human behavior. When deploying such a tool, you have to make sure that it is easy to use for your developers but that it also integrates well into your workflow and that it is not too much of a constraint for the developers. If code-reviewing is properly done (as it should be), guidelines must be given to the developers for a number of different situations: what happens if a change
is not validated? if there is a conflict with pre-existing code on the server? if some changes have to be reverted after being validated?

To answer this question, we will provide a theoretical study of available code-reviewing tools, their capabilities and an analysis of several code-reviewing strategies. The scope of this question is more particularly aimed at code-review in a corporate environment. In an open-source project, other policies can be implemented but they will not be discussed in this document (with the exception of code-review in the GNU/Linux Kernel, see section 3.2). Because the choice of the code-reviewing application and the source code manager was done before the beginning of this work, only a part of the answer to this question will be based on experience. But human behavior and problems encountered during this work provide a solid basis to extrapolate the results to other tools and policies.

The answer to this question will be a summary of the disadvantages and advantages of three different policies that can be implemented for code-review. Each described policy is as valid as the others and, depending on the context, the reader might choose one or another.

1.2 Which workflow can a company impose on its developers?

The workflow used to develop a software project is maybe the most complex question of the four. The possibilities are not so numerous if a centralized source code manager is used\(^1\) (cf. section 2.2) but in the case of a distributed one, they are almost infinite. More particularly, this question deals with how and when developers create new branches, commits and when they share their work with others. A distributed source code manager can be used in a very disorganized way, but in a company, it is useful to be able to search efficiently through the history of a project. A good workflow helps the team to create a readable history in which commits associated to a given bug or feature can easily be found. It also helps to minimize conflicts and the time spent to resolve them.

To answer this question, we will present two things: the possible organization of the code repositories, their advantages and drawbacks, and the workflow that has been developed for the ANSSI’s needs. It is based on a pre-existing workflow which has been adapted to this situation. The development of this workflow has been a major part of this work and its structure has been refined based on the feedback from the developers and managers of the ANSSI.

We will limit the answer to this question in two aspects: the workflow presented is aimed at a corporate environment and may not be applicable to an open source project where a more disorganized workflow may be necessary. The second limitation is that we will assume that a distributed source code manager is used, the case of a centralized one being quite uninteresting.

This question will show that the choice of the organization of the code repositories is almost entirely dependent on the size of the development team. The workflow presented will only be an example of adaptation of another model. It will certainly need to be adapted to suit your needs.

\(^1\)First you make some changes, then you push them to the server. You almost never create a branch and even less often merge them.
1.3 How can a continuous integration tool be used with a code-reviewing application?

A continuous integration tool is usually used this way: as soon as a change is detected on a given repository (usually the central one), the continuous integration tool fetches the last version of the code and performs a series of automated actions on this new version of the code. The actions performed vary according to the setup but usually the two main ones are compiling the project and running some automated tests on the result.

This is a very powerful tool that allows the user to automatize a lot of different things. In our setup, we use a special tool for code-review which can be a tedious activity. Is there any way to automatize at least a part of this activity? How can we minimize the work of the developers by maximizing the work performed by the continuous integration tool?

The answer to this question will be based on the experience at the ANSSI. First, we will examine what can be done with a continuous integration tool. Then, given the huge amount of information generated, we will sort this information and determine which part of it is actually useful. Finally, we will study the links that can be made between a continuous integration tool and a code-reviewing application.

This work is depends on the continuous integration tool used but even more on the code-reviewing application which is a more complex tool. As a consequence, most of what will be presented is based on the experience of the ANSSI where Gerrit is used as the code-reviewing application. The particularities of this application may render difficult the generalization of this work to other tools.

1.4 How can a bug tracker be used as the center of the development chain and which one should be chosen?

One of the problem in a software development chain is that it is, as its name indicate, a chain. There are a lot of tools in this chain and they generate a large amount of information that they each keep to themselves. As a consequence, the developers and managers have to navigate through the multitude of tools to find the information they are looking for. An obvious way of solving this problem is to centralize all the information about a project in a place that will be easily accessed and searched by the developers and the managers.

A bug tracker is a tool that can achieve that. Indeed, each feature of a software project and each problem the software has or had is referenced in a bug tracker along with every piece of information relevant to this feature or issue (code changes, comments from the developer or the user...). This characteristic of the bug trackers makes them the perfect candidate for the role of the center of the development chain. Most bug trackers also include other features that can be useful for the team.

We will therefore explain how this centralization is possible, what information should be stored on the bug tracker and how it should be redistributed to the users. As the bug tracker is the center of the development chain, the solution chosen is very important. To help the reader make his own opinion, we will provide a comparison of the main bug tracker solutions that exists to this day.
One limit to this exercise is that all bug trackers do not offer exactly the same functionalities. For this reason, the solutions offered to centralize and redistribute information are quite general and there implementation is not explained because it depends on the bug tracker. Also, if all the bug trackers presented have been tested for this study, most were not as used as Redmine and Mantis.

In the end, we will not give a definite answer on which bug tracker is the best, but we advise the reader to use either Bugzilla or Redmine, according to his needs. We also advise the reader to choose carefully which information he wants the bug tracker to send to the users: too much and they will be overwhelmed, too little and they will miss some important information.

2 Background

2.1 Definitions

In this part, we will define the terms that will be used throughout the document.

Bug tracker. “A bug tracking system is a software application that is designed to help quality assurance and programmers keep track of reported software bugs in their work.” [1] In this study, a bug tracker will always be a web application.

Ticket. “A ticket is an element contained within an issue tracking system which contains information about support interventions made by technical support staff or third parties on behalf of an end-user.” [2] This definition is a little restrictive. In our context, the issue can be reported by anyone (an end-user, developer, tester...). Also, a ticket can be a reference to a feature to be implemented instead of an issue to be corrected.

Source Code Management. Source code management, or revision control “is the management of changes to documents, computer programs, large web sites, and other collections of information.” [3] This is done with a software called source code manager (scm, or Version Control System vcs), which is usually a complex but powerful tool. A variety of source code managers exist and they are divided in two large groups: distributed or centralized. In a centralized source code manager, there is a central repository that contains a “reference version” of the code. All changes must be uploaded to this repository. In a distributed source code manager, every user can download changes from any other user and upload changes to any other user. Usually (but not always), a central repository still used for convenience, but even in this configuration, distributed source code managers are more flexible than centralized ones.

Code Review. “Code review is systematic examination (often known as peer review) of computer source code. It is intended to find and fix mistakes overlooked in the initial development phase, improving both the overall quality of software and the developers’ skill.” [4] Obviously, a code reviewing application is an application used to review code. Such an application is usually linked to the central repository of the source code manager. The review itself can be done at a different moment: before the changes are uploaded to the central repository, after the changes are uploaded or in between (the code is reviewed once it is uploaded to the central repository, but before it is incorporated to the main version of the code).

Continuous integration. “Continuous integration (CI) is the practice of merging all
developers workspaces with a shared mainline several times a day. […] Organizations using Git typically use a build server to implement continuous processes of applying quality control in general – small pieces of effort, applied frequently.” [5] Continuous integration can be used to perform a lot of different tasks. In our context, it will refer (unless otherwise mentioned) to the fact that a software is automatically built each time a developer uploads a change to the central repository.

**Code analysis.** “Static code analysis is the analysis of computer software that is performed without actually executing programs [whereas] analysis performed on executing programs is known as dynamic analysis. In most cases the [static] analysis is performed on some version of the source code and in the other cases some form of the object code.” [6] The code analysis can detect subtle errors that will not prevent the building of the software but can lead to problems during the execution (memory leaks, buffer overflows, security problems, etc.). The use of a code analysis tool is a major part of the quality control process.

### 2.2 Choice of the Source Code Management tool

Much of the work presented here is influenced by the choice of the source code manager tool. This choice was done before the beginning of this work and was not discussed later on. One of the main software projects the ANSSI is currently working on deals with Android, the mobile OS from Google. Android being developed with Git, a distributed source code manager, the choice of this tool almost imposed itself. Looking back, we can see that it is a good thing: Git is one of the best, if not the best, existing source code manager. The main other source code managers are:

- **Concurrent Versions System (cvs)** is a centralized source code manager. It was very popular, but is not actively developed anymore. It is also much slower and much less flexible than Git.

- **Apache Subversion (svn)** is a centralized source code manager, conceived as an evolution of cvs. It is still very popular (probably more than Git) but, like cvs, it is less flexible, powerful and quick than Git.

- **Mercurial (Hg)** is a distributed source code manager that is often compared to Git. It is usually considered easier to learn but less flexible and slower [7].

There is a number of other source code managers but those are the more important ones. The main concurrent of Git is Mercurial. Android alone is a reason important enough to choose the first one.

### 2.3 A short introduction to Git

[Git logo] This document is not aimed at being a Git manual, but a minimum understanding of Git is mandatory for the following parts.

---

2Even though this is not completely true: Hg simply hides more of the advanced features than Git.
Throughout the development of a software, Git constructs a tree of states in which the project has been. This tree is stored locally on each developer’s machine. To create a new state, a developer “commits” a change. Once one or several commits have been made, the local tree is not synchronized with the local trees of other developers and of the central repository. Synchronization can be achieved by either “pushing” the commits to a remote repository or by “pulling” the commits from a remote repository. This action can create a conflict if the remote commits change the same part of the same file than the local commits. Those conflicts usually have to be corrected manually.

Like in every source code manager, a git tree can have several branches. This happens when one commit has not one but two or more children. Each child is the beginning of a new branch, each branch diverging from the others at each commit. The branches can be reunified\(^3\) by an action called *merge*.

Git was conceived to make the branching and merging actions very easy. Those actions are much more complicated in other source code managers.

### 3 What policies can be used in a code-reviewing tool?

A code reviewing application let us, as its name indicates, review code. This process is important because it enables an early detection of mistakes that can lead to several problems: code that does not compile, logical error leading to a bug, unexpected cases...

The code review is always done by another developer than the one who actually wrote the code. In an open source project, it is usually done by one of the main developers and in a proprietary project (inside a company), this is done by another member of the staff.

Code reviewing can be done in a lot of different ways: just by showing the new code to someone, by asking them to get the new code and to review it later... The extreme programming practice takes it a step further by teaming two developers who will work on a single computer, code-reviewing being done instantly by the developer that does not hold the keyboard [8]. This practice is extremely hard to implement: you have to make teams of developers with equal skills, they have to get along, they have to learn to code in pair, etc.

A code-reviewing application allows a much more easy code-reviewing process by providing an easy-to-use interface\(^4\). A good application informs the developers when there are changes to be validated, when their changes have been reviewed, etc.

### 3.1 When is the review done?

Using a code reviewing tool implies almost always to have a central code repository that the tool can access (whether the source code manager tool is centralized or not). From now on, we will assume the existence of a central repository for this reason. There are three moments when code-reviewing can be done:

\(^3\)The tree is not, in this case, a tree in the mathematical sense of the term.

\(^4\)That is, if the application is a good one...
• Before the commits are uploaded to the central repository. This policy will from now on be called “pre-commit” or “pre-upload” review. The “pre-commit” name is the common name for this policy but it can be misleading as the word “commit” refers to the SVN notion of a commit. In the case of Git, a commit is done locally and is reviewed before its upload to the central repository, hence the name “pre-upload”.

• The commits are uploaded to the central repository, but they are placed in a “waiting” state until they are approved. Then, they are merged into the main repository that other developers will use as a reference. For instance, they can be placed on a new branch until they are reviewed. Once approved, this branch is merged into the “real” branch on which the commits are supposed to be.

From now on, this policy will be called “waiting-state” review.

• After being merged into the main repository. This policy will be called “post-commit” or “post-merge”. Like “pre-commit”, “post-commit” is the common name for this policy but “commit” refers to the SVN notion of a commit. In the case of Git, the review is done after the merge of the commit onto the central repository’s main development branch, hence the name “post-merge”.

Let us examine those three options.

3.1.1 Code-reviewing before the upload to the central repository

The pre-upload workflow assumes that your changes are reviewed before they are uploaded to the central repository.

As stated previously, the code reviewing application needs a repository to work with. This repository can logically only be a central one. If the code-review must be done before the upload to the central repository, it cannot be done by a code reviewing application, but it can be done in a number of other ways. Here are a few examples of pre-upload code-reviewing:

• If a decentralized source code manager is used, a developer can ask another to fetch some new code directly from his local work station. If the second developer approves the new code, the first one can push it to wherever new code is supposed to be uploaded.

• New code can be printed (yes, as in “printed on real paper”) and go through what could be called a “manual team code-reviewing session” [9]: the code is read line by line, each line is explained and everybody has to agree on what each line does. This method, while highly time-consuming\(^5\), is perhaps the one that eliminates the most errors in the code.

• As explained previously, extreme programming is a workflow in which developers are paired up (with a single work station for each pair) and code review is done “live” [8].

\(^5\) The author of this study recommends a speed of about 150 lines of code per hour.
• Some code-reviewing applications allow a trick where a patch is uploaded to the application in order to be reviewed, but this patch is not integrated to the main repository, even if this repository is managed by the same application. This case is considered to be a pre-upload code-reviewing workflow because, once reviewed, the change has to be uploaded to and merged onto the central repository by the developer.

While the applications that allow this particular workflow usually combine the two functions, you could easily have two different applications: one is the central repository, the other is the code-reviewing application.

Solutions like Review Board allow such a workflow.

3.1.2 Changes are in a “waiting” state on the central repository pending approval

The waiting-state workflow is a very rare solution and it seems that only Gerrit [10,11] is implementing it. The workflow in this case can be compared to a pre-upload review workflow where a patch is uploaded to the code-reviewing application but, in this case, both applications (central repository and code-reviewing) are logically merged into only one application.

However, unlike in this pre-upload workflow, the patch is here uploaded on a separate branch on the repository. If the change passes the review, this branch can be automatically merged on the development branch on the central repository. If the change does not pass the review, another patch (containing both the first patch and the changes made to it) is uploaded on another branch.

The workflow is explained on figure 1. On figure 1a, The developer makes a commit on his local tree which is, as a consequence, in advance on the central repository (this is indicated by the blue parts of the figure; the origin/develop label references the state of the branch develop on the central repository). Then, the developer can push his HEAD (it indicates his tree current state) onto the central repository. However, he does not update the central repository’s develop branch: he uploads his commit on another special branch in which the commit will be waiting for a code-review.

On figure 1b, the first commit has not passed the review and the developer is asked to rework it. With this intent, he creates a second commit (“parallel” to the first one: they have the same parent) that incorporates both the first commit and the changes demanded by the review. He pushes this second commit on the central repository where it will take the place of the first one.

On figure 1c, the second revision of the commit has passed the review and is merged directly onto the central repository’s develop branch. The developer can update his tree by doing a pull from the central repository (this will update the origin/develop reference).

This workflow has three main advantages. The first one is that it keeps the history tree of the central repository very clean: every change is represented by one and only one commit and not a commit and a second and a third commit that correct some errors on the first one. In this solution, only the last commit is kept: the former versions of this commit are discarded. The second advantage is that once his commit has passed the

---

6The first commit is still accessible on both the developer’s tree and the central repository, but not easily since all references have been switched to the second commit.
Figure 1: Commit waiting for a code review on a central repository.

(a) Initial commit upload

(b) 2nd revision: dotted commits indexed with 2 are the same on as in figure 1a, they are not referenced anymore and cannot be accessed.

(c) The 2nd revision has passed the review, the commit indexed 3 is merged onto the develop branch and the developer can update his tree.

review, the developer does not have anything to do: the commit will be merged directly on the central repository and he does not have to push it again (unlike the case of a pre-upload review where a patch is uploaded to the code-reviewing application). The third advantage is that it is much more time efficient than a pre-upload review, almost as efficient as a post-merge review.

However, this solution has the disadvantage of being harder to learn, because less intuitive than other workflows.
3.1.3 Changes are reviewed after being merged into the central repository

The post-merge workflow assumes that you have uploaded your changes to the main repository.

In this workflow, your changes are uploaded to the main repository in order to be easily retrieved by another developer who will review them. If the reviewer is satisfied with those changes, there is nothing to do because they are already integrated in the repository. However, if the reviewer is not satisfied, you have to upload new changes to correct the ones that have not passed the review.

This workflow may be the most time-efficient because both cases (review passed or not) require few steps to complete. This is without doubt why this workflow seems the most natural however it suffers from a great disadvantage: the main repository is very often in an unwanted state in which the last commit has not passed the review. It also has the consequence to increase greatly the number of commits needed to perform one change: there is the original commit, the commit integrating the changes demanded by the first review, the commit integrating the changes demanded by the second review etc. In the end, you routinely have two, three and even four commits\(^7\) in your central repository for only one change.

As a consequence, your history tree becomes quickly very difficult to read since for a single change you have to look at several commits (possibly separated by commits that concern something else and done by another developer).

Several code-reviewing applications are designed to use this workflow. Can be cited: Review Board (it allows both pre-upload and post-merge review), BarKeep, RhodeCode or even Google Code.

3.2 A little digression: the Linux kernel or “Darwinian” code reviewing

In a few open-source projects, no central repository is used. In such a workflow, it is the decentralized nature of the source code manager that takes its whole meaning.

Perhaps the most obvious example of such a project is the Linux kernel [12]. The kernel developers use Git\(^8\) and only Torvalds’s copy of the repository could be assimilated to a central repository.

But describing Torvalds’s copy of the kernel as the central repository would be misleading. The official versions are indeed tagged on his repository, but it is not a “reference version” of the kernel state at any given moment.

Torvalds accepts new changes to the kernel from only a handful of developers whom he knows and trusts. If one of those developers asks him to fetch a change from his local repository, Torvalds will be inclined to incorporate this change to his repository. If the change concerns a domain that he knows well, he will review it and maybe refuse it. If not, he will be likely to accept it because he trusts the developer. If the change provokes a conflict, he will ask the developer to solve this conflict.

---

\(^7\)Those are numbers that were observed in the installed solution at the ANSSI. Changes often demand several revisions.

\(^8\)Incidentally, Git was developed specially for this purpose, in part by Linus Torvalds himself.
This change may or may not have been written by the developer asking Torvalds to fetch and accept it. It is likely that this change was written by another developer who is himself in the “circle of trust” of this developer. Some developers are specialized in networking programming, other in file-systems, etc. As a consequence, developers will submit changes to Torvalds only if they reviewed and tested them themselves or if the changes come from another trusted developer who is known to have reviewed and tested them.

This totally decentralized workflow is built on trust between developers and allows each change to be reviewed and tested several times before being merged into an official release.

### 3.3 Choice of a code-reviewing strategy

As explained in part 3.1, there are three main code-reviewing strategies: pre-upload, post-merge and waiting-state. The three of them have their advantages and disadvantages that are summarized in table 1.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-upload</td>
<td>Highest rate of error detection</td>
<td>Time consuming</td>
</tr>
<tr>
<td></td>
<td>Clean history tree</td>
<td></td>
</tr>
<tr>
<td>Post-merge</td>
<td>Time efficient</td>
<td>Hard to read history tree</td>
</tr>
<tr>
<td>Waiting-state</td>
<td>Time efficient</td>
<td>Hard to learn</td>
</tr>
<tr>
<td></td>
<td>Clean history tree</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Less developer actions</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Summary of code-reviewing policies

With this information in mind, it is easy to choose a strategy and therefore eliminate tools that do not support this strategy. If you want your code to be completely ridden of errors, opt for a pre-upload strategy. But if you have an automatic code analysis tool, you may want to trade the high rate of error detection for some time efficiency as your code analysis tool will detect some errors by itself. If your team of developers is prone to learn quickly a new way of working, you may want to use a waiting-state strategy as it has a lot of advantages.

### 4 Which workflow can a company impose on its developers?

The answer to this question will concern a certain pre-defined context. First of all, we suppose that a distributed version control system is used (like Git, Mercurial or Bazaar). This is the only prerequisite for part 4.1. For part 4.2 and 4.3, we add the supposition that Git is used. Finally, for part 4.4, we suppose that Gerrit is used as a code-review application. Its use and configuration greatly influence the overall workflow, therefore we will investigate different solutions, all based on the use of Gerrit.
4.1 A single central repository vs. a repository per team

With the assumption that a decentralized version control system is used, three different organizations can be implemented:

- A totally decentralized organization, where every developer can pull changes from every other developer. This type of organization is described in part 3.2 and in reference number 12 [12]. While this can be used in an open source project with no real hierarchy between the developers, no example of this organization could be found in a company. Indeed, the hierarchy is much more important in a company and this fact alone is incompatible with a totally decentralized organization.

It also relies heavily on trust between the developers. In a company—or in a governmental organization like the anssir—that deals almost exclusively with information systems security, the slogan could be “Trust no one!”, which is in conflict with the “philosophy” of this decentralized workflow.

- A semi-decentralized or snowflake organization. This is an interesting workflow for software projects on which a large team of developers work. In this case, the developers are likely to be split up in sub-teams, each of them assigned to a specific part of the project. Around a single central repository, each sub-team has its own central repository.

When a sub-team works on a new feature, it uses its own repository which is unperturbed by the other sub-teams work. Also, a sub-team can make mistakes on its repository (push a commit that breaks the compilation for instance) without affecting the other sub-teams and the project on its whole. Once a feature is implemented and tested by a sub-team, then the changes can be pushed to the main central repository to be pulled later by the other sub-teams on their own repository.

The snowflake organization also allows a primitive access control system9: only users who can login on the sub-teams repository can push changes to the central repository.

- In the case of a small team of developer, it may be cumbersome to setup and maintain a repository for each small sub-team (if there are any sub-teams; if not, the logic of a snowflake organization does not apply). In this case, you may want to adopt a centralized organization where each developer pull from and push to a same central repository. With this organization, you loose most of the interest of a decentralized version control system, but you keep some of its advantages: speed10, flexibility (a developer can do whatever he wants him on his local history tree), ability (if needed) to fetch changes from somewhere else than the central repository...

---

9For instance, in Git, you either give full access or deny all access to a repository to a user.
10A decentralized version control system, even in a centralized organization, connects very rarely to the “server”. Network overhead is one of the main reason why centralized version control systems are slower.
With this organization, you also gain the ability to run an application on the central repository (a code review application for instance). In the case of the snowflake organization, it is much more difficult since you would have to install this application on every one of the sub-teams’ repository (with possibly different configurations; but this would also allow you to tune the application configuration for every sub-team’s needs).

Unlike the snowflake organization, the centralized organization implies that if you have a version control system that does not implement an access control system (like Git), then you have to give full access to the central repository to each developer for him to be able to push changes. However, Gerrit implements some access control lists (ACLs) and therefore solve this problem if you use it.

As presented above, only the snowflake and the centralized organizations are suitable to a company environment. The choice between the two of them depends on two things: the size of the development team, and the ability and willingness to setup and maintain several repositories. The advantages and disadvantages of those two organizations are summarized in table 2.

### 4.2 Git Flow

If you do not have a predefined workflow, working with Git can very quickly become messy: you have branches everywhere\(^{11}\), you do not know why you opened them, you do not remember which one you used to develop a particular aspect of your project... This is why, following a good branching model, a convention for naming the branches and preferably having this Git workflow shared between all the developers is important.

---

\(^{11}\)In this aspect, Git has the opposite problem than most version control system: it is so easy to branch that once you get used to it, you can not stop.
### Table 2: Summary of the possible organizations

<table>
<thead>
<tr>
<th>Organization</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
</table>
| Snowflake    | · Suitable for a large team  
               · Possibility of access control  
               · No interference between teams  
               · Adaptation of the repositories to a sub-team needs | · Not suitable for small teams  
               · Difficult to setup and maintain |
| Centralized  | · Suitable for small teams  
               · Easy to setup and maintain | · Access control depends on version control system  
               · Can get messy if large team |

Git has been around for a while now but the first real successful and widely spread branching model dates back to only 2010. It is called the Driessen model [13], is implemented in the form of Git Flow and it makes it easy to work with Git while keeping a simple and logical history tree.

Using Git Flow, you will have different branches and each one of them will have a specific purpose (a diagram from the official website can be viewed in figure 4):

- **master** branch: this branch always contains a production-ready code source. It only contains tagged versions of the project. No "development" commit should be made on this branch.

- **develop** branch: this is the main development branch that receives features and bug fixes. There is usually no "development" commit either on this branch: other branches are merged onto it. Once this branch is in a production-ready state for the next version, it is merged onto the **master** branch where the new version is tagged.

- **feature** branches: those branches start from the **develop** branch. For each new feature added to the project, there is a corresponding **feature** branch where it is developed. Once the feature implemented, the branch is merged back onto the **develop** branch.

- **hotfix** branches: when a critical bug is detected on the production code, a **hotfix** branch is created from the **master** branch (where the production code can be found) to fix this bug. Once the bug is fixed, the hotfix branch is merged back onto the master branch where a new minor version is tagged. The hotfix branch is also merged onto the **develop** branch for the other developers to have the bug fix incorporated in their work version.

- **release** branches: these branches are used just before the release of a new version of the project. They allow some "last-minute" work on the code: minor bug fixes, incorporating hard-coded version number or build date etc. Once the release is ready, the branch is merged onto the **master** branch and then tagged as the new version. If needed, it can also be merged onto the **develop** branch to report some minor bug fixes.
Git Flow makes an intensive use of branches and merges. Merges should be made as "non fast-forward" in order to keep all the branches in the history tree. Indeed, when possible and by default, Git makes fast-forward merges: if B is branched from A and is merged back onto A without any other commits being made on A in between, then the head of A is just updated to point to the same commit as the head of B. See figure 3 for a visual explanation.

However, Git Flow is not without flaws. For instance, the main reason why Git Flow could not be implemented "as is" at the ANSSI is that for some projects, several major versions are being maintained at the same time. In Git Flow, version numbers are always increasing, but it is quite often that you have two different main versions of the same project and you must be able to tag version 2.0.1 and then make some corrections on version 1.3.2 and tag a new version 1.3.3. This is useful when some users do not want or can not update to the newest version (2.0 here) but you still want to make some bug fixes or security updates.

In Git Flow, versions are tagged only on the master branch and therefore, a version B chronologically tagged after a version A has to be an evolution of A.

It is worth noting that a project called Hg-Flow [14] has adapted the principles of Git Flow to Mercurial. It uses the same branching model as the one described previously.

4.3 Our own adaptation of Git flow

As implied previously, we chose to not implement Git Flow per se at the ANSSI. The workflow that was implemented differs from Git Flow in several points. The main reason for this is that several main versions of a software project are under development at the same time. This has two consequences:

- There are several develop branches, one for each version of the project. If needed,
Figure 4: A visualization of the Driessen branching model
some changes made on one of the develop branches can be merged on the others.

- There are several master branches, one for each version.

One other main difference is that we can not use a release branch because of the confidential nature of the work. Last minute changes to be made just before an official compilation are done by the development controller who can include some confidential information in the project (signing binaries with a private key for instance). This information is not shared with the other developers and therefore can not be committed on a branch.

As release branches are not used, we decided to use the name for our "master" branches. For instance, all versions 1.3.x are tagged on the branch release-1.3 whereas all versions 2.0.x are tagged on the branch release-2.0.

Finally, the last difference is that hotfix branches are not merged back directly on the master branch (one of our release branch), but, instead, are merged on a develop branch which is the branch that will be merged back onto the master branch. The reason for this is that all test versions of the software project are compiled from the develop branch and therefore, the bug fix must be on develop before being sent into production in order to be tested. The ANSSI workflow can be visualized on figure 5.

![Diagram of ANSSI's adaptation of Git Flow](image)

Figure 5: ANSSI’s adaptation of Git Flow. Dotted lines indicate omission of commits.

This particular workflow suits the needs of the ANSSI, but may be completely unsuitable for other companies and too complex for some simple software projects. To conclude, Git Flow and the Driessen model are a great way to work but, most of the time, it can not be used "as is". The developers should thoroughly think about their
need, the version-naming system they use and how they will continue to develop and support their software. Once they have all of this information can they begin to examine the version control system workflow they will use. If it is possible, they can use Git Flow but, most probably, they will have to modify it.

4.4 With Gerrit

If you use Gerrit [10,11] for code-review, you have yet another choice to make. As Gerrit puts changes in a waiting state on some custom branches that are managed internally in the application, Gerrit has to merge those changes onto their destination branch once they are accepted.

Four different policies are implemented in Gerrit to this effect. The choice between them depends on the confidence you have in an automated tool to make merges (that can be difficult things to deal with) and final history tree that you want.

4.4.1 Fast-forward only

With the Fast-forward only policy, you do not give permission to Gerrit to perform complicated merges. As the policy’s name indicates, the merge will happen only if it is a fast-forward. Otherwise said, if a commit has been accepted and is about to be merged on its destination branch, the merge will proceed only if no other commits where accepted and merged on the branch between the submission of the first commit and its merge. Figure 6a shows a case where the merge is not going through.

In the case of figure 6a, Gerrit would fail to merge the commit 3 if there was a conflict between commits 2 and 3. Using the Fast-Forward only policy is a way to make sure that Gerrit does not try to merge commit 3. You may want this comportment if you do not trust Gerrit to make merges. It also creates a strictly linear history.

Let us see what would happen if the merge was allowed with the Merge if necessary policy.

4.4.2 Merge if necessary

In the case of figure 6a, if the Merge if necessary policy is implemented, Gerrit would try to merge the commit 3 onto the main branch. If no conflict is detected between commits 2 and 3, we would get the situation described in figure 7.
If the merge triggers a conflict, then the developer is asked to rebase the commit 3 to get the situation described in figure 6b.

This policy implements the comportment of the command `git merge`: if needed a merge commit is created (like in figure 7) but, if possible, a fast-forward is done.

### 4.4.3 Always merge

This policy differs with the previous in the fact that, even if possible, no fast-forward merge is used: a merge commit is always created. This is equivalent to the `git merge --no-ff` command which creates a history tree that is never linear.

The merge of a commit that is been reviewed and accepted is done as in figure 8. In the case of this figure, commit 2 has been reviewed and accepted and commit 3 is the merge commit corresponding to commit 2’s merge onto its destination branch A.

### 4.4.4 Cherry-pick

This policy takes its name from the eponym `git` command. In this case, the commit that is accepted is not merged onto its destination branch but is cherry-picked, that is to say that a copy is created and appended to the head of the branch, regardless of the original commit’s lineage.

Contrary to what is widely believed, Git never moves a commit and never changes its parents. When "moving" a commit, Git is merely creating a new one that contains the same changes but has different parents. That is why when a commit is "moved", its SHA1 identifier changes\(^{12}\). This is the case when a commit is cherry-picked.

---

\(^{12}\)The SHA1 identifier is computed from a number of things, including the parents of the commit.
4.4.5 Conclusion about Gerrit policies

The *Always merge* policy does not seem to have any advantage on the *Merge if necessary* policy: it only creates a non-linear history where the developer would expect a linear one. With the *Merge if necessary* policy, the linear history is created when possible.

The *Cherry-pick* policy can create some confusion for the developers by changing the commits’ identifiers without suppressing the conflict risk.

The *Fast-Forward only* policy ensures that a conflict will never happen on Gerrit and creates a linear history, that is the reason why it is the policy that has been chosen at the ANSSI. However, in a software project that is intensively developed, the case of figure 6a arises quite often. This is problematic because rebasing branches and commits is a non-trivial operation and several developers have had some problems with it, leading to a lot of time lost. This is why I advised for a change in favor of the *Merge if necessary* policy which, it is true, produces a less linear history, but allows a more streamlined workflow.

5 How can a continuous integration tool be used with a code-reviewing application?

In this part, we will study the possible interactions between a continuous-integration tool and a code-reviewing application. Part 5.1 will not be solution-dependent and is an overview of what a continuous integration tool can do. Part 5.2 is an attempt at sorting all the information possibly generated by a continuous integration application in order to identify which kind of information is relevant for code-review, can help the reviewers and the developers. Part 5.3 is an example of how two different parts of the development chain can communicate and how to secure this communication. While this part is solution-dependent and will be based on the solutions chosen at the ANSSI, some generalization will be made.

5.1 The possibilities offered by a continuous integration application

Numerous continuous integration tools exist and they all have the same purpose: allow the automatization of tedious tasks. Most of the time, this means building the software at each new commit for verifying that the new commit does not introduce some big error that prevents building and, sometimes, for always having the latest build.

But a continuous integration application can be used for much more than just building: it can automatically run tests to be sure that each new commit does not break something in the project. If your teams has some good practices, the developers are likely to write some unit and functional tests. Those tests can be run by the continuous integration tool.

In fact, most continuous integration applications function the same way: they watch for some changes in the central repository\(^\text{13}\), once a change is detected a predefined

\(^{13}\)The solution they use to "watch" those changes can vary from a version control system to another and from a continuous integration application to another.
action is executed. The application can have some famous build tools (such as Maven or Ant) pre-configured to simplify the setup but most of them also allow to simply execute a script when a change is detected.

The fact that the continuous integration application uses a script as action to be executed when a change is detected opens a lot of doors: the application can therefore execute anything as long as it is scriptable\(^\text{14}\).

Let’s take an example: you have a software project that is built at each new commit by your continuous integration tool. Until now, the only verification done is that your project can be built with the new commit. This is indeed important, but you can achieve much more. Let’s say you have a code-analysis tool: you can configure your continuous integration application to run this tool and analyze the new code at each commit. If this commit introduces a new defect in the code (memory leak, possible buffer overflow, etc.), then you can easily send an automated e-mail to the author of the commit to have him correct the error. Figure 9 is a schema of how such a setup would work.

On a less theoretical note, it is important to say that most continuous integration solutions allow a master/slave architecture. Most tasks given to the continuous integration application are resource-consuming and even on a powerful server, building (and analyzing) a large software project can take several minutes. If you have a large development team, this can be a problem because it means that there is a large delay between the commit of the developer and the end of the work of the continuous integration application. The master/slave architecture allows you to easily adapt the number of servers dedicated to building the project to your computing needs: need some more power? Just add a slave.

\begin{center}
\begin{tikzpicture}
\node (developer) at (0,0) {developer};
\node (repo) [right of=developer] {central repository};
\node (ci_app) [right of=repo, xshift=1cm] {ci app};
\node (ci_slave) [below of=ci_app, xshift=1cm] {ci slave};
\node (tool) [below of=repo, xshift=1cm] {code-
\node (tool) [below of=repo, xshift=1cm] {code-analysis tool};
\draw [arrow, line width=1pt, rounded corners] (developer) -- node [above] {1: push} (repo);
\draw [arrow, line width=1pt, rounded corners] (repo) -- node [below] {2: detection of change} (ci_app);
\draw [arrow, line width=1pt, rounded corners] (ci_app) -- node [above] {3: order of action} (ci_slave);
\draw [arrow, line width=1pt, rounded corners] (ci_slave) -- node [below] {4: build} (ci_app);
\draw [arrow, line width=1pt, rounded corners] (ci_app) -- node [above] {5: analysis order} (tool);
\draw [arrow, line width=1pt, rounded corners] (tool) -- node [below] {6: analysis results} (ci_app);
\draw [arrow, line width=1pt, rounded corners] (ci_app) -- node [above] {7: build & analysis results} (ci_slave);
\end{tikzpicture}
\end{center}

\begin{center}
Figure 9: Schema of developer/continuous integration tool/code-analysis interaction
\end{center}

5.2 Which information generated by the continuous integration tool is relevant for a code-reviewing application?

For any given change, a code-reviewing application needs some information to sort the change as accepted or not. This information can be given manually by a developer or generated by some automated tools, particularly by a continuous integration tool.

\(^{14}\text{And finding an action that is not scriptable is quite hard...}\)
We can note at this point that in order to make the continuous integration application a part of the code-reviewing process, we need to use either a waiting-state or a post-merge policy as code-reviewing strategy (see part 3.1 for a definition of those terms). Indeed, the continuous integration tool can not have access to the new change during the review process if this process takes place before the upload to the central repository (as it is the case with a pre-upload strategy).

By its nature, a continuous integration application can generate a lot of information (virtually any output of any script). For code-review, only a part of this information is useful. The most obvious piece of information needed is the fact that the compilation is a success or not. If not, code-review can stop here and the change has to be modified. This particular case happens more often than one could think and the use of a continuous integration tool saves a lot of time here because the reviewer does not have to look at a change that introduced a compilation-blocking error.

Compilers can give a lot of different warnings during the building of a software. Those warnings can be interesting for code-review but they can also be overwhelming because, specially on large projects, compilers tend to give a lot of them, so you may not want to fail the review for only one compilation warning. But some rules can be established: failing the code-review if the change added more than \( n \) new warnings, failing the code-review if a given type of warnings is given, etc. Some code-review applications (such as Gerrit) allow to give a "grade" to a change. In this case, warnings can induce a negative, but not blocking, grade to the change. The reviewers, upon seeing this negative grade can then decide if the warnings are important enough to block the change or not.

If you decide to use a code-analysis tool\(^\text{15}\), the same principle can apply. Indeed, such tools also produce warnings differing in their form to the compiler’s warnings, but they are essentially the same thing.

If you use the continuous integration application to run unit or functional tests on your project, the success of those tests should also be taken into account in the code-review process. Tests should never fail: either the change broke something and the change has to be modified or the change is a functionally one and the tests have to be modified. Either way, a test failure is the indication of a problem. As such, a test failure should lead to the refusal of the change on the code-review application. If a grading system exists, a negative but not blocking grade can be a solution but it would still be a paradox considering the purpose of tests.

5.3 How can a continuous integration and a code-reviewing application communicate?

In this part, a recurrent problem is tackled. In order to setup a development chain and to automatize a maximum of things, you need to make the different part of the chain communicate with each other, in a secure way and efficiently.

Fortunately, most of the available tools (and it seems that the trend is stronger in the open-source world than in the proprietary software one) come with an Application Programming Interface (API). An API allows to easily interact with the application behind the API, specially when the "user" is another software program. For instance, instead

\(^\text{15}\)And you should decide such a thing!
of using Twitter’s web application to post a tweet, you program your software to use Twitter’s API. What is true for Twitter is also true for code-reviewing applications, for bug-trackers, for code-analysis tools, etc. The information you get from and give to an API is much more computer-readable than a classic web page: JSON and XML are usually the preferred way to deal with information through an API.

An important part of the API is the authentication part: the API has to be protected at least as much as the "classic" application. Whereas for human users, the preferred authentication method is still the login/password, with APIs you have to opportunity to use much stronger methods as API keys (that are just like passwords, but much longer and with much more entropy).

In our case, an API is clearly the way to go in order to make our two tools (code-review and continuous integration) communicate. With this method, your continuous integration tool can send information to the code-review application directly through the latest’s API.

But let’s examine the case of the ANSSI: Gerrit [10,11] as code-reviewing application and Jenkins [15] as continuous integration solution. The two of them make a powerful pair, thanks to both the Gerrit Trigger plugin [16] of Jenkins and the fact that almost everything can be done on Gerrit through ssh¹⁶. Firstly, contrary to most of the other solutions, Jenkins does not have to keep checking the central repository for new changes¹⁷: Jenkins maintains a constant ssh connection with Gerrit through which Gerrit streams all new events (new commit, new review, etc.). Jenkins just has to sort through those events and decide which one of them trigger an action. We have a totally synchronous behavior which is faster and more efficient than the classical inspection loop.

The authentication of Jenkins in Gerrit is then done with a ssh key and not a password or an API key. ssh keys are usually considered as a very secure way of authenticating a user. The Gerrit Trigger plugin for Jenkins is also useful because it allows Jenkins to fetch changes that are in a waiting-state and not yet merged onto their destination branch. Thanks to this, Jenkins can be part of the code-review even with the "waiting-state" strategy (see part 3.1 for code-review strategies).

6 How can a bug tracker be used as the center of the development chain and which one should be chosen?

The development chain presented so far is quite complex and even qualified people can be lost by the amount of information generated and by its dispersion among the different elements of the chain. This is why it is important to have a single service on which people can find the information they are looking for or, at least, find the location of the wanted information. As you have certainly understood it, a bug tracker can serve as this center of information.

¹⁶Secure SHell: a program that allows to connect remotely and securely to another computer
¹⁷Usually, the continuous integration tool has an infinite loop: "Are there any new changes? No. Are there any new changes? No. Are there any new changes? No. Are there any new changes? Yes. Then I will do something."
Part 6.1 will present the functionality common to every bug tracker solution and define a "general purpose" of a bug tracker. Part 6.2 will explain why a bug tracker's functionality can be an asset when you need a fast and easy way to access information about your software project. (Un)fortunately, there are numerous bug tracker solutions and they can differ substantially one from the other. Being such an important part of the development chain, the choice of the bug tracker is very important. This is why part 6.3 is dedicated to compare the main and most famous bug tracker solutions and assess their strengths and weaknesses.

6.1 Purpose and general functionality of a bug tracker

While its form can vary greatly from one solution to another, the core of a bug tracker is always the same: it enables users to create objects called "tickets" that describe a bug in a project\(^{18}\). Once a ticket is opened and the bug described, users can add information to this ticket according to their role in the project development: add a date before which the bug should be solved, ask for more information on the bug, give a potential solution, etc.

Usually, tickets can be sorted thanks to a number of different filters and are separated by project (a single bug tracker is used for different projects). The tickets also have a status. Examples of tickets' statuses could be:

- **new**: the ticket has just been opened
- **working**: some work is being done to solve the bug described by the ticket
- **waiting validation**: a solution has been proposed, the user who has submitted the ticket must validate the solution
- **closed**: a solution has been proposed and accepted

Those four states are simplistic and you could not use them in a real project because they do not cover all cases. A more realistic approach will be described in part 6.2.

While creating and interacting with tickets is why bug trackers have been developed, all of them offer other possibilities: one common functionality is the wiki. For each project tracked on the bug tracker, there is a small wiki that can be used for a lot of things: best practices, a description of compilation process, setup of a development environment for this particular project, etc.

Another functionality of bug trackers which is very important is the creation of roadmaps. Each ticket is assigned a "target version" of the project. For instance, the software version 2.0.3 will be released only when all tickets the target version of which is 2.0.3 are solved. This enables the development manager to see easily the amount of work left before the release of a particular version. It also proves useful when you want to generate a changelog that describes the modifications added by a new version: the changelog is just a summary of all tickets whose target version is the new version.

---

\(^{18}\)It does not have to be a software project, but it is the most common case and we will assume that we are dealing with software projects in this argumentation
6.2 Information hub

6.2.1 Centralizing information

How all of this can make the bug tracker the perfect center of the development chain? If only bugs were mentioned until now, it is important to note that a ticket, while originally conceived to describe a bug, can also describe a functionality to be implemented in the project. Therefore, you have two big types of tickets\(^\text{19}\): those describing bugs and those describing new features or modification. Using those two types of tickets, you build a history of the project. Each modification (whether a solution to a problem or an evolution) is listed as a ticket and, if used correctly, each piece of information relevant for this ticket is accessible through the ticket (as a ticket attribute or a comment).

If for each code modification in the project you create a ticket before effectively modifying the code, then each modification of the project is listed on the bug tracker. If someone wants to inspect how a given functionality was implemented and the reasons behind the technical choices, he will find all the relevant information on the related ticket.

Most bug tracker solutions offer a powerful API that allows automated tools to post information on tickets. This API is the key to centralize all information on the bug tracker: every piece of information generated during the development process can be posted on the bug tracker. You usually need to ask the developers to include the ticket number in the commit message. If they do this, every tool (continuous integration, code analysis, etc.) can know which ticket is concerned by each commit. They can therefore post their results or links to their results as comments on the ticket.

6.2.2 Redistributing information

As we just have seen, centralizing all the information about a project and making it easily navigable is simple with a bug tracker. That is why the bug tracker is the best place to redistribute information.

The usual way to inform people involved in a project of something happening in the project is the e-mail, sent automatically or not\(^\text{20}\). While convenient (asynchronous, accessible from everywhere, can be sorted and stored, etc.), e-mails tend to have a big disadvantage: people receive so much of them that they lose track of them or take a huge amount of time to go through them which reduces the person’s productivity.

On the one hand, if too many e-mails are sent, people risk to overlook an important one but, on the other hand, if you reduce too much the number of e-mails, the risk is to not send one that could have been important. The problematic is therefore to find the right balance between the two extremes.

A good option to start is to precisely target your automated e-mails to people who really need them. For instance, a development manager may not need to get an e-mail each time there is an update on a ticket, but he needs to know when one is created and when one is closed. Whereas a developer needs only to be informed of updates on the tickets he is assigned: he does not need to receive an e-mail each time an other

\(^{19}\)Technically they are the same, but functionally, they are not.

\(^{20}\)IRC bots are also used sometimes but as they require users to be connected they are not as powerful as e-mails in this use-case.
developer does something.

As the bug tracker centralizes all the information about a project, it is advised to allow only the bug tracker to send e-mails. Indeed, if you want other services to send e-mails, you need to maintain several other mail configuration, you have to parametrize those services to sort through the information in order to send the right e-mails to the right people, etc. Bug trackers allow you to assign roles to users, to assign users to tickets, etc. which facilitate greatly the task of deciding who is to receive which piece of information. On most bug trackers, users can also decide what type of information they receive which allow them to customize the e-mails to their need and their way of working.

As we have seen, bug trackers are indeed the best place to keep track of all the information generated by the development chain and to keep people informed of what is happening in the project without overwhelming them with information.

6.3 Comparison of several solutions

There are numerous bug tracking solutions, and while they are generally designed to do the same thing, the differences between them are important enough to give a great deal of attention to the choice of the tool. Therefore, this part will try to assess the strengths and weaknesses of some of the most used bug trackers.

6.3.1 Bugzilla

Bugzilla may be one of the most famous bug tracking system. As its name indicates it is developed and supported by the Mozilla foundation. It has a lot of different features and you can access information in a lot of different ways (rss, iCalendar, REST API, etc.). It has features that are difficult to find elsewhere (such as the scheduled reports by e-mail).

It is developed in Perl and is one of the few mainstream open source projects that uses Bazaar as version control system. The installation process is cumbersome and, depending on your installation, the upgrade process can be difficult also.

Being one of the most famous bug trackers, Bugzilla has a lot of plugins but not as much as one could think. This is maybe because Bugzilla already has a lot of features, which leads us to its biggest disadvantage: this application is huge. It does a lot of things and most people do not need so much features so if they use Bugzilla they end up running an application that can be too complicated for their need.

6.3.2 Redmine

Redmine is a well known bug tracker and a project management tool. It is based on Ruby on Rails and as such is compatible with a lot of databases (MySQL and PostgreSQL among others). If the installation itself is quite simple provided you have an internet access, the configuration of Redmine can take time because everything can be configured: the ticket statuses, the workflow, who can update tickets at which moment, who can access what, etc. While this has the advantage of being highly customizable, it implies that you have to take some time to think through your workflow and your access control lists.

Redmine also has the advantage of being natively compatible with the most used version control systems (Git, SVN, Mercurial, Bazaar, etc.). Each time a new commit is
detected, if the commit message contains the ticket’s number, the commit is displayed in the ticket and the ticket’s status can be updated automatically.

Redmine being famous and based on Ruby on Rails, it possesses a lot of plugins\textsuperscript{21} that can certainly add the feature you are looking for if you could not find it in the forest of native ones (wiki, GANTT charts, calendars, forums, etc.). Redmine also has a great REST API which is very powerful and simple to use. Finally, the project is well maintained and new versions are released often.

Some small problems are the difficulty of installing and upgrading Redmine in a network not connected to the Internet\textsuperscript{22}.

6.3.3 ChiliProject

ChiliProject is not yet very famous because the project is recent. It is in fact a fork\textsuperscript{23} of Redmine and the two projects shared their base. ChiliProject was created because some of Redmine’s developers thought that the release process and patches were not handled the right way on Redmine.

ChiliProject has a few advantages over Redmine as a shorter release period, having Git as source repository, etc. But on the whole, Redmine is usually still a better choice, in part because its API is more extensive. Also, plugin compatibility is not complete between the two projects and Redmine has a lot more of them than ChiliProject. With the two code sources growing apart with time passing, plugins will be less and less compatible between the two.

6.3.4 Google Code

Google Code is the only hosted service presented here: you cannot install it on your own servers and the projects hosted on Google Code must be open sourced. This solution offers the minimum and can be enough for some projects\textsuperscript{24}. It has a wiki, a download section, integration with Git, SVN and Mercurial... If you want a ready-to-use solution, it may be a good choice.

GitHub and BitBucket, while more restricted on the compatible version control systems are more user-friendly and allow private projects for a fee.

It is to be noted that those services are primarily hosting services and that they offer a bug tracking system as a bonus service.

6.3.5 Mantis

Mantis\textsuperscript{25} is a bug tracker developed in PHP. It has a lot of features but suffers from a few big disadvantages: it is not natively compatible with Git (a plugin exists but such an important feature should not rely on third-party code), it is not ergonomic and the project itself is losing some momentum.

\textsuperscript{21}Ruby on Rails is a famous framework. Its use facilitate the creation of plugins.
\textsuperscript{22}While this is a not usual, it is a very good security measure to take if your development chain does not have to be connected to the Internet.
\textsuperscript{23}A project derived from another.
\textsuperscript{24}Gerrit is tracked on Google Code, but it is developed by Google, so it may not be a good example...
6.3.6 Trac

Track is a Python-based bug tracker and it has been developed for some time now. While it was initially only compatible with svn, it now supports most version control systems. It is still actively developed, a fact that demonstrates the maturity of the project: a lot of web applications lose their developing community after a few years. It has a few helpful plugins but their number is nowhere near Redmine’s.

While no numbers can support this fact, it seems that Trac suffers (or at least used to) from performance issues: pages would take a lot of time to be displayed on the client’s browser.

Finally, the experience you get using the application is quite different from what you could find in other bug tracking solutions. Some people may like it, others may not. But if you are migrating from one bug tracker to another, you may want to choose something that is at least a little similar in its user interface to what you are migrating from to ease the transition process for the users.

6.3.7 Summary of the presented bug tracking solutions

Table 3 is a summary of the advantages and disadvantages of the bug tracking solutions that were presented previously.

<table>
<thead>
<tr>
<th>Application</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugzilla</td>
<td>Lot of features</td>
<td>Maybe too many features?</td>
</tr>
<tr>
<td></td>
<td>Many ways to access information</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A lot of plugins</td>
<td></td>
</tr>
<tr>
<td>Redmine</td>
<td>A lot of plugins</td>
<td>Difficult to install and upgrade if not connected to the Internet.</td>
</tr>
<tr>
<td></td>
<td>Highly customizable</td>
<td>A lot of configuration to do to make it meet your needs.</td>
</tr>
<tr>
<td></td>
<td>A very good API</td>
<td></td>
</tr>
<tr>
<td>ChiliProject</td>
<td>Most of Redmine’s advantages</td>
<td>Not as much plugins</td>
</tr>
<tr>
<td>Google Code</td>
<td>Ready-to-use</td>
<td>Not a lot of features Open-source projects only</td>
</tr>
<tr>
<td></td>
<td>Easy to use</td>
<td></td>
</tr>
<tr>
<td>Mantis</td>
<td>A lot of features</td>
<td>Some important features are provided by a plugin. Not ergonomic Loosing momentum as a project</td>
</tr>
<tr>
<td>Trac</td>
<td>Mature</td>
<td>Interface different from other bug trackers May be slow</td>
</tr>
</tbody>
</table>
7 Conclusion

As we have seen, organizing the work of a team of developers is no easy task. In order to be productive, every aspect of the development chain has to be thought through.

Developing an efficient workflow is extremely important. If the developers can not adapt easily to a workflow, they will waste time, they will make mistakes, etc. At the same time, if your workflow is adapted to your project and well thought, not only will developers be able to follow it, but it will make the project’s history very readable and lessen the probability of future problems (e.g. conflicts during merges).

As seen in part 4, the workflow a company can impose on its developers can be divided in two parts: the overall organization of the code repositories and the branching model used. For a corporate environment, there are two organizations possible for the code repositories: centralized or semi-decentralized (referred to as the "snowflake organization"). In all cases, it is recommended to use a decentralized source code manager but it is not mandatory in the first organization. The choice between the two will mainly depend on the size of the development team: the bigger it is, the more it is recommended to choose a semi-decentralized organization where each sub-team can work on its own with its own code repository. The branching model is also extremely important if you use a decentralized source code manager –which, once again, is highly recommended. It is advised to design your own model because each software project has different needs and release process. The branching model should adapt to those constraint and not *vice versa*. A good base to design a branching model is the Driessen model [13], also known as Git Flow.

Another part of the workflow that has to be thoroughly thought is the code-reviewing policy as presented in part 3. In the context of a commercial software project where a code-reviewing policy *à la GNU/Linux* cannot be used, different code-review processes can be considered: extreme programming, manual team code-reviewing sessions or dedicated tools. In parallel, another aspect has to be considered: when is the code-review done? There are three possibilities: before the upload to the code repository, after the upload or "in between" (the code is uploaded but not yet merged). The last one has perhaps more advantages but few tools implement it and it can be hard to learn.

You also have to choose the right tools according to your needs and according to how they can interact with each other. If you choose a solution that is too difficult to apprehend people will make mistakes using it and you risk misconfiguring it. But if the tool is lacking features you could need, it will slow the developing process down.

For instance, as seen in part 5, the interaction between a code-reviewing application and a continuous integration tool can speed up a lot the code-review process by automatizing some tasks such as basic compiling to advanced tests on the code.

But, as explained in part 6, a full development chain that includes the two applications mentioned above generates a lot of information, most of which is not actually useful in the everyday development of a software project. Therefore, a way to sort this information is needed and a bug tracker is the obvious answer to this problem because it can easily serve as the center of the development chain. One problem that arises is the choice of the actual bug tracker solution. There is a lot of them and even if all provide the same basic features, a special attention must be given to this choice as it will highly impact the way the development team will work. In a corporate environment, where a
third-party service such as Google Code is probably not wanted, web applications such as Bugzilla or Redmine are probably the solution.

Finally, code-review and, if possible, automated code-analysis should always be a part of the development chain. The time you loose by rectifying the mistakes they allow to detect is more than compensated by the time you save later not having to solve bugs reported by testers or users. They also prevent some security problems, improving the overall quality of your project and better protecting both your users and yourself.
References


List of Figures

1. Commit waiting for a code review on a central repository 9
2. Possible snowflake organization for the development of an os 13
3. Fast-Forward vs. Non Fast-Forward 15
4. A visualization of the Driessen branching model 16
5. ANSSI’s adaptation of Git Flow. Dotted lines indicate omission of commits. 17
6. Limitations of the Fast-forward only policy 18
7. Gerrit managed to merge commit 3 19
8. Non fast-forward merge 19

List of Tables

1. Summary of code-reviewing policies 11
2. Summary of the possible organizations 14
3. Summary of bug tracking solutions 28