Elements for a Cloud-Based Development Environment: Online Collaboration, Revision Control, and Continuous Integration

Tommi Mikkonen
Tampere University of Technology
Korkeakoulunkatu 1
FI-33720 Tampere, Finland
tommi.mikkonen@tut.fi

Antti Nieminen
Tampere University of Technology
Korkeakoulunkatu 1
FI-33720 Tampere, Finland
antti.h.nieminen@tut.fi

ABSTRACT
In almost any other field than software development, the World Wide Web, or simply the Web has revolutionized collaboration by providing a platform for cooperative applications and services. In contrast, so far revision control, also referred to as version management, has been the key technology for enabling collaborative software development. However, even today it is feasible to build a cooperative development environment that runs in a cloud, following the spirit of Google Docs, where developers can cooperate in real time. In this paper, we take a step beyond from the design of the technical artifact that allows collaborative coding, and consider the consequences of such an approach to software development in a more general sense, in particular from the perspective of version management. The goal is to integrate collaborative development with traditional facilities commonly associated with software engineering. This in turn will lead to a development approach where new opportunities complement tools that are known to work and provide a lot of added value in software development.

Categories and Subject Descriptors

General Terms
Management, Documentation, Experimentation.

Keywords
Version management, revision control, collaboration.

1. INTRODUCTION
Like many other engineering disciplines, core technical activities of software engineering are commonly performed in isolation. While the main designs and principal guidelines can – and often are – be crafted in a collaborative fashion in meetings of different kinds, the final design of software artifacts that will later be compiled and run are usually written by individual developers. This way of working has led to established ways of crafting software in a collaborative fashion, where tools and interfaces play a major role in helping developers to cooperate in a seamless fashion. For instance, one of the prime drivers of architectural design is separation of concern, which allows developers to focus on different aspects of the system, located in different modules.

One of the commonly used tools is revision control software, also known as version management software. Revision control systems allow developers to freeze code files in a certain state. As code files can be checked out for upgrading them and checked back in after the changes have been made in a fashion that protects developers from each others’ changes, a revision control system forms a primitive tool for the collaboration of developers working on the same software project.

In almost any other field than software development, the World Wide Web, or simply the Web, has revolutionized cooperation. Twitter, Facebook, and many other services are manifestations of the promise of the web as an instrument for instant worldwide sharing of information. Consequently, given the communication problems that software engineers are commonly claimed to have, the web has the power to liberate developers to solving the actual problems in cooperation instead of detailing the practicalities and tools that are needed for basic communication. For instance, building on the technologies that are readily available, it is feasible to build a cooperative, integrated development environment that runs in a cloud, as we have already proposed and demonstrated in [1] [7]. Furthermore, cloud computing facilities can be used to offer virtually limitless capacity that at present is often poorly benefitted from during the development phase.

In this paper, we take a step beyond from the design of the technical artifact that allows collaborative, cloud-based coding, and consider the consequences of such an approach to software development in a more general sense. In particular, we will be revisiting the goals of revision control, whose role can be questioned in the presence of powerful features such as timeline in social media that has become commonplace in various services. This issue is particularly important when considering activities such as end-user programming, where the background of developers may vary and some of the developers may not be comfortable with traditional revision control systems. In addition, we will briefly address some of the practices that are commonly in place in web based applications and services in order to revisit revision control practices in the light of real-time

---

1 In the case of pair programming, individual pairs of programmers.
collaborative development. In particular, the effects of the seemingly infinite amount of cloud computing capacity will be addressed, as it is becoming increasingly feasible to perform compilations and run tests on the fly even when the developers are not asking for those. So far, it has been possible to check syntax and semantic information this way, whereas in the future, we believe that it will be possible to run also sets of test cases to help the developers working on a particular piece of code.

The rest of this paper is structured as follows. In Section 2 we introduce the background of the work by discussing the ways that have been commonly used to enable cooperation between developers working on the same system. In Section 3 we introduce the tool we have built for cooperative software development. In addition to the parts that have been implemented so far, we discuss extensions we believe can be easily added in the system, because similar implementations are already in widely deployed use. In Section 4 we consider the role of revision control in an environment where development can take place in a collaborative fashion. In Section 5 we briefly address related work, and in Section 6 we provide potential directions for future work. Towards the end of the paper, in Section 7 we draw some final conclusions.

2. BACKGROUND

As the size and complexity of software systems has increased, increasing amount of communication between developers has become a practical necessity in the construction. Moreover, communication is taking more agile forms. For instance, wikis have largely replaced traditional paper documents, web-based systems, offering support for cooperation, are available for managing requirements, and instant messaging systems are used by developers to exchange ideas on the fly.

However, when considering the cooperation that is directly associated with program code, less new collaboration opportunities has been introduced. When considering how designs are created in practice, there are two major ways to support the collaboration of developers, both of which have been traditionally used in numerous designs. One is to design the system so that there are a lot of loosely coupled modules that can be designed and maintained somewhat independently, and the other is to design a development process that minimizes the possibility of development work that might interfere with other activities.

When relying on loosely coupled components, the developers working on one module can take design decisions freely as long as they do not change the external interface of the module, following the spirit of information hiding in the large. While the components are coupled loosely, they can be tightly coupled internally. Because each component can be separately edited by a different set of developers, this approach is commonly used in many open source systems, where the development effort is distributed in ultra-wide scale. When pushing this approach to the extreme, a pattern called Core-Periphery (Figure 1) has been proposed [8]. This pattern employs a single, small, extensible core – which is typically highly coupled – surrounded by a much larger periphery, which typically exhibits low coupling. This pattern is intuitively suitable for meeting the requirement of ultra-wide distributed development. The core typically presents itself to the rest of the system as a “platform”, which the periphery accesses through a set of APIs. Examples of systems building on this architecture include Linux Kernel, Apache HTTPD server, and Mozilla Firefox, all of which have been designed by a community, with an intent to support different extensions for various purposes. Moreover, the internal structure of each of the periphery systems can follow the same pattern. Consequently, the different modules can evolve in accordance to their own versioning systems, and once a module builds on the facilities of other modules, it is possible to define the versions that are compatible.

![Core-Periphery architectural pattern.](image)

Figure 1. Core-Periphery architectural pattern.

When considering the alternative approach, aiming at a process where there are minimal inconsistencies in the system, the most obvious example is continuous integration [4]. In continuous integration, illustrated in Figure 2 at the level of concept, tools are used to build and test the system whenever a change – be it how small – is made in the master code base. The goal of the approach is that all the developers always work with the newest possible code base. When pushed to the extreme, if the build process fails, developers will stop working on their current tasks, and collaboratively they will first fix the build, and only then continue with the development of additional features. When considering the approach from the perspective of version management, new versions are created frequently. As the whole system is built and tested after the creation of each version upgrade, version management as well as associated integration and testing tools, where tools such as Bamboo (www.atlassian.com/software/bamboo) or Hudson (hudson-ci.org/), play a key role. To summarize, the prerequisites of continuous integration – code repository into which updates are made frequently automatic build and testing infrastructure – allow a development mode where every time there is a commit, a new build is made and tested. This means that the build and test system must be run in a capable enough environment, such as special cloud service that is elastic enough for changes in the load, something already proposed in e.g. [14].

In many real-life cases, elements of both of the above approaches are present, although not necessarily in a thoroughly planned fashion. For instance, architectural support can be used for major upgrades of the system, whereas continuous integration is used when smaller changes within the same architecture take place. However, since designing the architecture with collaboration explicitly in mind and setting up the version control system both require explicit effort, we believe that for many small-scale experiments these are considered too extensive means, and little facilities for cooperation – which could be later required in the form of e.g. assistance in the development of a certain snippet of code – are introduced.
In general, the web has revolutionized the fashion cooperation with which users can edit code at real-time and also communicate.

To summarize, the Web enables direct communication between developers already today via various means. Consequently, the traditional ways of software development, where upfront design and work was commonly invested in architectural design and version management is being challenged by more agile ways of working. As with Google Docs we do not wish to create intermediate versions but can always return to whatever stage of the document we wish, tomorrow’s software development will be increasingly treated in the same fashion – collaboratively, and with automatic support for numerous details we today handle manually in one way or another.

The challenge of this transition is to embed the new ways of working in the existing infrastructure. We have become masters in using the methods, technologies, and tools that already exist, and in order not to redesign them for collaborative development, approaches are needed where they can still play their traditional role.

3. DESIGNING A COLLABORATIVE DEVELOPMENT ENVIRONMENT

In general, the web has revolutionized the fashion cooperation takes place. Reflecting the facilities of services such as Google Docs already mentioned above, real-time collaborative editing of program code is increasingly practical. In the following, we introduce our proof-of-concept system CoRED [7], where the goal has been the creation of a collaborative development environment, with which users can edit code at real-time and also communicate using means that are familiar from social media to support the development.

CoRED is a cloud based IDE and hosting platform for implementing and publishing Java based web applications [7]. No other tool except the browser is needed, and therefore no explicit installation is necessary in any phase of the development. Applications are created using the browser-based editor. Once an application – the most obvious use is web applications – is complete, it can be immediately deployed to a cloud computing environment, which scales up by starting more server instances when needed. Servers are able to host multiple applications, which share many resources although direct interfering with each other is prevented. The structure of the development environment is illustrated in Figure 3. The main components of the figure are addressed in the following.

Ace. Ace is a commonly used open source code editor written using JavaScript (http://ace.ajax.org/). It can be easily embedded in any web page, and it has support for several different programming languages, including Java, which we use for composing applications. Furthermore, the behavior of Ace can be customized extensively without editing its source code. In addition, Ace offers information that helps in integrating the editor into a broader framework, which has helped us in the development.

The Vaadin Framework. Vaadin is an open source framework for developing Rich Internet Applications (RIA) using Google Web Toolkit, GWT [13]. GWT is an open source development system for creating Ajax applications [11] in a fashion where applications are constructed in Java and then compiled into JavaScript which can be run on all browsers. As indicated in the figure, Vaadin applications are partitioned in a fashion where all application logic is run on the server, and only user interface is downloaded into the browser. The Vaadin framework connects the client side user interface, compiled with GWT, to the server side application logic implemented directly with Java [6].

Java Developer Kit (JDK). JDK contains the necessary tools for compiling, executing, debugging, and documenting Java programs. Furthermore, it includes tools.jar package, which contains useful Sun/Oracle specific APIs for compiling, diagnosing, and parsing the source code. In our system, we use JavaCompiler as an interface for invoking Java compiler from the program. The compiler generates complete error and warning
diagnostics during compilation. In addition, by extending 
ClassLoader and StandardFileManager, the source and 
destination of compilation can be redirected. Finally, the offered 
API contains TreePathScanner that can be used for processing 
source code. The scanner visits all the child tree nodes of the 
source code and can therefore be used for finding classes, 
methods, variables and other kinds of Java structures. The system 
is available at http://antti.virtuallypreinstalled.com/cored for trial 
use.

In the present implementation, demonstrated in Figure 4 as a 
screen snapshot, we have built-in support for the most obvious 
features that a collaborative IDE should include. They are briefly 
addressed in the following.

Projects. Like with any IDE, the developers can select the project 
they plan to work on. In practice, a project consists of a collection 
of matching files that jointly constitute a system that can be 
compiled and run.

Editing. The system provides facilities for writing code in 
different code files. Editing can take place in parallel. The 
collaboration between developers takes place using Neil Fraser’s 
Differential Synchronization algorithm [5], a robust and 
convergent collaborative editing algorithm, which has been 
widely used in collaborative web applications.

Deployment. The present implementation is meant for 
implementing Rich Internet Applications (RIA) using Vaadin as 
the platform. Built-in facilities exist for publishing these as on-
line services automatically, and a simple click of the deploy 
button at the left-hand side will result in the creation of the 
service.

Instant messaging. The system includes support for instant 
messaging between developers. The goal of this facility is to 
separate code comments and developers’ discussions regarding 
certain pieces of code. Therefore, messaging can be associated 
with code lines, much in the same fashion as collaborative text editors allow one to insert comments in the text. In addition, we 
have implemented a global chat that can be used for issues 
regarding the system as a whole.

Facebook integration. In the present implementation, we use 
Facebook accounts and authorization mechanism for identifying 
programmers that join the development, although anonymous 
editing is also possible for convenience. In addition, introducing 
more social media features in the editor generally supports 
studying social aspects of software engineering. Consequently, the 
present implementation can act as a prototype that is used for 
gathering data on developers’ behavior in the long run. Again, 
more elaborate implementation is probably needed in the long 
run.

In addition to these features, mainly focusing on developer-level 
activities, there are a number of other facilities that are needed for 
supporting the full software engineering process. These include a
link into a requirements management system, test facilities, and version management. While linking with requirements is difficult to automate, testing can be treated the same way as we have used JDK so far – as changes happen, the tests take place automatically in a cloud-based service, and the potential problems are made immediately visible for the developers. This results in a system that is able to build the system being developed automatically as new versions of software are written. Obviously, as testing may reveal also more complex errors than syntax and semantic errors, locating the problems and displaying them in association with code requires more work, and may require augmentation of the code as well as associated test cases. Defining such detail remains future work at this point.

As for designing the test and integration environment as a whole, illustrated in Figure 5, such systems have been designed and tested numerous times when aiming at test-driven development and continuous integration (see, e.g. [3]). Therefore, although there are numerous methodological aspects associated with collaborative development, we consider that including the backend infrastructure in our implementation is to a great extent a simple matter of programming. Furthermore, we can use the output from the backend in a similar fashion as we now use JDK, and provide associated information immediately to the developers working on a particular piece of code.

With regard to version management, however, the situation is less obvious. The complicating factor is that some of the aspects of version management are associated with the user whereas some others can be automated. We will address those next.

4. REVISION CONTROL REVISITED FOR COLLABORATIVE DEVELOPMENT

When the environment described above is put into action, there are several consequences that affect the development. The three major changes are that 1) revision control need not be the main tool for collaboration between developers; 2) some of the revisions can be created automatically by the environment, if the system appears robust enough; 3) developers can join forces to create a common version. These three issues will be discussed in the following in more detail.

Real-time cooperation between developers instead of revision control based collaboration. The possibility to communicate in real time challenges the need to use revision control as a tool for collaboration. Moreover, more visual tools such as timelines can be used to show the progress of the project instead of logs from a version control system. For novice developers and end-user programmers in general, such an approach may be helpful, and even for a more seasoned developer, the ability to get back to a phase in development that was not recorded in version database is most likely a feature that at times increases productivity.

Still, while the above seems to imply that there is no need for explicit version control, we expect that developers still behave as if there are versions, because it simply adds value to the development to consider certain stages of the development as tried and true configurations – or versions, in accordance to the terminology of software engineering. However, these versions play a different role than the versions of today’s development. Since the part of revision control that is associated with collaboration has been removed, revision control becomes more closely associated with product management and less with developer activities. Thus, each coherent version reflects a de-facto baseline of further development, which can be integrated and released, at least potentially. These robust configurations have the potential to act as starting points for further development as well as natural candidates for forking the code base, for which revision management is also an important enabler.

Automatically created versions. Because the ability to cooperate online disregarding version management, and the fact that only tried and true configurations should be included in the version database, there can be automated decisions regarding revisions. With the above infrastructure, the build and testing facilities can deduce whether or not a given set of files forms a coherent system, in analogy to JDK that constantly tries to compile the file being edited in our present implementation in order to show syntax errors. When a coherent system that passes all the tests emerges, the files constituting the system can be inserted in the version database automatically. If tests fail, the data from the tests can be made immediately available to the developer, much in the same way we today show syntax and semantic errors in a code file based on parsing the code file. This in turn enables faster turnover times in the development, as the developer does not need to stop to run the tests.

To summarize, the tools have the capability to take proactive decisions on behalf of developers when a potentially usable
system results. We expect that this creation of implicit versions when everything is working can be well-suited for less trained developers who are already familiar with web way of working, where every action is recorded and made recoverable.

At this point, it is important to note that the automatic creation of versions does not prevent the creation of explicit versions if and when developers want to create them. However, most likely automatically and explicitly created versions should be treated differently, since their goals are likely to be different. In addition, a visual representation of versions (and version history) is needed, as otherwise it can be cumbersome to determine from which point to continue development

Cooperation for a jointly created version or solving a conflict. In addition, developers can compose a joint version that satisfies them all. In this case, the new version is created in a collaborative fashion, and in the most modes case collaborative editing is simply used to e.g. sort out conflicts in revision in collaboration.

While less revolutionary, the effects in the development process are less radical and with seamless integration into a version management system, collaborative editing still has the potential to provide added value for developers.

5. RELATED WORK

Version control has long been a key enabler for cooperation between developers. An investigation on current version control practices is provided in [12]. While the basic concepts – checking code in and out of the code data base – have remained the same, there are different designs that allow central and distributed version management. Furthermore, newer systems have additional features that encourage collaboration. The most obvious example is Git (http://git-scm.com/), a distributed version control system that is well-suited for distributed development of software, and associated repository GitHub (https://github.com/), where developers can collaborate with others. Despite the collaboration facilities that have been included in GitHub, both Git and GitLab are still building on properties of traditional version control systems, and do not aim at real-time collaboration between developers in a fashion where several simultaneous authors would be allowed. Same goes for numerous other commonly used version control systems, including systems such as Subversion (SVN, http://subversion.tigris.org/), Mercurial (http://mercurial.selenic.com/), or Concurrent Versions System (CVS, http://www.nongnu.org/cvs/), just to give a few examples.

Reconsidering a more fine-grained revision control in the light of collaborative software development has already been proposed by Magnusson et al. [9]. Although closely related, our work differs from this proposal in many ways. The main difference is that Magnusson et al. propose that each developer is still producing their own revisions, and they are joined in the same way as traditional versioning system works. We, however, aim at reconsidering the role of version management based on the assumption that the background system can determine when a complete, runnable system exists by relying compilation, testing, and integration capabilities available in the cloud. However we do accept that in both cases the technical approach is manifested in micro-scale revisions, where collaborative editing is enabled.

6. FUTURE WORK

While this paper builds on real implementation work as far as developer cooperation is concerned, a part of the required facilities are missing at this point. In order to test the feasibility of the proposed approaches to revision control, we must create support for automatic builds and tests. While this requires a considerable engineering effort, we expect that no major obstacles will be discovered upon creating the infrastructure, because numerous reference systems already exist and are in use in companies. Still, integrating such a system with our cooperative development environment Cored requires an extensive amount of engineering work.

Testing the above approaches to revision control and other pieces of infrastructure in practice is an obvious direction for further work. Our plan is to involve university students in the experiment as soon as the building and testing infrastructure has been integrated with Cored. Results from these studies will be reported separately, with the paper on cooperative resolution of conflicts currently being finalized [10], because this can be experimented even without the infrastructure for continuous integration.

Finally, in order to study how developers benefit from the proposed facility, it is obvious that a series of usability studies is needed. Based on the results, it is possible to refine the essentials of the system. Results of such studies also help in identifying social media features that can be applied in software development.

7. CONCLUSIONS

The introduction of a collaborative development environment challenges the role of revision control, which no longer needs to act as a tool for enabling collaboration between developers, but it can be used a true configuration management tool where versions that are known to be robust and runnable are stored in case of e.g. branches. While versions can and (most likely will) be made manually, it is also possible to have automatic revisions if the right backend is made available, with correct functionalities such as building and testing facilities. All the required technical enablers for building such backend already exist, and are widely in use in the form of continuous integration. Consequently, collaborative editing, constant builds and tests as code changes, and automated configuration management is not a mission for moon but something that is within the reach of developers today.

8. REFERENCES


