# Enabling collaboration and tinkering: a version control system for block-based languages

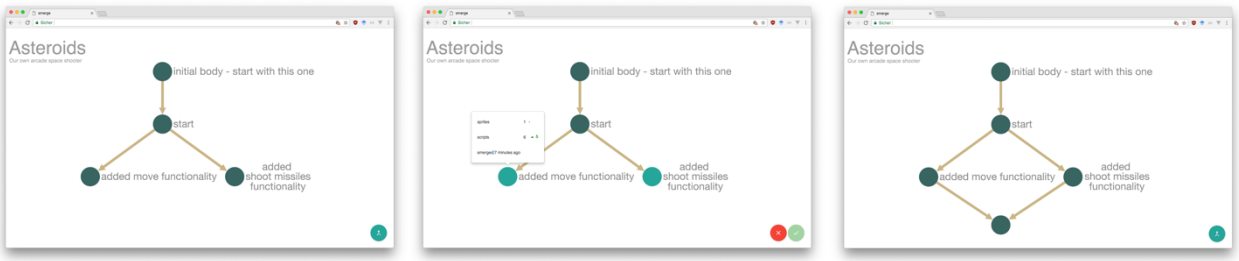
**Tilman Michaeli**[*], *tilman.michaeli@fau.de*
Friedrich-Alexander-Universität Erlangen-Nürnberg

**Stefan Seegerer**[*], *stefan.seegerer@fau.de*
Friedrich-Alexander-Universität Erlangen-Nürnberg

**Ralf Romeike,** ralf.romeike*@fau.de*
Friedrich-Alexander-Universität Erlangen-Nürnberg

## Abstract

Version control systems are essential for coordinating teamwork when working in projects. They support computational thinking practices such as collaboration and tinkering. Yet, when using block-based languages, which are an excellent choice for novice programmers, there is no adequate solution that allows this form of collaboration. This paper presents a concept for a simple and easy-to-use web-based version control system as well as an exemplary implementation for the popular language *Snap!*. Based on an analysis of existing version control systems, their use in Computer Science Education in university and school contexts, and specifics of block-based-languages, key design principles for a version control system for block-based languages are outlined. Based on this, possible use cases for such a version control system in classroom environments will be discussed.




*Easy-to-use version control system for collaboration and tinkering with block-based languages*

## Keywords

Version Control System; Block-based Languages; Snap!; Computational Thinking; Collaboration; Tinkering

[*] These authors contributed equally to this work.

### Abstract

Version control systems are essential for coordinating teamwork when working in projects. They support computational thinking approaches such as collaboration and tinkering. Yet, when using block-based languages, which are an excellent choice for novice programmers, there is no adequate solution that allows this form of collaboration. This paper presents a concept for a simple and easy to use web-based version control system as well as an exemplary implementation for the popular language *Snap!*. This concept is based on an analysis of existing version control systems and their use in Computer Science Education. Furthermore, possible use cases for such a version control system in classroom environments will be outlined.


# Introduction

Collaborative learning based on the work of Vygotsky integrates social aspects into constructionistic learning. Much has been written about the advantages of working in groups early on (e.g. Chase & Okie, 2000). Using Projects and project-based-learning (PBL) is one way to enable collaborative learning and a typical method of CSE. PBL is suitable for novices as well as for more advanced learners (Kastl & Romeike, 2015). When carrying out programming projects, one of the recurring challenges is that often arises that different versions of code have to be managed, project groups need to coordinate and merge their code. To work together efficiently, professionals use version control systems. Such systems enable collaboration by allowing teams to work together on the same project by sharing corresponding files. Furthermore, they keep track of revisions and, therefore, make it possible to go back to old versions, to track changes, to fix bugs, or to work in branches, which enables experimenting and tinkering in a sandbox. In a PBL context, such a version control system therefore enables collaboration and tinkering, which are approaches to Computational Thinking (CT) (Barr & Stephenson, 2011). Collaboration is an important aspect of working as a computer scientist. It includes factors such as decomposition of tasks or communication among each other and promotes motivation and commitment. When sharing or discussing their actions, learners can learn from, reflect and build on the work of others (Laurillard, 2009). In CSE, collaboration is considered to be important early on as well as throughout the whole curriculum. For example, the new ACM/CSTA standards for K12 education (Computer Science Teachers Association, 2017) require collaborative work already in Level 1b (Grades 3-5). Tinkering needs a risk-free environment that supports trial and improvement and fosters confidence, creativity and independent learning (Resnick & Rosenbaum, 2013).

The use of a professional version control system in the classroom is generally possible, but it is suitable only for text-based programming languages and comes with a lot of overhead. Even graduate students are often overwhelmed by the sheer complexity of professional tools (Haaranen & Lehtinen, 2015). For a lot of purposes, only a few functionalities like version history, merging and committing are needed. For novice programmers in Java, the integration of SVN and Git in BlueJ aims to reduce this overhead (Fisker, McCall, Kölling, & Quig, 2008).

Block-based languages like *Scratch* or *Snap!* are very popular in lessons with novice programmers for multiple reasons. As an example, they enable students to build creative programs without needing to worry about syntax (Maloney et al., 2004). Collaboration can take place in two dimensions. Currently, block-based languages only support collaboration in a sequential sense, by supporting and emphasizing remixing (Monroy-Hernandez, 2012). However, there are only limited solutions to enable parallel collaboration in the sense of working on one project at the same time. In a day-long workshop on agile project management with 9th to 12th graders, we recently had a team consisting of three programming pairs working in *Snap!*: Every time they wanted to put together their program pieces (e.g. for a prototype), every pair had to export their project and download it. Then they needed to transfer the XML files to one PC, e.g. via memory stick, cloud drive or e-mail. Afterwards, they had to import each project in *Snap!*, manually assemble their scripts, sprites, etc., test and fix bugs on this PC. Hence, the whole team sat in front of one computer to finish the prototype. This is not efficient, the same counts for redistributing the code to all team members. Afterwards, they needed to export the new project status, download it and share it to

the other two computers so that everyone was up to date. They also had to manage the versions properly in order to be able to use an old version if necessary. Experience has shown that this often causes problems. As one might expect, the students named these as major downsides to the workflow in the concluding reflection phase.

In order to address this problem, we decided to design a version control system for block-based languages. Therefore, a review of existing professional and didactically adapted version control systems and their use in CSE was carried out with the goal to identify important findings and adapt those for block-based languages. Using an exemplary implementation for the block-based language *Snap!*, the proposed concept is demonstrated and its benefits are highlighted.

# Context and Background

## Versions control systems

Version Control systems offer a variety of functions important to collaborative practices. First of all, they document changes and their reasons by providing a history for each file under version control. Each change can be described and summarized by the user through comments. Furthermore, version control systems offer the possibility to restore older versions. This way, unwanted or problematic changes can be reverted. Besides that, they enable coordination by offering features to resolve conflicts with multiple users working on the same file simultaneously. These features include locks to prevent multiple users from editing the same file at the same time, the automatic merge of concurrent edits or support for manual merges, if needed.

All files under version control are located in a *repository*. If a user adds new files to the repository and/or changes old ones, they *commit* their changes. Each commit contains *awareness information*, which describe the commit and can be divided into two categories. Internal awareness information includes changes made, time and date of changes, revision numbers, or names of committing users. This information is generated automatically. On the other hand, there is explicit awareness information, which is stated explicitly by the user. It requires explicit action. One example is a *commit message*, in which a user describes the changes he has made (Fisker, McCall, Kölling, & Quig, 2008). If no one else changed any files in between the last commit and the new one, this results in a new version, also known as a *revision,* of the project without any additional action. If someone else has made changes to files in between, but they are not in *conflict* to each other, these changes are automatically *merged*. If there is a conflict, when e.g. the same line of source code was changed by more than one person, it must be solved manually by choosing for each conflict the version which should be in the new revision. Changes between revisions (added through commits) can be viewed via *diffs*. Old revisions can be viewed or *reverted* to at any time. Furthermore, modern version control systems offer the possibility to *branch*. A branch is an alternative path starting from a certain revision, so that changes can be made in parallel. Branches are used for development of new features or experimenting, without impacting the current product and state of the project. A branch can be merged into the master/production branch again later on, e.g. if the new feature is fully implemented and tested.

Version control systems can be divided into *centralized* systems (like CSV or SVN) and *distributed* systems (like Git or Mercurial). In centralized systems, the repository is kept on a remote server everyone has access to. Whenever a user wants to introduce changes, they retrieve the latest version from the server first. As commits are transmitted to the remote server immediately, any recent changes must be merged, and conflicts have to be resolved before the commit. In contrary, distributed systems store the whole project history locally on every computer but also on a remote server. Therefore, each commit will initially be registered locally. To make changes by a commit available for other people as well, they have to be *pushed* to a server. From there, they can be *pulled* by each collaborator to be available locally. This way, merging and conflict resolution are not necessary for commits, but when interacting with the server (pushing and pulling). Distributed

version control systems are dominant nowadays. Their main advantages are the local “sandboxes” which enable local changes, reverts etc. for every user offline, the easy branching and merging, and the independence from just one location where everything is stored (Somasundaram, 2013).

## Version control in the classroom

Version control systems are used both at school and university level. At universities, control version systems are used frequently. Typical use cases include the provision of course materials or the submission of homework. Throughout literature, advantages of the use of control version systems can be divided in organizational and pedagogical ones. Organizational ones are the easy way to post assignments and give feedback, the possibility to start with skeletons, revert changes and work remotely, as well as having timestamps for submissions (Lawrance, Jung, & Wiseman, 2013). The pedagogical advantages include easier collaboration, the possibility to assess individual contribution, making the development process visible for the teacher and data security in the sense of a backup (cf. Reid & Wilson, 2005, Lawrance, Jung & Wiseman, 2013, Glassy, 2006). Overall, it is reported that version control systems are considered useful by students and teachers alike (e.g. (Isomöttönen & Cochez, 2014)). Just like the advantages, further experience and especially problems in the use of control version systems are reported. These hint at obstacles that must be addressed in a pedagogical version control system. One reported problem is a non-iterative workflow with long periods without a commit. This is an obstacle especially at the beginning and takes a lot of the advantages away (Glassy, 2006). Overall, professional version control systems are reported as hard to learn (cf. Isomöttönen & Cochez, 2014, Haaranen & Lehtinen, 2015). Students sometimes damage repositories so that tutors need to repair them, or misuse features, e.g. repeated checkouts instead of updates in SVN (Reid & Wilson, 2005). In some cases students even accessed third-party repositories (Reid & Wilson, 2005). From a student's point of view, conflicts and their resolution are the most complex and difficult tasks (Isomöttönen & Cochez, 2014). If the students always work in the centralized repository they have more problems than when they work in their own branches and merge when finishing a subtask (Lawrance, Jung, & Wiseman, 2013). In addition to students, teachers also need significant competencies to use version control systems successfully in the classroom. They need to set them up and configure them, and also support the students, e.g. when fixing broken repositories.

Brichzin and Rau (2015) give an overview of typical problems that can be addressed by the use of a version control system in a school context that matches our experiences. One problem is the pupils name convention and versioning practice – e.g. filenames like *game*, *game_2*, *game (copy)*, *game (working)*. This is an obstacle for collaboration as well as identifying the current state of the project to work on after holidays or a longer break. If the students make no backups of the current or former status of the project, there is always the danger of deleting the work of up to several weeks by accident. The next problem they mention is merging partial programs in PBL together regularly, no matter whether it is an agile project with iterations or a traditional waterfall project. Using a version control system facilitates regular merging and therefore helps to identify interface problems at an early stage of the project and address them accordingly. Another typical problem within the school context is that an entire team gets blocked because a student has forgotten the current code at home or they lack access to his account while the student is sick. Furthermore, enthusiastic students can't continue to work on the project at home, because the code is stored on the schools’ machines.
However, the introduction of a professional version control system is associated with a large overhead. Pupils must develop an understanding of the functionality of version control systems, familiarize themselves with the respective commands and working procedures. The complexity of this task poses problems even for entry-level professionals and graduate students. Therefore, Fisker et al. (2008) enabled group work support in the form of a simplified SVN and Git integration for the IDE BlueJ. One design principle for this was making awareness information available. This is important for group work and to keep track of others’ progress. Another explicit focus was simplicity by reducing overhead: files no longer need to be added to version control manually. Furthermore, many of the powerful but not essential features (such as branching, tagging, revert, single file functions) of SVN resp. Git are not available via the BlueJ IDE to ensure easy access. The same

goes for the graphical user interface, which is kept basic and simple. Functionalities such as *commit or update* are made clearer but still contain the standard terminology (e.g. “Update from Repository”)

### Block based programming

Traditional text-based programming languages have been used for introductory programming or computer science courses but are considered to be major entry barriers. Block based languages take away users' responsibility to take care of precise syntax compliance. They allow for easier realization of creative projects and give direct feedback by visualizing the current program’s state. Known examples of block-based languages are *Scratch*, *Snap!*, or *GP*. Most block-based languages are capable of running in the browser. Hence, they do not require an installation on student devices, making them a reasonable choice for educators. Those languages are used in schools and university courses, especially for novices. Using block-based languages with a traditional version control system is unsuitable: Traditional version control systems are built to manage multiple source text files, whereas in block-based programming environments, students interact with their project in a graphical way. In *Snap!*, objects are represented as so called “sprites”. Due to this, each object is represented graphically. Sprites have scripts, which are a sequence of several connected blocks (see Figure 1).

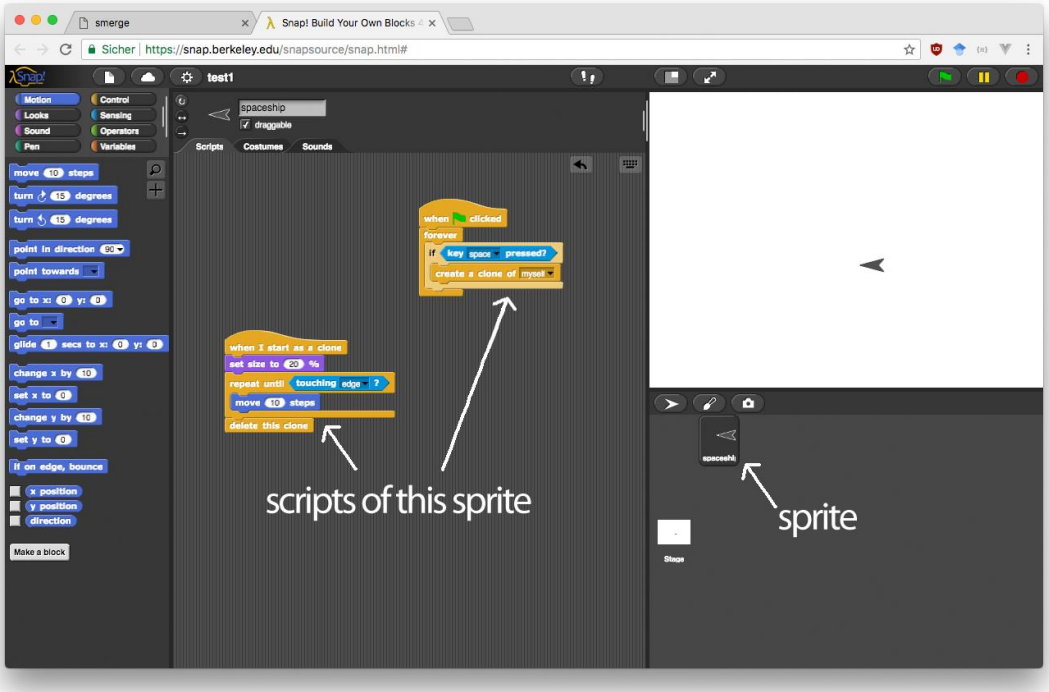



*Figure 1. Snap! user interface*

Certain block-based applications such as *Kanto, Blockly,* or *Netsblox* already allow multiple students to remotely collaborate on a project (Broll et al, 2017, Ohshima, Freudenberg & Amelang, 2017). However, they lack essential features that version control systems offer, such as version history, branches, or commits, which are considered essential in PBL settings.

## A version control system for block-based languages

Since existing tools of version control systems cannot be used for block-based languages, we have developed a solution based on research regarding the use of version control systems in the classroom. Therefore, we conducted a didactic transposition of professional version control systems with explicit attention to specific characteristics of block-based languages and needs of programming novices. It follows two guiding ideas:

- *Visualization.* The project status should always be visible at a glance. For this purpose, it should be displayed in a graphical way.
- *Easy to use.* The number of functionalities should be reduced, and unnecessary settings removed. The usage should be simple for both students and teachers. The latter usually having little experience with professional software development tools necessitate this guideline even more.

The second objective is consistent with the goals for version control realization in BlueJ (Fisker, McCall, Kölling, & Quig, 2008). In contrast to their solution, however, the operation is further reduced, and more simplifications are offered even for teachers. In the following, we describe the concept of the version control system and use images of our concrete implementation for *Snap!* to illustrate the concept.

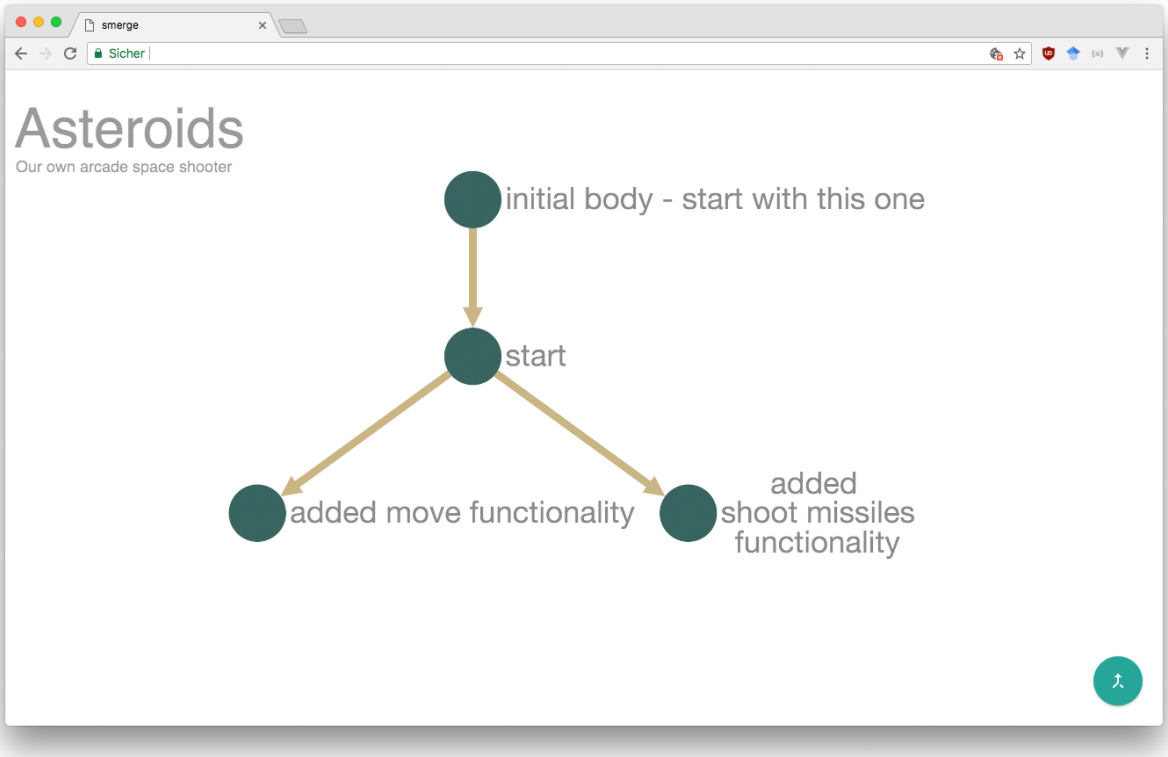



*Figure 2. Graphical visualization of a project*

## Conception

A version control system for block-based languages needs to be web-based. A project is represented by a graph similar to a *Git tree*. A node corresponds to a revision of version control systems (see Figure 2). It can be classified as a special case of a centralized version control system: there is only one project status stored centrally on a server. However, each user always works in their own branch, which is the norm for distributed version control systems. That means, if a user starts editing a revision, they implicitly start their own branch. If two or more users start working on the same revision, one individual branch per user is automatically created. Therefore, changes to the same revision cannot lead directly to a conflict. This also means that there is no explicit master branch. This addresses the experiences described by Lawrance that students had fewer problems when using their own branch for each sub-task (Lawrance, Jung, & Wiseman, 2013).

To create a project, users can either use an empty project or upload their own templates (e.g. with predefined blocks or sprites). It is also possible to upload additional files later on. As the version

control system runs in the browser, there is no need to set up an individual server to use the version control system or configure existing services. This allows the teacher to use version control systems without specific knowledge in a very flexible way.

Double-clicking a node in the graph opens the respective revision directly in the corresponding programming system (e.g. *Snap!*). In doing so, an additional button is inserted into the menu of the programming system. Clicking this button commits every change made by the user directly. By enabling a commit with only one command directly from the user interface, the described problem of few commits during long work periods is counteracted. The user is prompted to enter a commit message. In this way, students are motivated to briefly summarize their changes. Additional implicit awareness information such as timestamps, number of sprites or scripts added and removed, and total number of sprites or scripts are provided for each node. These make it easier for other group members to track changes in the project. Reverting to an old revision is done simply by opening the specific node and beginning to work from there.

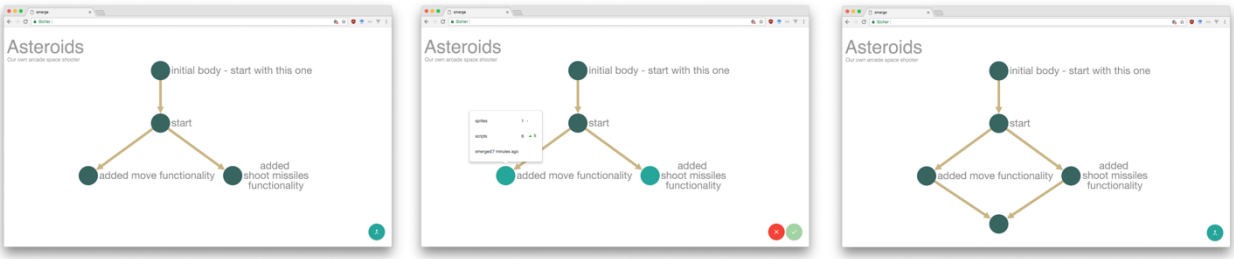



*Figure 3. Merge process*

The process of merging multiple revisions is initiated by selecting several nodes. If the selection is confirmed, the merge takes place. As long as there is no conflict, the system will merge automatically, similar to a professional version control system. This is the case if neither sprites nor scripts have been changed or only one user has made changes. The more recent version is identified by the ancestor relationship in the graph. Therefore, no interaction on the student side is necessary, unless several students have edited the same script. If this is the case, there will be a conflict. To resolve this conflict, we use a merge view providing all alternatives of the conflicting scripts side by side with comments attached. The students can then select the appropriate version or compose a suitable solution. For an example, see Figure 4: Bob and Susi edited the same script. In addition, Bob added another script. The new script will be merged automatically, while the existing script they both edited will raise a merge conflict. Therefore, both scripts are added to the merged version providing commentary details.

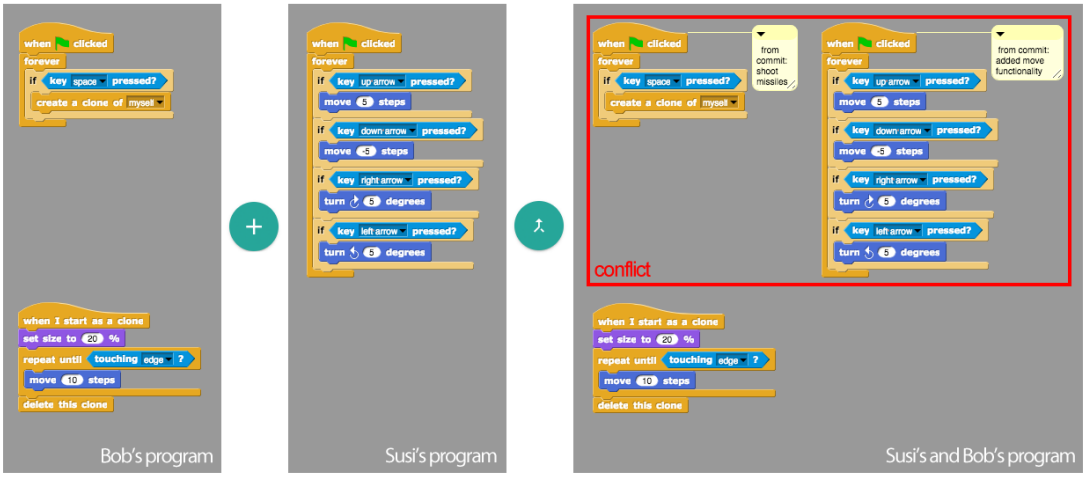



*Figure 4. Merge view*

Features such as add, push or status, which are known from professional VCS, are not necessary, because the structure of the version control system and block-based languages make these features obsolete. The only activities that students must actively do and learn are commit and merge.

The user guidance and interface are deliberately kept simple. The same applies to terminology, which must be tailored to the target group. Only two essential features need to be named. In discussions with computer science teachers, the use of the term *commit* was rated as difficult. The term merge, on the other hand, was considered suitable for students of all grades. Accordingly, the original term was used in this case, whereas the term *post to* <<*project_name*>> was introduced for *commit*. This term provides a suitable analogy for commit, comes from students' daily life and is appropriate for all ages.

In summary, key features provided are:

- visualization of the project and its history in a graph
- automatic branching for each editor of a revision
- opening each revision directly in the respective programming system in the browser
- easy commit from within the programming system used
- merging by selecting the respective nodes
- visualizing conflicts in a merge view
- providing implicit and explicit awareness information for each revision
- support for multiple templates and starting nodes

### Exemplary Implementation: smerge

With "smerge" (derived from the terms "*Snap!"* and "merge"), we provide an exemplary implementation of the described concept. The tool is implemented in Python 3 and JavaScript using the Django framework[1] and cytoscape.js[2]. For running a separate instance, only a server running Apache, Nginx, or similar is needed[3]. Instead of handling plain source code as with traditional version control systems, block based languages require a different approach due to their structure. Therefore, we utilize the XML file structure of *Snap!* projects, which differs from languages like *Scratch* or *GP*. On opening a certain revision, the associated XML project file is passed to a *Snap!* instance. In this step, a custom block containing the commit functionality (written in JavaScript) is injected. On commit, the current state of the project is exported in XML format and sent to our servers. As soon as the user triggers a merge, the corresponding project files are analyzed and compared on XML level. While conflict detection is easy when comparing source code in text form (usually line by line), once more a new solution for block-based languages is needed. Our solution regarding this conflict detection problem is based on sprite names and script coordinates. For conflict resolution, revisions and their ancestors are compared on XML level according to the auto-merge concept described above.

## (How to) smerge in the classroom

In the following, we will describe a possible workflow in smerge when using it in PBL. One way to implement PBL in the classroom is agile projects, which have already been used for PBL successfully (Kastl & Romeike, 2015). In doing so, agile practices such as user stories, standup meetings, pair programming, sprints or prototypes are adapted for the use in schools (Romeike & Göttel, 2012). We will use this framework to describe an exemplary workflow for smerge in PBL (see Figure 5).

For constructionist learning in school projects, students first create their own initial draft. Therefore, each group creates their own smerge project. In this way, they have created a place where all changes to the code are stored centrally. Each programming pair continuously works on one user

[1] https://www.djangoproject.com/
[2] http://js.cytoscape.org/
[3] A running instance can be found at smerge.org, the source at github.com/manzanillo/smerge.

story at a time. With the auto branch feature of smerge, every user story or feature is realized in its own branch. Pupils are therefore encouraged to work on tasks in parallel and can focus on a single feature each. Each pair has its own sandbox in which they can experiment and tinker.

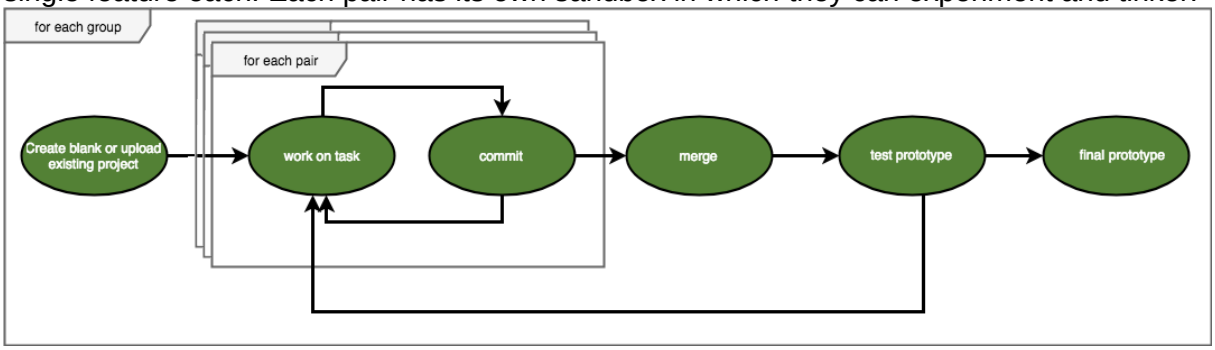



*Figure 5. Smerge workflow*

This type of workflow also promotes a more realistic form of collaboration in projects. At the end of a sprint, at the latest, the students will assemble their subprograms into a new prototype by using smerge to merge their branches. With smerge, every programming pair can then test the resulting prototype on its own as opposed to the described problem of all group members in front of a single pc. The merge is a ritualized team activity. A project also involves constantly improving the inner structure of the program through continuous refactoring. Smerge supports this with its merge view. By contrasting the individual parts from different sources with each other in the merge view, smerge motivates students to think about possible refactoring. For example, in order to make their own code easier to read, students tend to outsource redundant parts of the code into custom blocks. During the process, the visual representation of smerge, in addition to a possible project board, helps teachers and students keep track of the progress of the project.

The version control system can also be used highly flexibly in teaching outside of PBL. In the following, we will describe a lesson to introduce broadcasting in lower secondary education. In this scenario, every student will create his/her own individual instance of a prototype given by the teacher. Let's assume we want to make a group of penguins dance to a given beat. The beat is determined in a sprite created by the teacher and is delivered to the students via broadcasts. The students' task is to create their own penguin and let it react to the different bars of the music. For this purpose, the teacher provides a template with a simple penguin sprite. The students then rename their own sprite, design its looks and implement an individual behavior on the rhythm. After the students have finished their task, all individual solutions are to be combined into a complete work to emphasize the concept of broadcasts with more receivers. Without such a tool, it would require teachers to collect all students' solutions and combine them manually. In smerge, this combination is reduced to nothing more than the click of a button.

In addition, smerge can be used for a longer teaching sequence. In doing so, students develop multiple small programs to learn specific CT concepts. The class, or each student, can collect all these sub-projects within one smerge project. Every lesson, the students receive a new template in which they complete a specific task. After several units, the subprojects are combined to form an overall project, and a greater coherence becomes apparent. One example is the game Breakout, where the paddle, the ball and the bricks can be considered three sub-projects. So, handling user input, movement and bouncing of walls as well as list for placing bricks are main topics for individual lessons.

It would also be conceivable to use several templates for differentiation. Teachers can provide different templates for different types of learners. By providing weaker students with other tasks or more support, e. g. through given blocks, within the framework of a project, they can be supported individually. In addition, the process of tracking students' progress and assisting them accordingly is enhanced using smerge.

## Conclusion

In conclusion, this concept addresses all the initially described school-specific use-cases such as managing files and problems like blocked teams with code forgotten at home or sick students. It offers organizational advantages such as the possibility to share templates and skeletons in an easy manner, to start with multiple skeletons for different groups, to revert changes or work remotely from home. Teachers can concentrate on the pedagogical aspects of their lesson concept, as they are no longer involved in organizational activities such as setting up a server for a version control system. Regarding pedagogical advantages, both the current status of the project as well as its development process and progress become visible to both teachers and students. Because of the graphical visualization and the possibility to directly open, execute and test each node directly, this goes much further than professional version control systems. It allows for great flexibility and a constructionist way of teaching and learning: it supports PBL, differentiation, decomposing a greater whole in small learning lections, or class-wide collaboration.

The outlined concept for a version control system enables collaboration in block-based languages. The version history provides a risk-free environment that invites users to experiment and tinker. Features, design and interface are reduced and adapted to the target group of novice programmers and based on existing research and experience regarding the use of version control systems. Smerge as an exemplary implementation of the concept offers all these features and is ready to be used in CSE.

## References

Barr, V., & Stephenson, C. (2011, March Volume 2 Issue 1). Bringing computational thinking to K-12: what is Involved and what is the role of the computer science education community? *ACM Inroads*, pp. 48-54.

Brichzin, P., & Rau, T. (2015). Repositories zur Unterstützung von kollaborativen Arbeiten in Softwareprojekten [GERMAN]. *INFOS 2015 - Informatik allgemeinbildend begreifen* (pp. 73-82). Bonn, Germany: Lecture Notes in Informatics (LNI), Gesellschaft für Informatik.

Broll, B., Lédeczi, A., Volgyesi, P., Sallai, J., Maroti, M., Carrillo, A., Weeden-Wright, S., Vanags, C., Swartz, J., Lu, M. (2017). A Visual Programming Environment for Learning Distributed Programming. *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (pp. 81-86). New York, NY, USA: ACM.

Chase, J. D., & Okie, E. G. (2000). Combining cooperative learning and peer instruction in introductory computer science. *SIGCSE '00 Proceedings of the thirty-first SIGCSE technical symposium on Computer science education* (pp. 372-376). New York: ACM.

Computer Science Teachers Association. (2017). *CSTA K-12 Computer Science Standards, Revised 2017.* Retrieved from http://www.csteachers.org/standards

Fisker, K., McCall, D., Kölling, M., & Quig, B. (2008). Group Work Support for the BlueJ IDE. *Proceedings of the 13th annual conference on Innovation and technology in computer science education* (pp. 163-168). New York, NY, USA: ACM.

Glassy, L. (2006, Volume 21 Issue 3). Using version control to observe student software development processes. *Journal of Computing Sciences in Colleges*, pp. 99-106.

Haaranen, L., & Lehtinen, T. (2015). Teaching Git on the Side: Version Control System as a Course Platform. *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education* (pp. 87-92). New York, NY, USA: ACM.

Isomöttönen, V., & Cochez, M. (2014). Challenges and Confusions in Learning Version Control with Git. *Information and Communication Technologies in Education, Research, and Industrial Applications Communications in Computer and Information Science : 10th International Conference, ICTERI 2014* (pp. 178-193). Kherson, Ukraine: Springer International Publishing.

Kastl, P., & Romeike, R. (2015). "Now they just start working, and organize themselves" First Results of Introducing Agile Practices in Lessons. *Proceedings of the Workshop in Primary and Secondary Computing Education (WiPSCE '15)* (pp. 25-28). New York, NY, USA: ACM.

Laurillard, D. (2009). The pedagogical challenges to collaborative technologies. *International Journal of Computer-Supported Collaborative Learning. 4(1)*, pp. 5-20.
Lawrance, J., Jung, S., & Wiseman, C. (2013). Git on the cloud in the classroom. *SIGCSE '13 Proceeding of the 44th ACM technical symposium on Computer science education* (pp. 639-644). New York, NY, USA: ACM.

Maloney, J., Burd, L., Kafai, Y., Rusk, N., Silverman, B., & Resnick, M. (2004). Scratch: A Sneak Preview. *Proceedings of the Second International Conference on Creating, Connecting and Collaborating through Computing (C5 '04). IEEE Computer Society* (pp. 104-109). New York, NY, USA: ACM.

Monroy-Hernandez, A. (2012). *Designing for remixing: Supporting an online community of amateur creators.* Cambridge, MA, USA: Doctoral dissertation, Massachusetts Institute of Technology.

Ohshima, Y., Freudenberg, B., & Amelang, D. (2017). Kanto: a multi-participant screen-sharing system for Etoys, Snap!, and GP . *Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Experience* (pp. 7-10). New York, NY, USA: ACM.

Reid, K. L., & Wilson, G. V. (2005). Learning by Doing: Introducing Version Control as a Way to Manage Student Assignments. *SIGCSE '05 Proceedings of the 36th SIGCSE technical symposium on Computer science education* (pp. 272-276). New York, NY, USA: ACM.

Resnick, M., & Rosenbaum, E. (2013). Designing for tinkerability. In M. Honey, & D. E. Kanter, *Design, make, play: Growing the next generation of STEM innovators* (pp. 163-181). New York, NY, USA: Routledge.

Romeike, R., & Göttel, T. (2012). Agile projects in high school computing education: emphasizing a learners' perspective . *WiPSCE '12 Proceedings of the 7th Workshop in Primary and Secondary Computing Education* (pp. 48-57). New York, NY, USA: ACM.
Somasundaram, R. (2013). *Git: Version control for everyone.* Birmingham, UK: Packt Publishing Ltd.