In P. Romero, J. Good, E. Acosta Chaparro & S. Bryant (Eds). Proc. PPIG 18 Pages 114 - 121

Towards understanding Source and Configuration Management tools as a method of introducing learners to the culture of software development.

Aidan Delaney

University of Brighton a.j.delaney@brighton.ac.uk

Abstract. Despite the lingering stereotype, computer science and programming has long abandoned the development model consisting of a lone programmer writing code into the early hours of the morning. There may be aspects of this heroism in a product development cycle, particularly when close to a release date, however to bring a software product to market requires collaboration within a team of programmers. Developing software products, as opposed to simply programming, is a social process. We believe that learners gain a greater understanding of the software development process through social interaction with other developers. Such beliefs are consistent with the General Genetic Law of Cultural Development proposed by Lev S. Vygotsky. We consider Free and Open Source software development to be a readily available repository of differing development processes to use in introducing learners to the social side of software development. The contribution of this paper is to introduce and expand upon a vocabulary for discussing Source and Configuration Management tools in a Vygotskyian context.

1 Introduction

Free and Open Source Software (FOSS) is a social movement [1]. It involves thousands of developers collaborating over the Internet to produce software as an end product. Some of the more well known products are Mozilla Firefox [2], OpenOffice.org [3] and the Linux kernel [4]. Each of these products has a codebase of several million lines. As such, FOSS developers face the same issues as other developers who are working on large codebases. The similarity between FOSS development processes and non-FOSS development processes has been noted in [5, 6] with the similarities to eXtreme Programming [7] best practices particularly highlighted [8]. The FOSS development projects, over $100,000$ on sourceforge.net and over $3,000$ on savannah.gnu.org alone, represent a large fount of cultural information regarding software development.

There is a body of literature describing best practices in programming [9–11, 7]. This literature was derived from practice and can be considered a codification of aspects of the culture of software development. This culture has "been careful to avoid proposing a theory of software development" (Ward Cunningham in the

introduction to [10]) and is therefore independent of the underlying waterfall, incremental or agile software development model. Eric S. Raymond describes it as "hacker culture" [9]. It is our long-term goal to impart these best practices to students through the medium of FOSS culture.

Source and Configuration Management (SCM) tools are well understood in FOSS development and constitute one of the more important best practices in software development. In contrast SCM tools are poorly understood with respect to educational theory. We show why it is reasonable to provide a characterisation of SCM tools with respect to pedagogical theory. We characterise the source and configuration management tools used in FOSS development and present them in the context of Vygotskyian pedagogical theory. Finally, we use this relationship between FOSS, software development best practices, and educational theory to inform our teaching.

2 The Pedagogical Theory of Vygotsky

Vygotskyian theory neatly incorporates tools and culture into a pedagogy for developing scientific concepts. The theories of Vygotsky and the neo-Vygotskyian developments are well explained in [12–14]. The core of Vygotsky is the Zone of Proximal Development (ZPD) and the General Genetic Law of Cultural Development (genetic law).

Allen [15] provides both a succinct definition of ZPD as "the distance between understood knowledge as provided by instruction, and active knowledge as owned by individuals" and an example of what can be achieved when tool construction is founded on pedagogical dogma. The role of the ZPD in teaching has also been developed into the concept of scaffolding [8].

We are most concerned with the genetic law which states that learners internalise scientific concepts through inter-personal relationships [13]. Put simply; knowledge can be transmitted through engagement with culture. Such transmissions are mediated, meaning that learners "by the aid of extrinsic stimuli... control their [own] behaviour from the outside" [16]. There are three types of mediator

- signs and symbols,
- individual activities, and
- social relations.

FOSS development is a natural fit to the Vygotskyian model of a mediator as it consists of signs such as SCM tools; social interaction on mailing lists and online chat; and the individual activity of programming. The genetic law states that aspects of cultural development appear initially in social or interpsychological interactions and are internalised as intrapsychological interactions.

Mediators are either human, those developers we collaborate with which in an educational context also includes the lecturer and lab assistants, or symbolic.

> The sociocultural theory suggests that the style of human mediator cannot be properly comprehended unless the role of available symbolic mediators is acknowledged. [14, pg. 28]

It is therefore appropriate to understand symbolic mediators such as SCM tools before we consider the social relations involving human mediators.

3 Source and Configuration Management Tools

Source and Configuration Management tools are designed to allow software source code to be easily shared and developed in a structured manner. Modern SCM tools allow the same codebase to be developed by a small group of geographically concentrated developers as a large geographically dispersed team. Furthermore they allow new developers easy access to to the source. Distributed SCM tools such as Arch[1], Cogito, Monotone and Darcs treat new developers as equals to the existing project developers. This is opposed to centralised control of a repository where only a vetted group of developers are granted parity in the form of write access. Centralised SCM tools include the Concurrent Versions System (*cvs*) and Subversion (*svn*).

For example two developers, Alice and Bob, are sharing a codebase. If they work in a centralised fashion (see Fig. 1) they take a snapshot, or check-out, code and metadata from a repository. If Alice changes the codebase to add a feature or bugfix she then uploads her changes to the repository, a check-in. A check-in involves comparing the code in the repository with her working copy and determining the differences. The differences are computed using a delta-function, the most simple being the standard UNIX `diff` command. Bob also checks-out the code and modifies his working copy. During checkin Bob notices that there is a conflict, between his changes and Alice's changes. Bob will merge the changes before continuing the checkin. The differing SCM tools vary greatly in their strategy for merging.

If Alice and Bob were to use a decentralised method of managing the repository then Alice, rather than just checking out the current version of the code, would mirror a copy of the code and all the change history associated with that version. If Alice modifies the code, she checks her modification into her local mirror of the repository. Similarly Bob, on modifying his code, checks his changes into his local mirror. If both mirrors are publicly available, Alice can choose to synchronise with Bobs changes. Alice could also choose to merge a subset of Bob's changes into her repository.

That there are two categories of tool is not a function of technology but development culture as centralised SCM tools can simulate decentralised tools and vice-versa [17]. The differences between the tools can be explained by the differences in development style preferred by their users. However, the tools are self hosting meaning, for example, that *bzr* is the SCM tool for the *bzr* development project; the users of the SCM tools (at least initially) are the developers of the tools. Developers tweak their tools to operate in a manner which is intuitive

[1] Arch is the original specification of an SCM tool from which many implementations have been derived such as tla, baz, bzr and ArX.

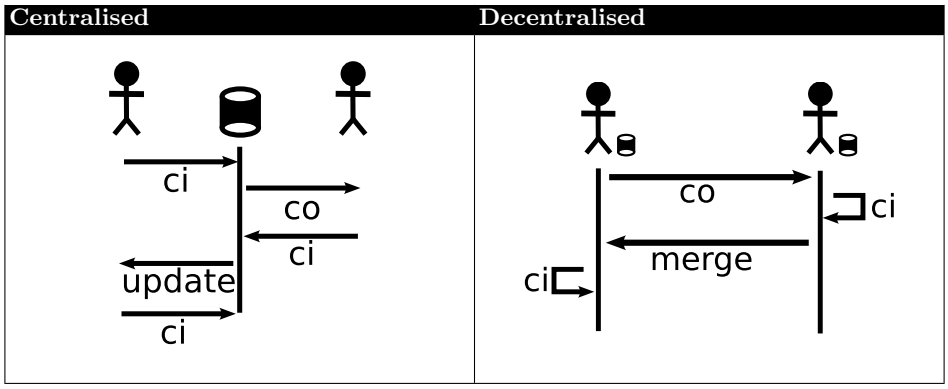

Fig. 1. A graphical depiction of the typical cooperation of two developers using both a centralised and a decentralised source and configuration management system. We use the common mnemonics *co* and *ci* to denote check-out and check-in respectively.

to them. As such an SCM tool embodies or codifies the behavioural process of development which is most intuitive to the community that produced it. Thus a modification to the development process manifests itself in the resulting artifact. The tool in turn feeds back into the development practice or communicates the newer development process to other communities of developers.

4 SCM Tools as Psychological Tools

Signs are those psychological tools whose uses are "directed towards the mastery or control of behavioural processes ... just as technical means are directed towards the control of nature" [18] as cited in [13]. SCM tools are signs directed toward the control of a developers concept of the software development process. We illustrate this using three examples, the first considering a personal software process style of development, the second considers how modern SCM tools encourage branching of the codebase and the third considering transmission of the *principle of orthogonality* [10, 9].

We consider the case of two distributed SCM tools, both are implementations of the GNU Arch specification. Though the tools share strong technical similarities they differ in their culture of dealing with ChangeLog entries. Users of *tla* are encouraged to "write the log message as you go along. In other words, take notes as you hack." [19]. The tool supports this using "wierd, often problematic filenames featuring leading "++"" [20]. In contrast *bzr*, supports a convention that is more recognisable to users of CVS and Subversion in which the user is encouraged to write the log file entry at the end of the iteration. One tool prefers a personal software process style of development whereas the other promotes backwards compatibility and a low barrier to entry for new users. These tools are cultural artifacts and psychological tools; they propagate a specific development

practices which inform development culture. The cultural practices encoded in the tools are transmitted to new users.

Project organisation is often encoded into SCM tools. In the case of Arch and *svn*, Arch imposes an explicit repository organisation whereas *svn* supports a specific way of working where the repository organisation is adopted by convention. The Arch developers view an SCM tool as a librarian for source code. From this point of view there is merit in users learning a revision naming schema, similar to the way users of a library learn to use the Dewey decimal system. Subversion imposes no such naming schema. It does, however, have a documented suggested schema for the organisation of a repository. Such repository naming schemas support ease of branching the codebase. Developers are encouraged to conceive of project progression along multiple parallel tracks rather than as a simple linear progression.

A trait of hacker culture is orthogonality, an insistence that a unit-testing framework be separate from, but complementary to, a language or paradigm. Thus a learner can study Java and JUnit in isolation. SCM encourages maintenance of orthogonality within the project development life-cycle. Iterations of project development are associated with a specific theme such as optimising a certain function or fixing a certain bug. Separation of these concerns can help to lower the cognitive load on a learner as they are encouraged not to attempt optimising a routine and fixing a bug within the same iteration. As such SCM tools mediate transmission of the cultural value of orthogonality to new developers.

Unlike introducing the use of an IDE or a unit test framework into a curriculum, SCM changes perspective from viewing software as a collection of source files to viewing software as a snapshot of a development process. Viewing software development over the longer term allows the staged introduction of other practices such as unit testing. It allows learners to see the long-term consequences of introducing a quick fix into code instead of undertaking refactoring. Other best practices such as automated testing and continuous integration can be easily introduced in a single quick iteration. Table 1 shows examples from FOSS projects of best practices that were introduced within a single iteration.

5 Curriculum Development

We have modified our level two software engineering course using SCM to provide structure for the introduction of orthogonal concepts. Such concepts include unit testing, automated building and software patterns. This course is a double-length module, spanning the entire academic year, in which students develop a codebase in accordance with the theory delivered through lectures.

The course is divided into a number of short iterations (see Fig. 2). Each iteration is assessed formatively during the weekly two hour lab slot. Three sessions are chosen during the year. In these sessions students are invited to summatively assess their own work. We moderate their assessment using short ten minute interviews. Students are not informed in advance of which labs are summative and

Staged introduction of	Example
unit testing	The author of tinymail added support for unit testing using the gunit framework in revision 143. The following revision added more tests.
automated building	The author of mono-tools added support in revision 18369 for the GNU autoconf build system, deprecating the more ad-hoc Makefile based build system.
continuous integration	An installation of buildbot at http://build.fluendo.com:8080/ continuously builds the latest gstreamer code pulled from their Subversion repository.
refactoring	The author of tinymail has dedicated several iterations such as revision 25 to the sole purpose of refactoring.

Table 1. Examples of the implementation of software best practices are taken from arbitrary FOSS projects.

which are formative. We do not believe that our approach violates the principle of *truth in sentencing* [21] as students are aware, at all stages, of the work we have expected them to have completed to date.

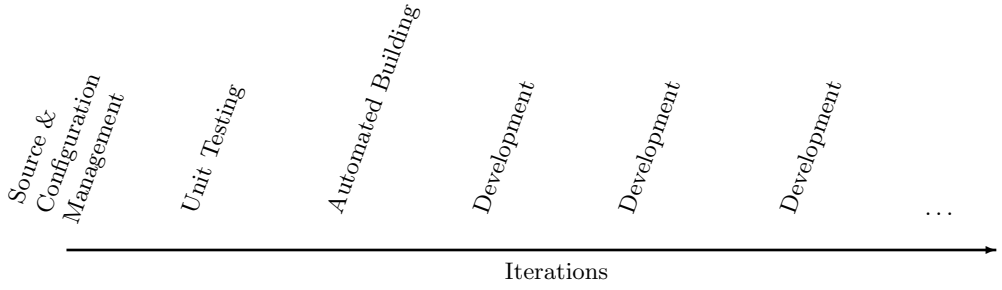

Fig. 2. Development practices are front-loaded onto our second year software development module. It is intended that students use these practices when they approach the development iterations.

Of the four topics covered we address SCM initially. Unit testing and automated building are covered before the development iterations commence, though we do not believe that there is an inherent advantage in covering unit testing before automated building or vice-versa. Each development iteration entails lectures which address a single software pattern [22] or small group of related patterns. Common patterns covered include

- Iterator,
- Factory Method,
- Adaptor,

– Composite,
– Decorator,
– Façade,
– Observer

We discuss patterns in lectures according to the needs of the project students are undertaking. For example, were the project to develop a Java SWING GUI we would cover the Iterator and Observer pattern during the first development iteration.

We suggest that there exists a class of problems which particularly suit this mode of teaching. Problems that involve a common communication protocol or developing a common library help to encourage communication and collaboration amongst students. For example, we have defined a simple chat protocol. All students are required to, independently, implement a graphical client for the chat protocol. Resolving ambiguities in interpretation of the chat specification and incompatibilities amongst clients requires students to debate the issues.

6 Conclusion

We do not claim that source and configuration management is, in itself, a conceptual threshold [23] or an example of computational thinking [24]. However, it does display aspects of conceptual thresholds such as *transformation* and *integration* and of computational thinking such as *conceptualisation* and *ideas, not artifacts*. By introducing SCM into a curriculum, software development is not reduced to the action of programming but is seen as a complex social operation. It allows learners to conceptualise the whole development life-cycle in small increments regardless of whether the engineering team (is supposed to) use the waterfall model, an incremental model or an agile model. SCM is an idea which is implemented in various ways in artifacts which facilitate learners to easily adopt best practices of software development. Understanding of the idea allows learners to adapt to each artifact. Though SCM does not promote communication in "our daily lives" [24] it solves communication issues in the daily lives of software developers.

We use SCM as a tool to introduce students to the social context of the software development process. We believe that we now have a good comprehension of SCM as a psychological tool and can develop our studies to investigate the effectiveness of introducing SCM as the core practice on our level two software engineering course.

References

1. Asiri, S.: Open source software. SIGCAS Comput. Soc. **33** (2003) 2
2. : The mozilla project homepage. (`www.mozilla.org`)
3. : The openoffice.org homepage. (`www.openoffice.org`)
4. : The linux kernel project homepage. (`www.kernel.org`)

5. Cusumano, M.A.: Reflections on free and open software. Communications of the ACM **47** (2004)
6. Hubbard, J.: Open source to the core. Queue **2** (2004) 24–31
7. Beck, K.: Extreme programming explained : embrace change. Addison-Wesley (2000)
8. Linder, S.P., Abbot, D., Fromberger, M.J.: An instructional scaffolding approach to teaching software design. In: CCSC 2006: Consortium for Computing Sciences in Colleges. (2006)
9. Raymond, E.S.: The Art of UNIX Programming. Addison-Wesley (2003)
10. Hunt, A., Thomas, D.: The Pragmatic Programmer from journeyman to master. Addison-Wesley (2000)
11. Kernighan, B.W., Plauger, P.J.: The Elements of Programming Style. 2 edn. Mcgraw-Hill (1978)
12. Daniels, H.: Vygotsky and Pedagogy. Routledge (2001)
13. Daniels, H., ed.: An Introduction to Vygotsky. Routledge (1996)
14. Kozulin, A., Gindis, B., Ageyev, V.S., Miller, S.M., eds.: Vygotsky's Educational Theory in Cultural Context. Cambridge University Press (2003)
15. Allen, K.: Online learning: constructivism and conversation as an approach to learning. Innovations in Education and Teaching International **42** (2005)
16. Vygotsky, L.S.: Mind in Society: The development of Higher Psychological Processes. Harvard University Press (1978)
17. Wheeler, D.: Comments on open source software / free software (oss/fs) software configuration management (scm) systems. `http://www.dwheeler.com/essays/scm.html` (2005)
18. Vygotsky, L.S.: The instrumental method in psychology. In Wertsch, J.V., ed.: The Concept of Activity in Soviet Psychology. M E Sharpe Inc. (1981)
19. Lord, T. (`www.gnu.org/software/gnu-arch/tutorial/Checking_002din-Changes.html`)
20. Moen, R. (`http://linuxmafia.com/faq/Apps/scm.html`)
21. Lister, R.: Objectives and objective assessment in cs1. In: SIGCSE '01: Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education, New York, NY, USA, ACM Press (2001) 292–296
22. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
23. McCartney, R., Sanders, K.: What are the threshold concepts in computer science? In: Proceedings of the Koli Calling 2005 Conference on Computer Science Education. (2005)
24. Wing, J.M.: Viewpoint: Computational thinking. Communications of the ACM **49** (2006)