"Free Libre OpenSource Software" implications in Computer Science education

Andrea TRENTINI
Dipartimento di Informatica e Comunicazione, Università di Milano
Milan, Italy

Abstract

This article reasons about the positive influence "Free, Libre and OpenSource Software" philosophy has (or should have) on the approach to Computer Science education, it is based on the author's experience in teaching "Operating Systems" and "Programming" for quite a few years. The rationale can be summarized by the following: guided Piaget constructivism (human knowledge is continuously and iteratively refined by the interaction between an internal representation of the world and the direct experience of the world itself, helped by a facilitator) is recognized as a good mechanism for learning and should be favored; FLOSS (Free Libre OpenSource Software) is a software philosophy that strongly encourages interaction with the internals of any (free) system; thus a FLOSS *attitude* (involving both method and subject) should be incorporated in teaching. After a short chronology of "freedom" in the world of software and documentation this article will compare the two main teaching approaches (traditional and constructivist), analyze how FLOSS may help teaching and describe an object model explicitation to help students better understand their learning process.
Keywords: Free Software, Teaching, Learning, Educational model, Constructivism

1. INTRODUCTION

Teaching is not always easy. A teacher should combine many qualities... and knowledge about the subject taught is just one of them.
The author works as assistant professor at the University of Milano (Italy), Department of Computer Science, and has taught topics such as Object Oriented Analysis & Design, Operating Systems, Programming for a few years (since 1996). During these years his teaching style has changed, a lot. If such a change can be considered positive will be left to the judgement of his students and colleagues of course, but some considerations and reflections can be gathered here to be evaluated by the reader. The following thoughts strictly reflect the author's environment and may not be applicable to other countries or universities, it would be interesting to collect experiences from all over the world...
Any teacher has probably started his career as a "traditional teacher": someone who sees himself as special, someone chosen to represent knowledge, to be a master among would-be adepts, an authority deserving respect... and so on. A "top-down" teacher, referring to the prevalent direction of interaction, who *leaks* knowledge to his students at his pace.
But, soon, the very same teacher will face the difficulties of his role: preparation of course material, confronting many students (in Italy we may have classes with hundreds of them), exams, self motivation and students motivation, keeping up with the subject, many questions and, last but not least, trying not to loose authority over students.
Any student entering university has to face a completely new environment, a big switch from high school. He must learn a new way to study, he has more independence, he is less directed, he can self-allocate study resources, timeframes, exams order, etc.
It's a suitable time to help him in his learning career, attitude and method. A traditional teacher would ignore this aspect and would probably try to keep the "distance" between himself and the students as big as possible, seeing them just as receptacles, knowledge receivers. Maybe there is a different approach.
The rationale of this article is that since some form of Constructivism-based teaching/learning (see Sections 2 and 6) is the most effective way to transfer knowledge while the Free Software philosophy (see Sections 4 and 5) implies constructivism, the conclusion is that FLOSS (and a FLOSS attitude) should be widely used in teaching. Roles (teacher, learner) are changing, authority on any *freed* topic is more evenly spread, no one can declare himself "ultimate expert" since information is public and can be used openly. The educational approach should change towards a more open and abstract system/object model, towards the *reification of a meta model* in the process.
This article argues that the teaching process should constructively take into account this difference in object models: capitalize it by reifing the model during teaching and giving students the means to "learn to learn". It is a change in *teaching style*: in classroom the model should be explicitly explained

and the role of the teacher should drift from "teacher of contents" to "teacher of methods" (with an explicit and dynamic object model). Of course, this meta-method is already exercised by any good teacher... this article wants to assess that FLOSS environments naturally ease this transition.

2. LEARNING AND CONSTRUCTIVISM

One of the definitions of Constructivism is the following: "a theory of knowledge (epistemology) that argues that humans generate knowledge and meaning from an interaction between their experiences and their ideas." Jean Piaget[10] (`http://www.archivesjeanpiaget.ch`) formalized it at the end of the fifties, according to this theory every person grows knowledge through the processes of *accommodation* and *assimilation*. *Accommodation* is the mechanism by which failure leads to learning: when we expect some behavior from a system we are examinating and the expected behavior does not happen we say that we have failed, our mind failed, so that we need to *accomodate* (reframe) our model of the system. i.e. we refine our model on failures. *Assimilation* occurs when our mind model does not fail on the real system, in this case we are fixating our model more and more. Constructivism is usually associated with teaching approaches promoting active learning, or learning by doing, by trial and error. Studies such as [6] found that in academic environments some form of constructivism influences positively the learning process, above all for the motivation injected into students. Another study [3] found that constructivism brings "better retention". Pure constructivism (without external intervention) is seen as not effective[7], some form of help from a teacher or a "facilitator" is always needed. "According to the social constructivist approach, instructors have to adapt to the role of facilitators and not teachers"[1]. Some also argue that the responsibility of learning should reside increasingly with the learner[14]. And "Social constructivism"[9] emphasizes the importance of the learner being actively involved in the learning process, unlike previous educational viewpoints where the responsibility rested with the instructor to teach and where the learner played a passive, receptive role. Although some criticism[5] arose in the pedagogical community all the researchers agree on the fact that a good shift away from the traditional/top-down/from-pedestal teacher is needed towards a more modern and efficient new/bottom-up/social facilitator.

"A teacher tells, a facilitator asks; a teacher lectures from the front, a facilitator supports from the back; a teacher gives answers according to a set curriculum, a facilitator provides guidelines and creates the environment for the learner to arrive at his or her own conclusions; a teacher mostly gives a monologue, a facilitator is in continuous dialogue with the learners"[12].

3. THE STORY SO FAR...

From the beginning[8], Computer Science has greatly evolved (`http://en.wikipedia.org/wiki/List_of_timelines#Computer_science`) from both the theoretical and the applied points of view. Also from the legislative point of view, many changes, not always positive (goal was meant to be positive but actual use is distorted), have been introduced, such as copyright, patenting and licensing.

During the **early eighties** a pioneer created a legislative innovation that influenced, and still influences today, all of us: the **copyleft** (`http://www.gnu.org/copyleft/copyleft.html`) concept. Copyleft gives rights to users instead of taking them away. A "copylefted" product can be used, studied, copied, modified and redistributed (with few constraints) over the net (often) for free. This innovation has not, however, disrupted the world of software... On the contrary, it has taken (and still has to) a long time to spread and pervade.

In the beginning of the *copyleft* era almost nobody outside a small fraction of highly involved people ("geeks" - `http://en.wikipedia.org/wiki/Geek`) even knew about this innovation.

In the **nineties**, with the advent of Web and GNU/Linux, more people began to know about "free" software since they could use it on their own computers, but they were still very few while others were stuck on MSDOS and early version of Windows... and most of the software was still **bought at high prices** and came with documentation in heavy printed manuals.

The first quantum leap began with the **new century**, when the first "easy" or "end-user" distributions (collection of a free operating system, applications and easy installer software) appeared on the market. These products were, from the users' point of view, evenly comparable at worst and far better at best with their "proprietary" counterparts (MSDOS, Windows, MacOS, etc.).

Today, again from the user point of view, the distinction between (almost) any proprietary product and its "open" counterpart resides (almost) only in its... license (and price, because the "free/open" product is usually also free).

4. FREEDOM OF SOFTWARE

Author's hope is that a section of this kind will no longer be required in the near future, as more and more people will know and understand what Free Software is. Any reader already aware about Free

Software may skip to the next section.
Free (as in "Freedom") Software (`http://www.gnu.org/philosophy/free-sw.html`) is a way of giving rights to users. Users of Free Software have:

0: The freedom to run the program, for any purpose.
1: The freedom to study how the program works, and change it to make it do what you wish. Access to the source code is a precondition for this.
2: The freedom to redistribute copies.
3: The freedom to distribute copies of your modified versions to others. By doing this you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this.

The two more interesting freedoms for the purpose of this article are #0 and #1.
#0 lets the user run the program any times he wants, in any condition (purpose), so that the program can be tested at will, thus fully reverse engineered (if there is no interest for the source code) without any legal limitation. By contrast, there are many proprietary EULAs (End User Licence Agreement) that explicitly prohibit reverse engineering practices (`http://en.wikipedia.org/wiki/Software_license_agreement#Reverse_engineering`) or even study the performance of a program.
#1 lets the user study the internals of the program without even running it, i.e. by reading the source code, thus giving the user the ability to fully analyze it and to "construct" an internal representation in his mind (more in Section 6).
The fourth (#3) freedom is also important because of **motivation**. The possibility to modify the program and then redistribute it is a good incentive, albeit the reward is just a slice of fame and not money.

5. OBJECT MODEL

In computer programming "object model"[4] has a well defined meaning, here we will use it in a lighter way, i.e.: "a collection of objects (with attributes, behavior and relationships, often described in a formal language) representing a system to be described in an abstract way".
From the **educational approach** point of view, there's an interesting distinction in **"object model"** between open and closed products. Different licensing implies a different set of objects. Even the matching names in the two sets may have slightly different meanings or importance, e.g., the "manual" in open products may be more outdated than in the case of closed software. For the sake of this article it is enough to define a list of objects related to the context of computer science education with a minimal description, because the interesting aspects are the differences between the world of proprietary software and the free one, differences in:

- meaning
- source of authority (*battled* between the producer and the users)

The following list is not complete of course, it is just a "start of discussion", it is also not related to a particular product in mind, the author tried to think as generally as he could. The list of definitions below has the following structure/format:

term: general definition
libre: applicability and semantics in FLOSS
prop: applicability and semantics in proprietary

Definitions are taken mainly from wikipedia or wiktionary.

author/owner (person): "The originator or creator of a work, especially of a literary composition." - in software is the person (programmer) who actually writes code
libre: the author is the one holding the copyright
prop: usually not known, copyright is held by the firm who pays the programmers

owner/producer/distributor (firm): an entity that creates goods and services and sells/delivers them to customers
libre: sometimes nonexistent if the author delivers by itself (usually through its own website), sometimes an organization (often nonprofit) that packages (with an installation program) collections of software to be distributed (e.g. `http://debian.org`), often by download
prop: the firm that owns the brand (MicroSoft for Windows, Oracle/Sun for Java) and physically packages and sells a software product (and maybe support services) to customers

version: "A specific form or variation of something."
libre: programmers set the version number
prop: not related to the revision of source code, often not even a number but a name, usually decided by the marketing division to be "appealing" (such as "Vista") for customers

source code: text, written in some programming language, that originates the actual program (binary) to be run in a computer
libre: completely available
prop: not available, the end user cannot even know the original programming language of the product

binary program: the final form of a program, tho only one understandable by a computer **libre:** can be generated from the sources
prop: the only form of the product available

license: textual agreement describing what rights are given to the user of a software product
libre: they usually specify what the user **can** do with the product (see also section 4)
prop: also called EULAs (End User Licence

Agreement), they usually specify what the user **cannot** do with the product

course: a learning program, sequence of lessons

libre: anybody can study enough to build a course about a product, usually no certification is involved, authority of source is defined by actual content/ability/etc.

prop: the producer is usually also the supplier of courses about a software product and it often set up a certification program for the would-be teachers/experts

manual: a (possibly printed) document that explain how to use a particular product

libre: sometimes the documentation of free products is less exaustive than for proprietary ones, there is a tendency to push towards the "hands-on" approach or other forms of documentation (forums, FAQs, etc.)

prop: manuals are often the "presentation" of the product, they are elegantly packaged and are usually accurate

knowledge base (KB): an "automated" system to collect and retrieve systematic information about a product/system/context

libre: this term is almost never used, see below ("forum", "wiki", etc.)

prop: many big/complex products generate problems and questions from the users, software producers sometimes create and manage these KBs to solve problems once and for all, one notable example is the Microsoft KB (`http://support.microsoft.com`)

frequently asked questions (FAQ): a less "automated" (it's almost always a static web page) system (than KB) to collect and retrieve systematic information about a product/system/context

libre: usually compiled by the user themselves (see "wiki" below)

prop: preferred to a KB for smaller products and for a narrower user base

forum: a (virtual) place for discussion, usually organized in topics and subtopics - there is no technical difference in forums **about** libre or proprietary products, but some distinction should be made in perceived authority depending on who is hosting the forum: if the host is also the author/producer of the product the perceived authority is **usually** higher

wiki: a collaborative website, editable by the readers themselves - same consideration as "forum" above

bug: a "defect" (deviation from expected behavior) in a software product - the meaning is the same in the two worlds, **but**:

libre: a bug may be corrected by any user able to read and modify the source code - the discovery of a bug is seen as a POSITIVE fact: one of the advantages of FLOSS is that "given enough eyeballs, all bugs are shallow"[11] - the author THANKS you if you send him the full description of a bug you found!

prop: a bug can be notified to the producer in the hope it will be corrected - the discovery of a bug is seen as a NEGATIVE fact: bugs depreciate any product and the producer does not want to let users about the "bugginess" of its product - there are even some cases in wich exposing bugs is considered a criminal act (cryptography algorithms, cellphones: see `http://lwn.net/Articles/368861`) - you may even be arrested if you publish a bug description!

bug tracking system: a software system to keep track of bugs - the meaning is the same in the two worlds, **but** the associated information may be very different:

libre: the user who notifies a bug may also suggest where in the source the bug may originate

prop: the user can only describe the sequence of operations that brought out the defect, no clues about the origin

price: cost to get possession of something - the meaning is the same but:

libre: almost always free, voluntary donations are often solicited, also notifications of bugs (see above) or suggestions for better code are welcome - professional support is not free

prop: almost always > 0, sometimes a free (but stripped down of functionalities) version is available for download as a "try before you buy" item - professional support may be included in the price of product

There is one last object in the model that is only somewhat part of the model itself, it is an aspect of many items listed above. The questions are: "Who has authority over a particular object?", "Where this authority comes from?", "Can this authority be trusted?". We may define authority such as:

authority: an entity **accepted** as a source of **reliable information** on a subject

libre: can be anyone, anyone can study a product and prove to be expert by demonstrating it in the *battlefield*, his/her knowledge is always verifiable (remember you can study the source code!)

prop: can be almost anyone, BUT his/her authority derives directly from some kind of certification by the owner/producer/distributor and his knowledge can be verified up to a certain border: the industrial secret border - anything that resides beyond this border cannot be verified (maybe partially through reverse engineering) and must be taken from granted, you must believe the product owner - a very famous example is the case of Windows suspected of sending private data back to MicroSoft servers, users unaware...

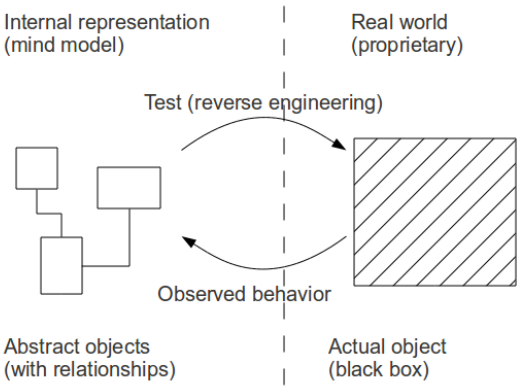

Figure 1: Constructivism: proprietary context

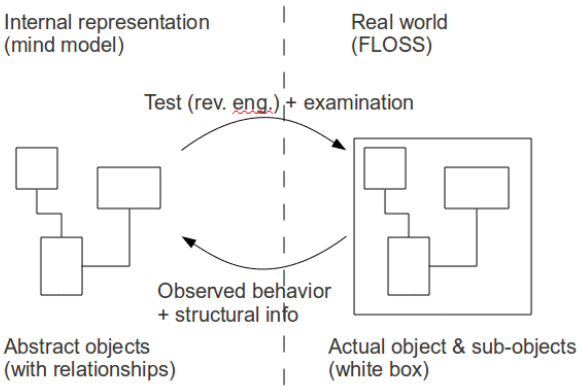

Figure 2: Constructivism: FLOSS context

> if a MicroSoft expert/teacher is asked about this "feature" he will tell you the "official truth" MicroSoft has told him, so you will have to trust the complete chain of information

As you can see in the case of FLOSS authority is questionable, **verifiable**, dynamic while in the case of proprietary software authority must be trusted, the only choice you have is whether to use the product or not. Of course this is an exaggeration, an extremization, but it summarizes the opposite directions, trends, tendencies of FLOSS and proprietary: the former is based on verification, the latter on "secretization".

6. CONSTRUCTIVISM AND FREE SOFTWARE

Following the definition of constructivism given in Section 2 we may depict a schematic diagram of the learning process, in Figure 1 and in Figure 2 the reader will find the author's interpretation. The goal of knowledge is to create a mind map of a "thing" (a physical object, a software product, anything) we want to know so that we may reason about it, expect some kind of behavior and in general usefully interact with it.

On the **left side** of the diagrams there is an "internal representation": a set of abstract objects that we can play with in our brain, this set of objects should mimic the real thing as closely as possible if we want to know the real system. The **curved arrows** on both figures represent the possible interactions/connections between the mind map and the real system. On the **right side** of Figure 1 is depicted a black-box object, since in the proprietary world no internal details are available. The only way to *know* a black box-object is by stressing/testing it, i.e. reverse engineer[2] it, by trial and error: your mind, based on the current abstract map, hypothesizes an (expected) behavior so that by actually trying it on the real object you can verify your mind's hypothesis. By iterating this cycle many times your mind should at last create a satisfactory internal representation of the object. **How many iterations** are necessary to reach a correct model (if a correct model can even be reached...)? Probably not many for simple objects (e.g. a pair of scissors), but what about a complex system such as a word processor or a programming language compiler? The reader may argue that some kind of documentation (user manual, techincal manual) should be available in complex cases. Of course, but who can check document correctness in a proprietary context? On the **right side** of Figure 2 is depicted a white-box object, since in the FLOSS world all internal details are available. You may still use a reverse engineering approach to create your mind map, but at any time you can study the source code to actually see the real object structure. **It's up to the learner.** Of course any teacher can propose his own model, but he will have to let students compare it to the model they can create by themselves while freely studying the inner details of the object (software system) under examination.

An example of this attitude, closely tied to the teaching of Java, happened to the author. One day a student asked about the internals of the java.lang.Vector class (sources of all library classes are available), the author had only a partial answer. In a proprietary context a smart teacher could have packaged a FUD (Fear, Uncertainty & Doubt) response, pretty sure about the impossibility to be contradicted.

The FLOSS attitude led the author to the explicitation of the whole process: formulate a behavioral hypothesis (the *add()* method execution speed is not constant, from time to time it takes longer because the inner array has to be cloned), test it black-box

and also check it in the source code. Students were impressed and they demonstrated afterwards to have learned the lesson (method).

7. METHOD PROPOSAL

"Do not believe what I say." This is the first phrase the author pronounces at the beginning of the first lesson of every course. It is meant to shock students out of their prejudices about professors and encourage them to verify information. The goal is to lower the "authority gap" to help *participate* in the learning process. The next thing they hear is: "Please do make questions! The worst thing that can happen is that I don't know the answer... but I'll have the meta-answer!", i.e. the teacher presents himself as a facilitator, one who can help in the constructivist process of learning. Then they are **presented with the differences between proprietary and FLOSS** worlds and they are explained why they will be offered to study mainly FLOSS artifacts (motivations in Section 6). Then the course flows in what may appear as traditional (front lessons, website for materials, etc.) but it is not.

Front lessons in classroom are supported by the use of a collaborative editor (gobby - `http://gobby.0x539.de`) where any student can edit the very same source code the teacher is showing, and in fact they do, a lot, even anticipating what the teacher is explaining (sometimes the author has to stop them by voice because he wants to explain the whole sequence of common mistakes in programming before getting to the correct solution).

The course website is in fact a wiki (`http://en.wikipedia.org/wiki/Wiki`) so that any student (even anonymously) can edit, add, remove, pages.

They are also encouraged to **interact with the teacher** (but also among them) by being offered many communication channels, such as email (of course), chat (Msn, Facebook, skype), microblogging (twitter, identi.ca). The choice of many "modern" channels tries to lower the "energy barrier", so that everyone can participate, the hope is that any student can find his suitable way of interaction. And they do: some of them use classical email, some twitter, some chat, some ask during/after lesson, etc.

Please be aware that this is not a (criticized[7]) "minimal guidance" or "pure discovery-based teaching technique" approach since students are not "left by themselves to try and understand", they are well guided, they are protected from "misconceptions or incomplete or disorganized knowledge"; author's goal is to let them be as active (in the learning process) as they can, inside a coherent knowledge framework. Is there any problem with this approach? Yes of course. Teachers must jump down from pedestal and let students participate, somewhat loosing their "given" authority while deserving it "in the field" (i.e. proving to be a reliable source). Moreover, in the FLOSS world knowledge runs fast[13], so the new teacher must keep up with the pace or his students will rapidly overcome his knowledge and he will loose respect, i.e. he must run (metaphorically) to remain in the same place.

References

[1] H. Bauersfeld. *Constructivism in Education*, chapter The Structuring of the Structures: Development and Function of Mathematizing as a Social Practice. Lawrence Erlbaum Associates Publishers, 1995.

[2] E.J. Chikofsky and J.H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE software*, pages 13–17, 1990.

[3] M. Dogru and S. Kalender. Applying the Subject "Cell" through Constructivist Approach during Science Lessons and the Teacher. *Journal of Environmental & Science Education*, page 11, 2007.

[4] Martin Fowler. *Analysis Patterns*. Addison-Wesley, Boston, 1997.

[5] J.H. Holloway. Caution: constructivism ahead. *Educational Leadership*, 57(3):85–86, 1999.

[6] J.S. Kim. The effects of a constructivist teaching approach on student academic achievement, self-concept, and learning strategies. *Asia Pacific Education Review*, 6(1):7–19, 2005.

[7] P.A. Kirschner, J. Sweller, and R.E. Clark. Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational psychologist*, 41(2):75–86, 2006.

[8] Steven Levy. *Hackers: Heroes of the Computer Revolution*. Dell Publishing Co., Inc., New York, NY, USA, 1994.

[9] C.H. Liu and R. Matthews. Vygotsky's philosophy: Constructivism and its criticisms examined. *Published by Shannon Research Press Adelaide, South Australia ISSN 1443-1475 http://iej. cjb. net*, 6(3):386–399, 2005.

[10] J. Piaget. *The psychology of intelligence*. Routledge, 1999.

[11] Eric S. Raymond. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly Media, Inc., revised & expanded ed. edition, 2001.

[12] L.K. Rhodes and G.T. Bellamy. Choices and Consequences in the Renewal of Teacher Education. *Journal of Teacher Education*, 50(1):17–18, 1999.

[13] Andrea Trentini. The Borg "architecture" as a metaphor for FLOSS. In *IADIS International Conference Applied Computing*, Rome (IT), Nov 2009. IADIS.

[14] E. Von Glasersfeld. Cognition, construction of knowledge, and teaching. *Synthese*, 80(1):121–140, 1989.