# HOKKAIDO,JAPAN

Conference Proceedings
January 19-21, 2017

ACENS

Asian Conference on Engineering and Natural Sciences

ICAS

International Conference on Applied Sciences

# Conference Proceedings

**January 19-21, 2017**

**Hokkaido, Japan**

ACENS

Asian Conference on Engineering and Natural Sciences

ICAS

International Conference on Applied Sciences

**ACENS**

**Asian Conference on Engineering and Natural Sciences**

ISBN 978-986-5654-18-4

**ICAS**

**International Conference on Applied Sciences**

ISBN 978-986-5654-34-4

# ACENS-67543
# Interdisciplinary Connections of Software Development Education

**Evgeny Pyshkin**
Software Engineering Lab, University of Aizu, Japan
E-mail: pyshe@u-aizu.ac.jp

## Abstract

In this paper we analyze software development education considered as a natural convergence of many interdisciplinary efforts. Within the context of programming and computer science teaching agenda, connections of software related courses to the mathematical, natural science and liberal art disciplines are examined. We emphasized the particular aspects which are important at the very beginning stages of software education, including such topics as a transition from subject domain models to the software models, software changeability, understanding software as a creative manifold process involving both artistic and engineering activities, using problem-based learning for a case of programming, as well as usefulness of introducing code review practices in software design courses.

Keywords: Software, software education, programming, liberal arts, interdisciplinary

## 1. Background

Teaching and learning software development requires creating a context, where software related expertise interacts with and even penetrates to the wide range of technical and non-technical academic disciplines. Among various software curriculum disciplines (including programming, software engineering, informatics, requirement engineering, object-oriented analysis and design, to cite a few) our particular attention is paid to programming which is often mistakenly considered as an equivalent of all the activities related to computer usage, ranging from writing software to system administration. Despite computer science and technology are developed dramatically during recent decades, the problems of programming teaching are still much discussed, and a discourse might lead to quite emotional conclusions similar to “we still have only a vague understanding of why it is so difficult for many students to learn programming, the basis of the discipline, and consequently of how it should be taught” [1].

Professionals and university informatics and software engineering instructors would probably agree that there are particularities that programming courses have if we compare them to other fundamental academic disciplines. These particularities are conditioned by a huge variety of problems faced by a software teacher. Academic course objectives are manifold: teachers have to explain the foundation programming concepts; at the same time, they have to teach programming languages expressing and implementing the concepts differently; there are standards that change; there are development technologies and tools which, in turn, progress rapidly. We have to include such things as libraries, frameworks, subversion and project management solutions, bug tracking tools, software deployment and maintenance issues to the context of the course. So, for a responsible tutor, there is no other choice, but to revise the course constantly. In contrast, changes in other "traditional" academic disciplines might not be so dramatic even after fairly revolutionary events. Thus, works of Andrew Wiles on proving Fermat's Last Theorem or a proof of Poincaré conjecture by Grigori Perelman didn't change significantly the landscape of the university academic courses in higher mathematics.

There is another interesting issue: one programming problem might have many legal solutions (and many incorrect ones among them), so, in a sense, programming is closer to linguistics rather than to mathematics: psychologist Richie O'Bower noted that a programmer is rather a philosopher and a linguist all in one [2]. It is worthy of attention that recently there is a lot of works analyzing humanitarian connections of software engineering: the non-engineering connections of programming are strongly related to software aesthetic properties and its social significance [3]. As far back as 1972 Andrey Ershov remarked: "in its creative nature programming goes a little further than most other professions, and comes to mathematics and writing" [4]. Despite the fact that the software elegance and readability issues were widely discussed in "ancient" years of software, Wilson and Oram conclude that university students are rarely taught how to see the software elegance which is contrasting to the traditions existing in other creative fields like painting, plastic art or architecture [5]. Of course, beautiful solutions aren't obvious. Despite a concept of (code) beauty is very subjective, it could be considered as one of important properties for judging the software quality.

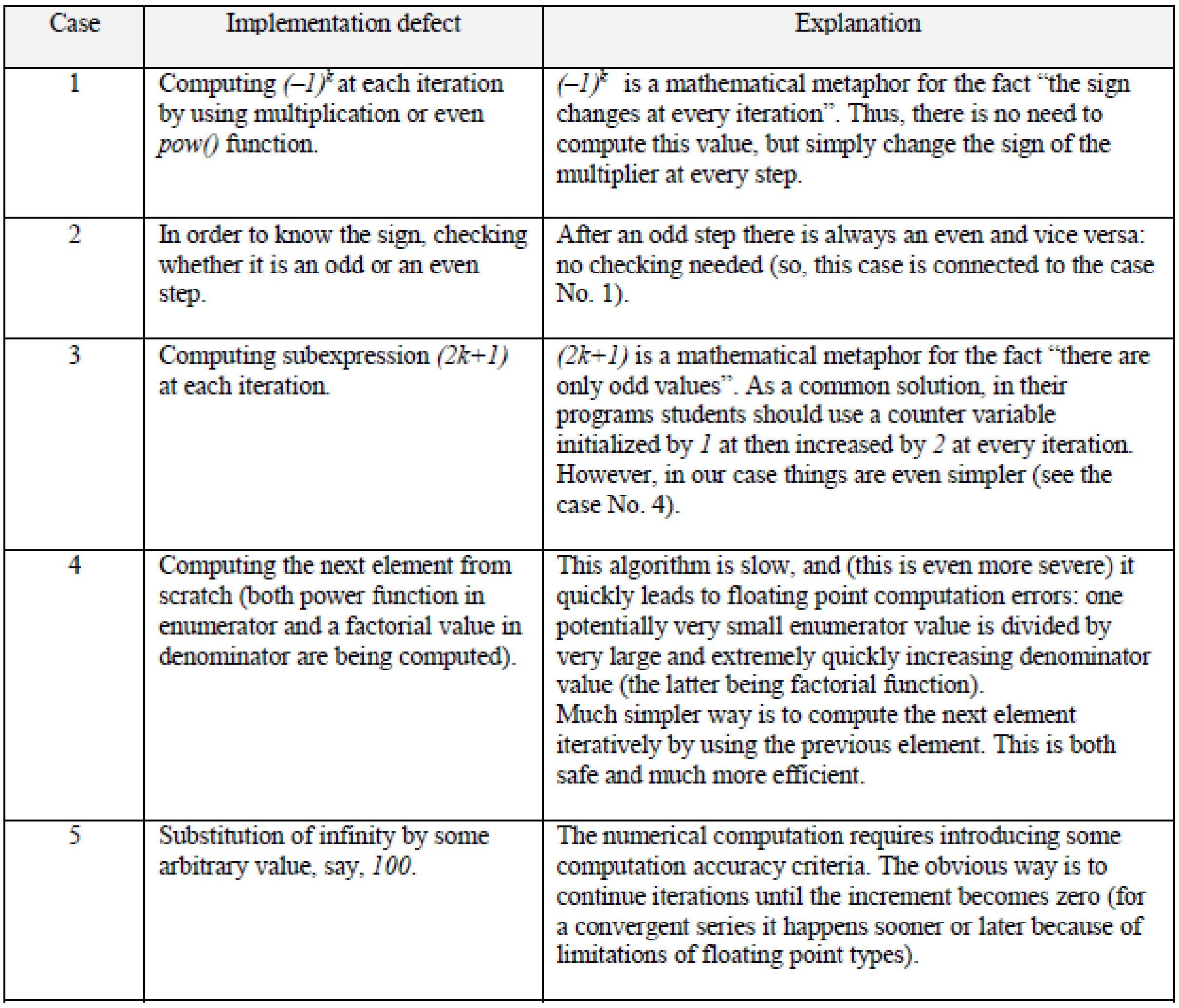
## 2. From a Subject Domain's Language to a Programming Model

In the very beginning lessons in programming, it is important that a teacher gives students primary understanding of how to translate the problem and solution description from a subject domain's language to a programming domain's one. It is true even if there is a formal discipline subject domain, like mathematics. In the CEE-SEC(R) 2011 presentation I used the Taylor series as a good example from the undergraduate course of mathematical analysis [6]. Here below the example is examined in more details. The problem of computing a series sum is mathematically clear, and almost every student is able to understand the following formula which is effectively a definition of the $\sin(t)$ function

$$\sin(t) = \sum_{k=0}^{\infty} (-1)^k \frac{t^{2k+1}}{(2k+1)!}$$

However, when my students attempt to write a computer program in, say, C language, most of them do some very common mistakes (see Table 1).

Table 1: From mathematical language to a programming model: students' mistakes



| Case | Implementation defect | Explanation |
|---|---|---|
| 1 | Computing $(-1)^k$ at each iteration by using multiplication or even *pow()* function. | $(-1)^k$ is a mathematical metaphor for the fact "the sign changes at every iteration". Thus, there is no need to compute this value, but simply change the sign of the multiplier at every step. |
| 2 | In order to know the sign, checking whether it is an odd or an even step. | After an odd step there is always an even and vice versa: no checking needed (so, this case is connected to the case No. 1). |
| 3 | Computing subexpression $(2k+1)$ at each iteration. | $(2k+1)$ is a mathematical metaphor for the fact "there are only odd values". As a common solution, in their programs students should use a counter variable initialized by *1* at then increased by *2* at every iteration. However, in our case things are even simpler (see the case No. 4). |
| 4 | Computing the next element from scratch (both power function in enumerator and a factorial value in denominator are being computed). | This algorithm is slow, and (this is even more severe) it quickly leads to floating point computation errors: one potentially very small enumerator value is divided by very large and extremely quickly increasing denominator value (the latter being factorial function).<br>Much simpler way is to compute the next element iteratively by using the previous element. This is both safe and much more efficient. |
| 5 | Substitution of infinity by some arbitrary value, say, *100*. | The numerical computation requires introducing some computation accuracy criteria. The obvious way is to continue iterations until the increment becomes zero (for a convergent series it happens sooner or later because of limitations of floating point types). |

Students make these "typical" errors mentioned in Table 1. However, a teacher shouldn't prevent them: it wouldn't be right if a teacher doesn't allow students to see why their solution might be unreliable or even incorrect. Only after making a mistake, students can understand completely the idea, and therefore learn the lesson.

## 3. Introducing Programming Course Aspects

In this section we examine some important aspects to be presented to students attending a programming course.

### 3.1 Target Audience and Target Language

Computer scientists and programmers are not the same professionals, so the course contents and organization in computer science (CS) and software engineering (SE) curriculum might differ significantly [7]. In particular, these difference might affect the choice of programming language used for a programming course. CS students require to get understanding of the basic data management and computation processes, while SE students have to learn a tool which doesn't only allow expressing existing ideas, but creating new mechanisms and models. SE students should be able to work being not limited by some programming paradigm. Finally, nowadays programming courses are being also taught to students with no computer system development background. They nevertheless require some basic knowledge of computer system usage: information technology is an important part of every modern curriculum in physics, mathematics, economics, design, medicine and even in low and arts.

## 3.2 Target Language

Choosing a right programming language is an essential aspect of a programming course. There is a common consideration that a language isn't of much importance, concepts are of much importance. In general, it is true. However, together with the requirements to be universal, a language used in a SE class should be not too far from the tools used in IT industry or/and in software research. An instructor should be able to explain to students, why a certain language is used for the course, why an instructor prefers to use this language, and, finally, why this language is interesting for learning. Following Dmitry Likhachev's considerations about natural languages, we could say that a teacher should discover for students an idea that there is a strong connection between a language and a way of thinking, between the form and cognition: terminology and grammar systems provide a foundation for human self-actualization.

## 3.3 Software is about Changes

In software development changes are essential. So even in the very first projects, tutors have to explain to students that expecting changes is a standard, necessary and inevitable situation in software design. Changes are conditioned by many obvious and less obvious reasons, including discovered bugs and defects, modified requirement specification, user feedbacks, changes in external libraries and third party components, etc. Since changes are natural phenomena, students have to learn how to write a modifiable code. It means that at the very beginning stages of their education tools and methods supporting software modifications (project management tools, version and bug control systems, software quality assurance methods) should be included to the scope of an academic course. Some experiments arranged with a group of students show that due to using subversion and *trac* systems the development and teaching process becomes more productive for several reasons [6]. First, students get more opportunities to communicate with the teacher outside the classroom. Second, the design process became more systematic and better organized. Finally, students start using tools which are similar to those used in software industry.

## 3.4 Problem-Based Learning and Multi-Aspect Tasks

Problem-based learning (PBL) originally came from the fields of knowledge not directly connected to informatics and programming (e.g. in medicine, linguistics, and natural sciences) [8, 9].

Within the PBL, students are expected to work in a team on projects close to real-world problems. PBL proponents believe that it leads to better student involvement in research, since it favors to investigating ways to find missing knowledge and necessary methods and tools, to justification of decisions made, as well as to developing communication skills and teamwork practices. All these factors aren't against using PBL elements in programming and other SE cycle disciplines.

One of the approaches that could be discussed as an implementation of PBL model in SE courses, is using multi-aspects tasks which require a combination of different models and methods [10]. One possible example of such a task is developing a compiler for a small language, or at least its parts such as syntax and semantics analyzers for a language defined by its grammar. Discussing mechanisms which are necessary to design such a component would allow introducing some necessary units to be learnt in a course of informatics or programming, including formal grammar theory, lexical analysis, binary code, finite state machines, program structure visualization, data storage models, search algorithms, etc.

The inverted curriculum model is another approach focusing on similar targets [11, 12]. At the core of this approach there is an idea that the traditional software courses are organized in a way that begins with presenting elementary knowledge and low-level algorithms and data structures up to large project organization. It allows giving students a fairly deep understanding of the internal structure of programs and computational process organization. However, this approach does not always meet the requirements of training professional software development engineers who should be able to develop, modify and re-use software components, including legacy software. Instead of an approach "from simple to difficult" students are encouraged to immediately begin working with a large project and professional libraries. In process of studying the components students go into the details of their internal structure, examine the underlying mathematical models, algorithms and data structures.

## 3.5 Code Review as a Software Education Practice

Code review is a process of software source code examination performed by an expert aimed at discovering potential defects for comments, suggestions or approval. In process of code review a teacher might attract students' attention to the different functional and non-functional software defects including deviation from standards (code style, or internal company code organization standards and practices), architectural defects, memory and resource usage defects, code readability concerns, test coverage defects, improper usage of formally syntactically correct language constructions, security issues, documentation defects, error handling errors.

As "a discussion between two or more developers about changes to the code to address an issue" [13], code review can be considered as a practice used in software static analysis which fits well software development learning process. Code review is then referred not only to the source code but also to other readable artifacts like requirement specification, system architecture design documents, and testing scenarios. A teacher should be able to explain to students that even outside a classroom they must re-read the source code, and therefore be able to proceed with their own first formal inspection. As it was nicely noted by Wiegers, “if you are in an organization of two or more people, some kind of inspection activity should be a part of your standard software development process. If you work alone, well, at least read the code” [14]. However, Wiegers recommends to avoid style issues in code review “unless they impact performance or understandability (…) When a group is beginning to use reviews, there is a tendency to raise a lot of style issues that are probably not defects but rather preferences” [14]. However, we believe that in teaching practice even code style related discussion may be considered as a code review issue, since students have to be taught how to recognize and to use different styles properly.

## 4. Conclusions

As noted by Walker and Kelemen, having a symbiotic relationship with the liberal arts, computer science might be considered the ultimate of liberal arts disciplines [15]. As a counterpoint we could mention Knuth’s consideration that in medieval universities such disciplines as grammar, logic, arithmetic, geometry, music, rhetoric, and astronomy were taught among liberal art disciplines (it is no question that the first three are essential component of informatics) [16].

Being part engineering, part art, programming should interact with other disciplines, both technical and non-technical. Presently informatics and programming are increasingly becoming universal disciplines studied not only by future developers of software systems. That’s why development of the teaching methodology, improvement of learning models and learning paths with paying attention to significant differences in target audience remains one of major problems of university education.

## 5. References

[1] Malmi, L. et al. (2010). Characterizing research in computing education: a preliminary analysis of the literature. In Proceedings of the 6th international workshop on computing education research (ICER’10), ACM, New York, pp. 3–12, 2010.

[2] O’Bower, R. (1997) Programming as a best creative specialty.

[3] Pyshkin, E. (2014). In the right order of brush strokes: A sketch of a software philosophy retrospective. SpringerPlus, 2014, 3:186. DOI:10.1186/2193-1801-3-186.

[4] Ershov, A.P. (1972) Aesthetics and the human factor in programming. Communication of the ACM, 15(7): 501–505. DOI: 10.1145/361454.361458.

[5] Oram, A., and Wilson, G. (2007). Beautiful code: leading programmers explain how they

think. O’Reilly Media, Inc., Sebastopol, CA, USA.

[6] Pyshkin, E. (2011). Teaching programming: What we miss in academia. In Proceedings of the 7th Central and Eastern European Software Engineering Conference in Russia (CEE-SECR), pp. 1–6, 2011.

[7] Blake, J.D. (2011). Language considerations in the first year CS curriculum. Journal of Computing Sciences in Colleges, vol.26, no. 6 (June 2011), pp. 124–129, 2011.

[8] Gallagher, S. A. (1997). Problem-based learning: Where did it come from, what does it do, and where is it going? Journal for the Education of the Gifted, vol. 20, no. 4, pp. 332–362, 1997.

[9] (2011). Problem-based learning. Speaking of Teaching. Stanford University Newsletter of Teaching, winter 2001, vol. 11, no. 1, pp. 1–6, 2001.

[10] Pyshkin, E. (2014). Multi-aspect tasks in software education: a case of a recursive parser. International Journal of Advanced Computer Science and Information Technology, Helvetic Editions Ltd., Switzerland, 3(3), 2014, 282–305. ISSN: 2296-1739.

[11] Pedroni, M., and Meyer, B. (2006) The Inverted Curriculum in Practice. In Proceedings of the 37th SIGCSE technical symposium on Computer science education (SIGCSE '06), ACM, New York, NY, USA, pp. 481–485, 2006

[12] Meyer, B. (2010). Touch Class. Springer, 2010.

[13] London, K. Code Review Best Practices. Web: http://kevinlondon.com/2015/05/05/code-review-best-practices.html. Accessed: Oct 9, 2016.

[14] Wiegers, K. (1995). Improving Quality Through Software Inspections. Software Development, 1995. Web: http://www.processimpact.com/articles/inspects.html. Accessed: Oct 9, 2016.

[15] Walker, H. M., and Kelemen, C. (2010) Computer science and the liberal arts: A philosophical examination. ACM Transactions on Computing Education (TOCE), 2010, vol. 10, no. 1, p. 2, 2010.

[16] Knuth, D. E. (1974). Computer programming as an art. Communications of the ACM, vol.17, no. 12, pp. 667–673, 1974.