Applying Pantomime and Reverse Engineering Techniques in Software Engineering Education

Vladimir L Pavlov [1], Nikita Boyko [2], Alexander Babich [3], Oleksii Kuchaiev [4], Stanislav Busygin [5]

Abstract - **During the past six years, the authors have experimented with various modeling and quality control techniques while teaching software engineering to university students. The first group of experiments compared UML to natural languages (e.g. English); the second group modeled the entire software development process as a sequence of translations from more abstract languages to more formal languages (e.g. from English to UML, from UML to C++, etc.). In these experiments the authors and their students discovered new approaches to increasing productivity of software developers. On this basis, the authors have developed the framework that extends traditional development processes. The framework is built upon two simple, yet powerful principles: Reverse Semantic Traceability (RST) and Speechless Modeling.**

The authors have over two years of experience of using RST in computing curricula, and also have over six years of experience of applying Speechless Modeling in teaching computer science and software engineering. Both approaches resulted in improving the quality of education and generated positive feedback from students. Some students started to practice the authors' framework in their professional careers. In this paper, the authors present their experience of integrating RST and Speechless Modeling into university computer science/software engineering curricula.

Index Terms – Software Engineering Curricula, Pantomime, Speechless modeling, Reverse Semantic Traceability

CHALLENGES OF TEACHING SOFTWARE ENGINEERING

Many educational institutions face similar, major obstacles while teaching Software Engineering (SE) in their undergraduate and graduate programs.

One of these obstacles is that students cannot be convinced of the benefits of using software engineering techniques until they experience these benefits themselves [1]. The majority of software engineering techniques are suitable for use in large "real life" projects, and it is challenging to provide students with a realistic opportunity of working on such a project during an academic semester. Instead, students have to work on simple (even oversimplified) textbook examples [2] and, therefore, perceive software engineering as something which has only theoretical value. Consequently, there isn't any real motivation for students to study and even much less to use software engineering.

While the core SE techniques are focused on organizing team efforts, SE educators are faced with the problem that students become very accustomed to working completely alone during their undergraduate years, or have worked only in teams of up to three or, perhaps, four other students. The assignment of typical SE team roles to students is a challenge for both students and instructors - grade inflation is a serious issue in this case, because all team members receive the same grade while the amount of work done by each individual may vary significantly [2]. In fact, SE instructors spend too much time monitoring and enforcing the work within the team instead of concentrating on SE principles and methodologies.

With the understanding of these challenges to students and SE teachers, some new approaches and frameworks were proposed by various authors to solve these and other SE education problems. For example, Stiller and LeBanc [1] recommend utilizing the following principles as guidelines while designing SE curricula:

- Make SE real
- Make SE fun
- Make SE critical
- Make SE accessible
- Make SE successful
- Speak with a uniform, consistent voice

Similar approaches were proposed by other authors. Sometimes SE educators even recommended simulating problems for students during the capstone project so students could get an experience similar to real life. For example, Dawson [3] proposes to apply so called "dirty tricks" to the students working on the software project, such as:

- Give an inadequate specification
- Make sure all assumptions are wrong
- Have conflicting requirements and pressures
- Give additional tasks to disrupt the schedule
- Change the deadlines
- Crash the hardware

[1] International Software & Productivity Engineering Institute, vlpavlov@intspei.com
[2] University of Florida, nikita@ufl.edu
[3] International Software & Productivity Engineering Institute, ababich@intspei.com
[4] International Software & Productivity Engineering Institute, akuchaev@intspei.com
[5] University of Florida, busygin@ufl.edu

1-4244-1084-3/07/$25.00 ©2007 IEEE

The true value of these "dirty tricks" is that they are very common challenges that software engineers regularly face during their professional career. Therefore, providing a good education in Software Engineering really means preparing students to overcome these and many other similar "dirty tricks" while working on their real life projects.

A capstone project with elements of tough reality is obviously the best drilling exercise. However, such training in academia requires an experienced instructor and an environment that is as close as possible to the industry. In addition, a few weeks should be available in students' schedules for such an activity.

Since SE has become a recognized engineering discipline and all the major problems of SE education are very common for universities, over time the academic community has developed Software Engineering Curriculum Guidelines [4]. These guidelines include recommendations for those who develop and teach the curriculum. Some of the guidelines are:

- Curriculum designers must strike an appropriate balance between coverage of material, and flexibility to allow for innovation.
- The underlying and enduring principles of software engineering should be emphasized, rather than details of the latest or specific tools.
- In order to ensure that students embrace certain important ideas, care must be taken to motivate students by using interesting, concrete and convincing examples.
- Software engineering education in the 21st century needs to move beyond the lecture format: It is therefore important to encourage consideration of a variety of teaching and learning approaches.
- Important efficiencies and synergies can be achieved by designing curricula so that several types of knowledge are learned at the same time.

However, these guidelines are quite general, and implementing them in the curriculum is a challenge for SE educators. In this paper the authors present innovative, practical techniques aimed to help implement these guidelines. The new framework developed by the authors gives students the opportunity to gain knowledge and skills that normally require several weeks (or even months) of industrial internship in a one-day training session.

EVOLUTION OF THE P-MODELING FRAMEWORK

The P-Modeling Framework [5] originates from "The Babel Experiment" designed by Vladimir L. Pavlov in 2001 as a training program for software engineering students. The Babel Experiment consisted in the following steps [6,7].

A team of students was assigned the task of designing a software system under very unusual circumstances. Namely, Unified Modeling Language (UML) was the only language allowed for communication while working on the task. Any verbal or written communication involving natural languages was forbidden. The intention of this experiment was to make the students work through communication problems typical in large software development projects and to provide them with the experience of applying UML to overcome these problems. However, the design session was presented to students as an experiment in determining whether UML is sufficient for communicating all information relevant to the software system design needs and also how efficient it is in expressing the intentions of designers when no other communication means are available. In other words, the students were to explore whether UML is "a real language" or not, and whether a project team can benefit from its ultimate utilization.

After the speechless modeling sessions, the modeling team had to present results of their work to invited guests. To intensify the students' experience and make the event really important for them, we usually invited architects and project managers from local software development companies (local employers), as well as software engineering instructors from local universities to attend the final presentation. After the presentation, the guests were asked to judge whether the team managed to create a solid model.

The experiment was repeated dozens of times in both academic and industrial environments and was always ultimately successful. In no single case did the participants fail to accomplish the assigned task by delivering a vague, incomplete, or obviously wrong design. On the contrary, they always succeeded in expressing their ideas, bringing them to the team's vision, and finding, generally speaking, "the common ground" while using nothing but UML. This always resulted in the successful development of the proposed system model. The experiment received strongly positive feedback from both the participants and the customers [6,7].

Once, accidentally, during a design session there were two independent teams working on the same task. Communication means of the first team was restricted to UML as described above while the other team was allowed to communicate in addition verbally, using a natural language. It turned out that the first (more restricted) team succeeded in performing the task with a greater efficiency than the other team. The UML diagrams created by the first team were more sound, detailed, readable, elaborate and elegant. It strongly suggested that there is much more "uncharted territory" in applications for UML than previously assumed. Vladimir L. Pavlov conducted a number of additional experiments intended to reveal whether the "silent" modeling sessions are more productive than the traditional sessions. In all these experiments, silent teams were at least as efficient as traditional teams, and in most cases the silent teams clearly outperformed the traditional teams.

Our interpretation is that the crucial reasons for such a boost of performance are the following:

- The restriction on using a natural language stimulates creativity of the designers, as well as forces them to stay focused on their job;
- Work in speechless mode forces designers to explicitly uncover all underlying assumptions at the very early stages of the design process;

- UML is no longer treated as the "write-only" language – instead, the designers start to care a lot about the readability and quality of their models to make sure that their speechless peers understand them.

These experiments were designed with educational and research goals in mind, however, they encouraged several software development companies to incorporate speechless modeling into their Software Development Life Cycles (SDLCs).

Meanwhile, Vladimir L. Pavlov came up with further experiments intended to compare UML to natural languages. The idea was to set up forward (from a natural language to UML) and backward (from UML to the natural language) "translation" tasks for two teams of professional designers (with one team performing the forward translation and the other one performing the backward translation), and to observe how much the outcome of the backward translation resembles the original text. If we performed such an experiment with two natural languages (say, English and Russian) using two teams of professional translators, it is obvious that the backward translation would create a text semantically very close to the original text (though it might have a different number of words or even sentences).

These experiments confirmed that, for information describing software systems, UML has sufficient expression power to maintain the content. Similarly, as expected for the forward-backward translation experiment involving natural languages, the texts obtained after the backward translation from UML were almost semantically equivalent to the original.

However, the most prolific conclusion of these experiments went far beyond the linguistic matter. The experiments suggested the model of the entire SDLC as a series of translations. Indeed, when a business analyst communicates with a client, he/she translates the requirements formulated by the client from a natural language to a business model. Then, architects translate it into a design expressed in a modeling language, and programmers translate this design into source code written in a programming language. Finally, a compiler creates an executable having the source code as the input. Each of these steps usually introduces some new misinterpretations, or even errors. Unfortunately, most of the existing testing techniques are based on validation of the very final result – the executable. Why not to try to extend the testing to the other "translations" created throughout the development cycle? The backward translation verification comes in very handy here as the method that guarantees the deliverables of the current step do not lose or misinterpret anything that was dictated in the previous step. An obvious idea is to employ an independent "traceability testing" team(s) that receive(s) intermediate deliverables of each step as soon as they are created, translate(s) them back to the language used for the input artifacts of the current step, and then compares the restored artifact with the actual ones. If no information is lost or misinterpreted, then the development should proceed to the next step. Otherwise, it is necessary to correct the deliverables of the current step. This methodology has been named "Reverse Semantic Traceability".

Typically, the most expensive errors are introduced during the analysis and design process, so it is especially critical to test the models created during the analysis and design process as soon as possible. Therefore, the professional analysts and architects, who participated in the experiments described above, were particularly inclined to implement Reverse Semantic Traceability in their companies. As the development process became easier, the overall duration of the software production cycle became shorter and the quality of the created software significantly improved.

Based on the feedback received after all these experiments, the authors developed a one-day design and training event that integrates both techniques described above (Reverse Semantic Traceability and Speechless Modeling). This event was called a P-Modeling Session. It involves two independent teams working in parallel on two different assignments. They start with designing the job in Speechless Modeling mode, but then they exchange their models and perform Reverse Semantic Traceability for each other. Then both teams correct/improve their models upon completion of Reverse Semantic Traceability. Finally, they present their results to each other and invited guests (see Table 1)

TABLE I
APPROXIMATE SCHEDULE OF A P-MODELING SESSION

TEAM A	TEAM B	DURATION	SPEECHLESS
Introduction, ice-breaking		1 hour	No
Speechless work on modeling assignment A, creating the first version of model A	Speechless work on modeling assignment B, creating the first version of model B	3-5 hours, includes speechless lunch	Yes
Reverse Semantic Traceability for model B, created on the previous phase	Reverse Semantic Traceability for model A, created on the previous phase	1 hour	No
Analyzing results of the traceability session, conducted by Team B; creating the second version of model A	Analyzing results of the traceability session, conducted by Team A; creating the second version of model B	1 hour	No
Final presentations, session closing		1-2 hours	No

EXPERIENCE OF USING P-MODELING FRAMEWORK IN SOFTWARE ENGINEERING EDUCATION

The first experiments of using Speechless Modeling were organized in 2001; then Reverse Semantic Traceability was added in 2005. Some of the experiments were organized for professional architects/designers and were collocated with industrial conferences. However, most of the experiments were organized for students in universities and were integrated into the educational process. Initially, every experiment was organized somewhat differently. Later, after the P-Modeling Session was developed, all new events of this kind were

organized in the unified way. So far the P-Modeling Sessions have been incorporated into CS/SE curricula in four Eastern-European universities: Dnipropetrovs'k National University (Ukraine), University of Nijniy Novgorod (Russia), Kharkiv Polytechnik University (Ukraine) and University of South Ural (Russia). P-Modeling Sessions were never a required part of curriculum; there was always an option for students to choose an alternative activity. Usually 30%-50% of students choose to participate in a P-Modeling Session. As of today, 172 students have completed such sessions. All of the participating students were undergraduates in their third, fourth or fifth year of studying Software Engineering or Computer Science.

It is worth mentioning that 100% of those who participated in the P-Modeling Sessions said that they would want to participate in such events again and/or to organize such events in their professional practice in the future. All the participating students said they would recommend attending such sessions to others. Also, 100% of the participants assess Reverse Semantic Traceability as an extremely powerful tool to validate software design and want to use it in their practical work in the future. 93% of all participants consider Speechless Modeling a powerful tool for learning Object-Oriented Analysis and Design with UML. Some advantages of the Speechless sessions are: 90% of participating students mentioned improving of their team working skills and 71% of participants think that such sessions can help new team members to understand the domain area more quickly than traditional approach. Almost half of the participants said that these Speechless sessions taught them to create more precise models more quickly.

To give readers a feeling on what happens during the P-Modeling Sessions, below we provide two sample texts (see Tables 2 and 3), as well as examples of feedback from students (Table 4).

TABLE 2
SAMPLE DOMAIN DESCRIPTION USED AS AN INPUT FOR A P-MODELING SESSION

A research expedition will be sent to Outer Space in 2030. The mission will last very long and it will not be possible to equip the astronauts with the regular medical gear because of the physical constraints. Therefore it was determined to use self-learning nanorobots as the only diagnostic and medical treatment means. After appropriate training, the nanorobots can be used to cure almost any disease.

Depending on the condition of the patient and the type of illness, these nanorobots can be implanted into the human body by many ways, such as taking a pill, by injection, putting on the skin, etc. Upon the invasion into patient's body, the nanocomputers find the target organ and treat it as a self-organized group.

The nanocomputers can apply chemical drugs, which they are able to synthesize, as well as physical impacts such as mechanical, thermal, etc. During the treatment, they collect information about the effectiveness of various treatment modes, illness specifics, etc. The nanocomputers are equipped with transmitters whose broadcast range is very low. Therefore the collection of the data is performed by a portable device attached to a patient.

Periodically, depending on the disease (for example once per day or every 4 hours), this information is transmitted from the portable device to the server, which is located in the space ship ER. This server collects and processes the data received with the wireless connection. Another responsibility of the server is to train nanocomputers to treat various diseases.

The process of training nanocomputers to treat a certain illness consists of a few stages. A certain number of "pristine" nanorobots, which are not yet ready to treat the disease, are produced by chemical and mechanical technologies. These nanocomputers are embedded with self-training algorithms. Then the nanocomputers are placed into the training environment which is controlled by the server. During several hours they develop the ability to treat a specific decease. After this the effectiveness of the training they received is tested if possible. "Pristine" nanocomputers can be trained to treat any illness.

TABLE 3
SAMPLE DOMAIN DESCRIPTION CREATED AS AN OUTPUT FROM THE P-MODELING SESSION

A new crew health-monitoring and disease-curing system will be installed in outer space exploration ships. Therefore the medical nanocomputer engineers are introduced into the crew. Every space ship is equipped with servers that collect and systematize the information from the patients' portable devices, and improve the process of treatment. The portable device is connected with "pristine" nanocomputers that are embedded into human body and can receive the information that they send about the sick organs, the type of the decease and its progress. The server also controls and navigates the self-organized nanocomputers. The output data is the information about the disease and the means to cure it.

An engineer creates the "pristine" generic nanocomputers. The creation of a nanocomputer is a chemical or mechanical process. The server trains the "pristine" nanocomputers by putting them into specific environment of a certain disease, where they are self-trained and tested. As a result, we obtain specialized nanocomputers which can be used for the treatment of that specific illness. In order to treat the patient, the nanocomputers penetrate the human body where they can organize themselves into groups, find sick organs and treat the patient thermally, mechanically and chemically. Necessary substances can be synthesized during the chemical treatment. The nanocomputers also gather the information on the disease.

The employment of nanocomputers does not make traditional treatment methods (such as pills, injection and plaster) useless.

TABLE 4
SAMPLE STUDENT FEEDBACKS AFTER P-MODELING SESSIONS

"There was an atmosphere of playing a game: joyful and energetic. I liked that I had to work and think in a tough mode for a long period of time as if it was a contest. It was pretty much a challenge for the brain. I liked that I had learned some new things and tried them in practice."

"This is a wonderful means to build the team. It reveals leaders and creates more expressive design. Moreover, speechless mode encourages focusing on ideas, not on the words which explain those ideas. The absence of verbal communication promotes more effective information exchange and prevents repeating meaningless 'buzzword-bingo' words."

"Personal impression: communication restriction increased effectiveness of each participant and helped to focus on the model."

"The Speechless approach is very effective because it helps to

improve focusing - intensify thinking. It forces participants to comprehend the idea before they state it. It decreases human factor side effects such as useless discussions."

"It is quite possible to make a mistake during the work on system design. This can have a dramatic impact on the further development of the product. I think that Reverse Semantic Traceability (P-Modeling) helps to look into the problem from the alternative point of view, and to discover the mistakes and incompleteness made by the designer."

"Reverse Semantic Traceability is a very valuable method that helps to control the quality of diagrams and the designers' comprehension of the scope. It also allows formalizing the text version of the task, eradicating ambiguities and uncertainties, and clarifying what the customer really wants."

SUMMARY

In this article the authors have presented a practical and easy-to-implement framework that helps address some of the key challenges of SE education. The framework is based on using Speechless Modeling and Reserve Semantic Traceability, while organizing students' practical work. The authors have used Speechless Modeling since 2001, and Reserve Semantic Traceability since 2005 in their educational work. The overall feedback from students and faculty is extremely positive. Some students have started practicing the authors' techniques in their real-life work after graduation from a university.

ACKNOWLEDGMENT

The authors would like to thank Eric Gilliat, Anatoliy Doroshenko and Bekah Sondregger for their valuable comments that helped to improve this article.

REFERENCES

[1] Stiller,Evely; LeBanc, Cathie "Effective Software Engineering Pedagogy", *Journal of Computing Sciences in Colleges,* Volume 17, Issue 6, May 2002, pp 124-134.

[2] Bracken, Barbara "Progressing from student to professional: the importance and challenging of teaching software engineering", *Journal of Computing in Colleges,* Volume 19, Issue 2, December 2003, pp 358-368.

[3] Dawson, Ray "Twenty Dirty Tricks to Train Software Engineers", *Proceedings of the 22nd international conference on Software engineering, 2000*, pp 209-218.

[4] "Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering"*, Computing Curricula Series,* August 2004.

[5] http://www.intspei.com/products

[6] Pavlov, Vladimir; Yatsenko, Anton "Using Pantomime in Teaching OOA&OOD with UML", *Proceedings of the 18th IEEE Conference on Software Engineering Education and Training,* April 2005, pp 77-84.

[7] Pavlov, Vladimir; Yatsenko, Anton "'The Babel Experiment': An Advanced Pantomime-based Training in OOA&OOD with UML", *ACM SIGCSE Bulletin , Proceedings of the 36th SIGCSE technical symposium on Computer science education SIGCSE '05, Volume 37 Issue 1*, 2005, pp 231-235.