Title	One system, two ideologies: integrating the two worlds of software engineering education
Author(s)	Tse, TH
Citation	Proceedings - Ieee Computer Society's International Computer Software And Applications Conference, 1999, p. 246-247
Issued Date	1999
URL	http://hdl.handle.net/10722/48435
Rights	Creative Commons: Attribution 3.0 Hong Kong License

One System, Two Ideologies: Integrating the Two Worlds of Software Engineering Education *

T. H. Tse †
Department of Computer Science and Information Systems
The University of Hong Kong
Pokfulam Road, Hong Kong
tse@csis.hku.hk

1. Integrating the two worlds of software development

There are two contradicting ideologies in software engineering education. One ideology emphasizes on the popular methods such as object-oriented analysis and design. These methods have been developed by internationally renowned specialists according to their consulting experience. They are generally supported by comprehensive CASE tools with a user-friendly graphical front-end. They are well received because of the ease of understanding and the flexibility of use. Unfortunately, students have to learn the trade as a craft rather than an engineering process. There is no theoretical foundation enabling software engineers to agree on an unambiguous interpretation, to verify the correctness of the implementation, or to solve complex problems such as concurrency conflicts. The so-called standards are only practical guidelines. Unlike civil engineers in structural design, no software engineer can provide users with a guaranteed degree of confidence on the software designed.

The other ideology in software engineering education advocates formal methods, which help to specify and reason with precision the properties of software systems. They make use of formal tools including abstract models such as Z, algebraic models such as OBJ3, and concurrency models such as Petri nets. They can guarantee whether the systems have been implemented according to the specifications by means of correctness proofs or refinement methods. Unfortunately, software engineering students seldom have the chance to apply their theoretical knowledge after graduation. Although formal methods serve as an excellent means of reasoning with target systems, they are generally perceived by practicing software engineers to be too difficult. They do not have a user-friendly front-end. Moreover, they are primarily reasoning techniques that do not support the full development cycle.

Rather than having to make a difficult choice on one of the two ideologies, we would like to approach the problem from another perspective. In electrical engineering education, for instance, students are taught not to be satisfied with designs that are based purely on circuit diagrams and not supported by mathematics. Neither are they taught to present complex Fourier transforms to users for validation.

We advocate that in the future education of software developers, the two worlds should be integrated with each other. Students should be taught that a specification is a model of a real world solution. We must analyze and evaluate feasible models with a view to selecting the most suitable one. We employ graphical and mathematical techniques because they are better reasoning tools than narrative text. They cannot, however, supersede each other. The graphical notations in popular methodologies have proven track records of acceptance and practicality. They are very useful for conceiving abstract ideas and hence serve very well as blueprints to users. To solve the problem of ambiguity and incompleteness, these blueprints must be supported by a mathematical foundation, which is essential for reasoning and verification.

2. Integrating the two worlds of software verification

Popular software packages are only renowned for their user-friendliness but not their reliability. Effective software testing plays a very important role in reducing errors and improving the reliability. Partition testing is the most popular technique. The input domain is divided into subdomains, each of which will be tested separately to detect potential failures. Most software testers believe that partition testing is of course better than simply testing the software with random data. Recent empirical and simulation studies show, however, that partition testing is

*This research is supported in part by grants of the Hong Kong Research Grants Council and the University of Hong Kong Committee on Research and Conference Grants.

†Also with the Vocational Training Council, Hong Kong

0-7695-0368-3/99 $10.00 © 1999 IEEE

no better than random testing in many cases. This state of affairs is disheartening to software testers.

In spite of the popularity of object-oriented programming, additional challenges are in fact imposed on software testers. Because of abstraction and encapsulation, it is no longer a trivial matter to compare an expected outcome with the execution result. The observational equivalence of objects is very difficult to verify. As a result, object-oriented software testing turns out to be more complex than the wishful thinking of many programmers.

On the other hand, formal techniques for proving the correctness of programs have been in existence for a long time. They are, however, far from popular in the industry because the proofs are too demanding for the average software engineers and automatic theorem provers cannot possibly be constructed.

Similarly to software development, the future of software verification education lies also in the integration of the two worlds. Mathematical techniques should be used to analyze the situation to come up with practical guidelines on partitioning techniques and test data allocation techniques in partition testing, as well as techniques for testing object-oriented software. Software engineering students should be educated to be proficient with such techniques and guidelines.

3. Methods integration in the Pacific Rim

Researchers in the Pacific Rim have been recognized as pioneers in the integration of formal and practical methods. The advocacy was started by a Ph. D. student from Hong Kong who published his results in the Australian Computer Journal [1, 2] and received very favorable comments by software engineers from both worlds. It was followed up with the work of another Ph. D. student in New Zealand [3]. Similarly, the integration of mathematical techniques with practical software testing methods has been emphasized in the joint projects between Australia and Hong Kong [4, 5, 6].

Let us keep up with this promising direction and integrate the two worlds of software engineering education for the future.

4. Towards a comprehensive education

In order to support the above recommendations, we must provide a comprehensive education to software engineering students.

(*a*) Students should be trained in the formal aspects of software development and verification, in order to help them visualize complex systems more accurately by constructing abstract models and verify the correctness of the implementation.

(*b*) They should be trained in the tools and techniques in popular analysis and design methodologies, such as UML.

(*c*) They should be trained in other aspects of computer software, in order to design, implement, test, and maintain software systems.

(*d*) They should be trained to understand hardware technology, including the selection and design of hardware support.

(*e*) They should be trained in communication and interpersonal skills, so that they can elicit user requirements and present their cases.

(*f*) They should be trained in psychology and sociology, in order to understand real user needs and socio-psychological factors in the introduction of new systems.

(*g*) They should be trained in business and economics, in order to understand the financial and other management objectives behind software systems.

(*h*) They should be trained in project management, such as in software process management techniques.

5. Conclusion

There are two contradicting ideologies in software engineering education. Each of them, however, has its own problems. We advocate that the future of software engineering education lies in integrating the two worlds. SE education should not only be an engineering discipline in name, but also an engineering discipline in substance. Software engineer students should not be artisans who regard their trade as an art and learn only from experience; nor should they be mathematics students who are more comfortable with theory than practice. They should be trained as genuine engineers, who are competent with industrial applications as well as the supporting theory.

References

[1] T. H. Tse, "Integrating the structured analysis and design models: an initial algebra approach", *Australian Computer Journal*, vol. 18, no. 3, pp. 121–127, 1986.

[2] T. H. Tse, "Integrating the structured analysis and design models: a category-theoretic approach", *Australian Computer Journal*, vol. 19, no. 1, pp. 25–31, 1987.

[3] R. B. France, T. W. G. Docker, and C. H. E. Phillips, "Towards the integration of formal and informal techniques in software development", in *Proceedings of New Zealand Computer Conference*, New Zealand, pp. R-57–74, 1987.

[4] T. Y. Chen and Y. T. Yu, "On the relationship between partition and random testing", *IEEE Transactions on Software Engineering*, vol. 20, no. 12, pp. 977–980, 1994.

[5] T. Y. Chen and Y. T. Yu, "On the expected number of failures detected by subdomain testing and random testing", *IEEE Transactions on Software Engineering*, vol. 22, no. 2, pp. 109–119, 1996.

[6] H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen, "In black and white: an integrated approach to class-level testing of object-oriented programs", *ACM Transactions on Software Engineering and Methodology*, vol. 7, no. 3, pp. 250–295, 1998.