Insight in the Reuse Process?

Liesbeth Dusink

TU Delft
Tel: +31-15-783832
Email: betje@twi.tudelft.nl

Abstract

In this paper I take the position that at this moment we do not have insight in the software engineering process and as reuse is a software engineering process directed towards using existing (partial) solutions, we do not have insight in the reuse process either. We just do things without being able to explain why things work, should work, go wrong, etc. The only explanations given are based on common sense which is not enough from the scientific point of view.

Keywords: science, methods, techniques, process

Workshop Goals: Learning; networking; advance state of theory of reusable position papers.

Working Groups: reuse process models, reuse maturity models, reuse education, reuse and OO, reuse and formal methods

1 Background (reused)

In 1986 an Ada program library had to be build for the Delft Ada subset Compiler. During this work we were thinking about facilities to offer to the users. The step to reuse was made. Because of the Ada background we concentrated on components-based reuse on code level.

As reuse has two sides — the form of the reusable components influences how one can reuse and the process of reuse influences in which form one wants the components – a two-track research program was started in which the ideal form of components and the ideal process had to be found and adapted to each other.

2 Position

Since my first experience with software engineering and ever after I have wondered why all methods only state the products which have to be delivered. The syntax is described, the semantics get less attention, and how the products have to be made, how to transform a requirements document to a structure chart for example, is totally neglected.

When I started to develop my own software engineering method which supports the reuse of existing artifacts and knowledge, I got a further shock. The existing methods were based on experience only. No sound theories backed them, no explanations were given why it should be done in the way described or why it should work. No sound statistical comparisons were made.

Software engineering research is not yet research in the scientific sense. Software engineering methods are developed and tools are build based on reasonable sounding arguments only. Afterwards it is claimed "it works because of the higher productivity", but we all know the Hawthorne effect. A change can have a positive effect independent of the kind of change. Changing back to the old procedure has again a positive result. Therefore it is impossible to use this kind of information to build a unifying theory on software engineering. See [1, 2, 3, 4] for a discussion about this topic.

Another reason is that if one tries a statistical sound way of measuring the effect, there is the problem of the metrics. As Fenton [5] shows, we do not know what to measure, eg. should we measure lines of code in a month for productivity or should we measure centimeters documentation. We also do not know what we measure. What does it say when we write so many lines of code?

As I got the feeling that software engineering is problem solving by humans for humans, and that reuse is problem solving with existing (partial) solutions, I went to cognitive psychology to see whether theories existed on how people solve and should solve problems to base my own method on.

It appeared that several kinds of problems existed, i.e. formal problems and non-formal problems [6], and that software engineering could be classified as solving formal problems. For formal problem solving several theories existed.

Software engineering can be seen as a special form of problem solving. The splitting of the software lice cycle into several steps is the division of a problem into subproblems (a general problem solving method). The use of program plans [7, 8, 9, 10] can be compared with the schemata [11]. Functional fixedness in programming was proved by [12].

The human understander is best viewed as an opportunistic processor, capable of exploiting both bottom up *and* top-down cues as they become available [13]. This is consistent with the memory model of the schema theory. We see that software engineers do the same. Generally, it is found that software engineers first play with the problem, give partial solutions, go into details, etc. until they feel grip on the problem (a mental problem model is created) [14]. From then on they first give a solution in general terms before they specialize [15].

All these findings supported the idea that conclusions from cognitive psychology can be used in software engineering.

My work gives theories based on cognitive psychology, and from the conclusions drawn a process is derived and necessary characteristics for describing reusable components are derived.

My work has improved the state of the art by giving a process model which is also refined to a method.

My work has improved the state of the practice as HP has used ideas from it in their reuse project, the reuse process model as discussed in former workshops has incorporated several ideas.

2.1 Reuse

Without knowledge of problem-related concepts, the memory quickly reaches its limits when trying to understand code [16]. This is because the knowledge can not be related to existing schemata and thus has to be stored as separate facts in the short term memory. As the short term memory is non-associative and can contain up till 7 items, one sees the relevance of laying relations with existing knowledge [17, 6, 16, 18].

There are different approaches when trying to understand code, a systematic strategy and an ad hoc strategy [8, 19]. In the systematic approach first the total documentation is studied in a systematic manner. In the ad hoc approach documentation is read at random. The systematic approach can take too much time for large pieces of program and the ad hoc strategy gives poorer results when adapting a program. Therefor documentation has to be in such a way that it is easy to combine both strategies in an intelligent manner. The documentation has also to be in such a way that the limits of the memory are not reached very quickly.

In [20] mental laziness is remarked as one of the problems with reuse. In [21] some experiments are done about how to prevent that the habit masters the individual instead of the individual mastering the habit. It appears that by promoting productive thinking the problem of mental laziness could be overcome. If a solution is actively verbalized transformation to new situations becomes easier [11]. This improves reuse. The active participation in finding a solution improves the recognition of the possibility of applying the solution to other areas [11].

There are two basically different approaches to understanding a program. The first is the systematic strategy, where the programmer traces data flow and control flow throughout the program. The second strategy is the as-needed strategy, where the programmer reads only those part of the documentation or code as (s)he thinks to be of interest [19, 8].

3 Comparison with Other Work

Maiden and Sutcliffe explain findings from experiments with help of cognitive psychology. They also base tools on hypothesis from cognitive psychology.

In Bill Curtis' [22] collection of articles, one finds a lot of articles which compare methods or techniques for parts of the life cycle. But, as one of the comments from Curtis states, these comparisons are mostly not sound on methodological level, no experienced programmers are used thus the conclusions can not be extrapolated to experienced programmers as it it known that experienced programmers work different than less experienced programmers.

Fisher and his group have a model on how programmers work, based on cognitive psychology, and base a series of tools on their model.

One sees that other persons and groups concentrate on the tool side of software engineering. Where Mayer [23] made clear that the syntactic sugar in which concepts are modeled is important for faster and better understanding of the used concepts. And it is not yet clear what kind of syntactic sugar is helpful and what kind is not. There are conflicting studies [24].

References

[1] G. H. Bradley, "Cognitive Science View of Software Engineering," in Gibbs and Fairley [25], pp. 35–51.

[2] A. N. Habermann, "Report of the Software Engineering Priciples Working Group," in Gibbs and Fairley [25], pp. 369–380.

[3] W. E. Richardson, "Why Is Software Engineering So Difficult?," in Gibbs and Fairley [25], pp. 98–106.

[4] R. H. Thayer and L. A. Endres, "Software Engineering Project Laboratory: The Bridge Between University and Industry," in Gibbs and Fairley [25], pp. 263–291.

[5] N. E. Fenton, *SOFTWARE METRICS: A Rigorous Approach.* Chapman and Hall, 1991. ISBN 0-442-31355-1.

[6] W. Wickelgren, *Cognitive Psychology.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1979.

[7] R. Rist, "Planning in Programming: Definition, Demonstration, and Development," in Soloway and Iyengar [26], pp. 28–47. (Human/Computer Interaction Series).

[8] D. Littman, J. Pinto, S. Letovsky, and E. Soloway, "Mental Models and Software Maintenance," in Soloway and Iyengar [26], pp. 80–98. (Human/Computer Interaction Series).

[9] C.-C. Yu and S. Robertson, "Plan-Based Representations of Pascal and Fortran Code," in Soloway *et al.* [27], pp. 251–256. special issue of the ACM/SIGCHI Bulletin.

[10] S. Wiedenbeck, "Processes in Computer Program Comprehension," in Soloway and Iyengar [26], pp. 48–57. (Human/Computer Interaction Series).

[11] R. Mayer, *Thinking, Problem-Solving, Cognition.* W.H. Freeman and Company, 1983.

[12] R. Mayer, "Different Problem-Solving Competencies Established in Learning Computer Programming With and Without Meaningful Models," *Journal of Educational Psychology*, no. 67, pp. 725–734, 1975.

[13] S. Letovsky, "Cognitive Processes in Program Comprehension," in Soloway and Iyengar [26], pp. 58–79. (Human/Computer Interaction Series).

[14] R. Guindon and B. Curtis, "Control of Cognitive Processes during Software Design: What Tools are Needed?," in Soloway *et al.* [27], pp. 263–268. special issue of the ACM/SIGCHI Bulletin.

[15] K. Duncker, "On Problem Solving," *Psychological Monographs*, vol. 58, p. 270, 1945.

[16] B. Shneiderman, *Software Psychology: Human Factors in Computer and Information Systems.* Little, Brown Computer Systems Series, Little, Brown & Company, 1980.

[17] A. Newell and H. A. Simon, *Human Problem Solving.* Engle Wood Cliffs N.J.: Prentice-Hall, 1972.

[18] G. A. Miller, "The Magical Number Seven— Plus or Minus Two: Some Limits on our Capacity for Processing Information," *Psychological Review*, no. 63, pp. 81–97, 1956.

[19] J. Pinto and E. Soloway, "Providing the requisite Knowledge Via Software Documentation," in Soloway *et al.* [27], pp. 257–262. special issue of the ACM/SIGCHI Bulletin.

[20] N. Maiden and A. Sutcliffe, "The Abuse of Re-use: Why Cognitive Aspects of Software Reusability are Important," in *Software Re-use, Utrecht 1989: Proceedings of the Software Reuse Workshop, 23-24 November 1989, Utrecht, The Netherlands* (L. Dusink and P. Hall, eds.), ch. 10, Springer-Verlag, 1991.

[21] A. Luchins and E. Luchins, "New Experimental Attempts at Preventing Mechanization in Problem Solving," Penguin Modern Psychology, ch. 6, Penguin Books, 1968.

[22] B. Curtis, ed., *Tutorial: Human Factors in Software Development (second edition).* IEEE Computer Society Press/ North-Holland, 1986.

[23] R. E. Mayer, "Comprehension as affected by structure of problem representation," in Curtis [22], pp. 320–326.

[24] S. B. Sheppard, E. Kruesi, and B. Curtis, "The effects of symbology and spatial arrangement on the comprehension of software specifications," in Curtis [22], pp. 327–334.

[25] N. E. Gibbs and R. E. Fairley, eds., *Software Engineering Education: The Educational Needs of the Software Engineering Community.* New York: Springer-Verlag, 1987.

[26] E. Soloway and S. Iyengar, eds., *Empirical Studies of Programmers, papers presented at the First Workshop on Empirical Studies of Programmers, June 5-6, 1986, Washington, DC.*, Ablex, 1986. (Human/Computer Interaction Series).

[27] E. Soloway, D. Frye, and S. Sheppard, eds., *CHI'88 Conference Proceedings, Human Factors in Computing Systems, May 15-19, 1988 Washington, DC.*,, ACM Press, 1988. special issue of the ACM/SIGCHI Bulletin.

4 Biography (reused)

Liesbeth Dusink is a lecturer at Delft University of Technology, chair software engineering, since 1985. She teaches introduction in programming in Modula-2 with VDM, software engineering and object oriented approach, and software engineering environments. Since 1992 she also works as research engineer at Cap Gemini Innovation, Rijswijk. At Cap Gemini she designs in VDM and VDM++ and programs in Ada. At this moment she works on a project for tracking and tracing of trains. These two jobs together offer a unique combination of theory and practice.