Software Engineering and Reuse

E.M. Dusink
TU Delft, Dept. Computer Science
Julianalaan 132
2624 BL Delft
The Netherlands
e-mail: betje@dutiaa.tudelft.nl

Abstract

This article contains relevant concepts of cognitive psychology and software psychology and a framework of the software engineering method based on these concepts. Cognitive psychology was studied with special attention for the differences between problem solving with help of existing (partial) solutions (reuse) and without help of existing (partial) solutions (no reuse).

Keywords: reuse supporting method, cognitive psychology, software psychology

1 Introduction

This article contains relevant concepts of cognitive psychology and software psychology and a framework of the software engineering method based on these concepts. Cognitive psychology studies how humans solve problems. Software psychology studies a.o. specialised problem solving methods for software engineering. And thus can be seen as applied cognitive psychology. Cognitive psychology was studied with special attention for the differences between problem solving with help of existing (partial) solutions (reuse) and without help of existing (partial) solutions (no reuse). In a later stage the framework will be completed with insight from software engineering and reuse to a software engineering method which supports reuse. The method will be verified with a statistical experiment.

A rational for the use of cognitive psychology is given. The used concepts of cognitive psychology and software psychology are described with their literature references as they are no common knowledge in the software engineering society. They are also described to provide the reader with the necessary information for criticising the used ideas, our interpretation of the ideas, or our usage of the ideas. For an overview of our findings in the field of cognitive psychology see [5].

The software engineering method being developed has to support compositional reuse and has to result in at least the same quality products as existing software engineering methods. The goal, group, application domain, metrics, and the way of testing the method are described in [6]

2 Why Cognitive Aspects in Software Engineering

The software engineering process can be seen as the capturing of information and the solving of problems with humans as the driving force. In this view it is of interest how *humans* work with information and how they solve problems. Problem solving is the creative activity of finding a correct implementation given certain requirements. Within this view on software engineering cognitive psychology can be a help.

Humans tried to formalise and control the problem solving aspect of software engineering by developing the waterfall model. This conventional life cycle model can be traced to the reductionist mode of inquiry in planning and logistics. Planning and logistics were emerging as organised and systematic approaches to problem solving [1]. The waterfall model is nothing more or less then a general problem solving method stated in software engineering terms.

As software systems grew complexer and larger, a systematic and organised approach for problem solving on a more detailed level was needed. An approach to fill this need was in the form of increased attention on intermediate products and verification steps of the transition [1]. This verification is a step which was already accentuated in cognitive psychology [16]. That need could already have been fulfilled before feeling it.

As software had to be written for more divers systems and for less well informed people problems occurred with stating all the requirements beforehand and taking all design decisions beforehand. This was already known in cognitive psychology from experiments with problem solvers. But than in the form that a problem can only be stated in the direction of a possible solution [7]. An example is with students who do not understand a lecture, they only can tell you that they do not understand the lecture but not in what way or what part of the lecture was outside their comprehension.

Another part of this problem, not yet understood by software engineers, is caused by the fact that only experienced people can work top down and give a good structured solution. Less experienced people have to go forwards and backwards in abstraction level to be able to solve a problem.

Software engineers now try to fill this need in very divers ways: spiral model; rapid prototyping; transformational approach; incremental development; etc. As the evaluation of what caused the problem only gave part of the cause, this solution is only partial and in the near future software engineers will see that rapid prototyping does not solve all of the problem.

Because of all the reasons for changing the conventional life cycle model and the fact that it is faster to use knowledge already in existence, in this paper the software engineering process will be redeveloped with help of knowledge from the cognitive psychology in order to profit from things which are already correct and to get knowledge about what to avoid.

3 Why Cognitive Aspects in Reuse

Reuse is the use of existing partial solutions (components) by humans in such a way that combined together given requirements are fulfilled. Several cognitive problems prevent users from successfully exploiting reusable components, a.o. understanding what a reusable component does and when to apply it. In this view reuse is also problem solving.

Understanding a component is done by learning (about) the component. ¿From cognitive psychology we know that there are basically two ways of understanding a component, the broad general

notion of the features which makes it possible to use the component outside the scope of learning (important for reuse) and the functional fixedness understanding which prevents use outside the scope of learning (hinders reuse). In the first kind of learning a brick has volume, weight, colour, ..., in the second kind of learning a brick can be used for building buildings. This different perception makes that the question: "give thirty different usages of a brick" is easier to answer if the first kind of learning has occurred. In cognitive psychology it is known how to learn about things in order to get the first kind of learning and how to stimulate the second kind of learning. This problem is not yet recognised in the reuse world, but it is not necessary to encounter the problem if we profit now from knowledge of cognitive psychology.

The other problem, when to apply a component, is already encountered. In an experiment with reuse Maiden and Sutcliffe [10] saw that without guidelines reuse was degraded to copying without understanding why and what was used. And therefor not the right usage was made of the reusable components. It was a form of mental laziness. This phenomenon was already known for at least *twenty* years in cognitive psychology [9]. So, once again, problems relevant for reuse were already known in cognitive psychology before they were known within the software engineering/reuse field. And, in cognitive psychology a solution for these problems exist [11] where we, software engineers and reusers, have to start with looking for a solution.

As we want our field of science to mature as quickly as possible, let's make use of the knowledge of cognitive psychology now we know it exists and now we know it is applicable to both the software engineering process as well as to reuse.

4 Main Conclusions from Cognitive Psychology

The main conclusions are on the field of problem solving methods, how reusable components have to be learned and how to stimulate creativity. They are described in this section.

Depending on the understanding of the description of the problem (ie. requirements) and the familiarity with the reusable components, there exist three kinds of obstacles for finding a solution. The three kinds are: interpolation problem, dialectic problem, and synthesis problem [4]. An interpolation problem occurs when the combination of a good understanding of the problem and the components happens. A dialectic problem occurs when the combination of a good understanding of the components and a low understanding of the problem happens. A synthesis problem occurs when the combination of a bad understanding of the components and a high understanding of the problem happens. A combination of dialectic and synthesis problem exist in case of a low understanding of the problem and of the components. The problem solving methods which one has to follow depending on the kind of obstacle are described in [4]. General problem solving methods are described in [16, 17, 25].

Two kinds of learning of components can be of importance for the reuser. The first kind of learning is the acquisition of broad, non-specific, general notions about the properties of the component experienced. This seems to provide the knowledge repertoire essential for productive ($\sim$creative) thinking. The second kind of learning is the acquisition of experiences which provides perceptions of specific, limited, functional characteristics. This seems to lead to "functional fixedness" in problem solving perceptions. Such fixedness limits the range of perceptual organisations capable of being developed by the reuser and so interferes with problem solving [3].

It is clear that the functional fixedness kind of learning is of little use for a reuser, but it is of use for a problem solver who only solves one kind of problems. The discrepancy between usefulness and uselessness of functional fixedness can be overcome by learning basic specific facts coupled with general problem solving techniques. In [11] it is advised to learn first the global concepts used by the component before the mechanics.

The main difference for problem solving with and without existing components was lain in the fact that components could be seen as giving a hint as in which direction to look for a solution.

The solving of problems asks for creativity. By accepting the idea that the long term memory is associative [26] and that knowledge in the long term memory is clustered into schemata [12] and that the retrieval of knowledge from the long term memory is keyword-based [26] we can sketch an approach for being creative [26]. The memory has to be prepared to meet the conditions required for creativity. By storing a lot of questions in the memory answers will be recognised when they come along. Thus during the process of learning one has to ask oneself all the "w"-questions, why, when, where, etc. One also has to try to relate new information to the own schemata [26]. If a rich store of novel integrative schemata and unanswered questions exists in memory, and if good cognitive plans for playing with ideas exist, ideas can be created on demand much as any other skill can be performed [26].

5 Main Conclusions from Software Psychology

The main conclusions are about the existence of cognitive processes in software engineering, necessary mental make-up, how to understand problems and what the problems are with reuse.

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 [18, 8, 28, 27] can be compared with the schemata [12]. Functional fixedness in programming was proved by [11]. All these findings supported the idea that conclusions from cognitive psychology can be used in software engineering.

A different mental make-up was needed in the different stages of the software engineering process. The ability to live with uncertainties and the ability to make decisions is necessary in the beginning of the life cycle, whereas the ability of convulsive preciseness is necessary at the end [21].

Without knowledge of problem-related concepts, the memory quickly reaches its limits when trying to understand code [21]. This is because the knowledge can not be related to existing schemata and thus has to be stored as separate facts (for the time being in the short term memory which is non-associative and can contain up till 7 items) [14, 26, 21, 13]. Thus documentation of code has to give the used concepts of the application domain and the used concepts of the computer domain.

There are different approaches when trying to understand code, a systematic strategy and an ad hoc strategy [8, 15]. 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. The documentation is best understood

if it is in a very succinct symbology [20], the spatial arrangement is of less importance.

In [10] mental laziness is remarked as one of the problems with reuse. In [9] 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 verbalised transformation to new situations becomes easier [12]. This improves reuse. The active participation in finding a solution improves the recognition of the possibility of applying the solution to other areas [12].

6 A Framework for a Software Engineering Method

Although software engineering life cycles can differ very much, they all have about the same division in analysis, requirements, design, implementation, testing and debugging. These same phases, although not necessarily in this order or with the same emphasis, are the phases of the framework. In this section concrete use is made from the findings described in the former two sections.

Before modification is possible, the part of the product to be modified has to exist. Before debugging, tests have to be applied. Before it is possible to test, design and/or implementation has to be done, etc. But, is it necessary to have the requirements finished before looking at the design phase? As humans can be seen as opportunistic processors [7], and as we always use past experience, the answer is "no". Before the requirements are finished, it has to be known whether they are implementable. For making a choice between alternatives with regard to reuse, ease of implementation, and risks one has to look ahead. Also because a problem can only be stated in the direction of a possible solution one has to look ahead [19]. And of course, because of changing insights and bugs one has to go back to previous phases and one has to look back to be able to learn. Because of all these reasons a yoyo approach is suggested as an ordering of the phases, where the going down and up is steered by an "as-needed" strategy.

In every phase the understanding of the problem, the focusing on reuse, and learning are stressed. Understanding is emphasised because the right problem has to be solved.

Components have to exist in the environment to be able to reuse. A components base is coupled with a means to select potentially useful components. In an experiment [12] 2.4 minutes were necessary to find a solution when only useful components were given, 7.5 minutes were necessary when all components where given and hints about which ones were useful, and 15.2 minutes were necessary when the same components were given as to the former group but no attention was drawn to potential useful components. This means that the selection mechanism is critical.

The method emphasises learning as only components which are known can be used correctly and as things which one has learned can be reused easier than things fairly unknown to the (re)user. Possible ways to integrate learning are:

- The organisation of persons on jobs has to be done in such a way that a junior software engineer is mentored by a senior one. In this way real-time feedback on the work of the junior is possible.

- Feedback from next stages has to reach the engineer also as one learns most from ones own errors.

- All software engineers have regular sessions to play with the (new) reusable components, in order to learn them in a context free way.
- The software engineers have to learn general problem solving techniques to improve their creativity.

A possible manner to help learning passively is by documenting. Documentation has to state the relations among concepts backgrounds and why's. Also alternatives with their pro's and con's and the reason for their disregarding have to be documented. This emphasises learning. The experienced software engineers in the application domain have to contribute to be sure the information is already computer oriented formulated.

7 Conclusion

A framework for a software engineering method was retrieved from cognitive psychology and software psychology, without any considerations on management, economics, and current software engineering practices. The resulting idea corresponds with current developments in software engineering practice about rapid prototyping and operational approach [2]. The way of working when developing the framework was the same as suggested in the framework. This first "evaluation" gave the impression of an ergonomic and tailorable whole in which much reuse was done. Work has to be done on the software engineering side of the method.

References

[1] William W. Agresti. The Conventional Software Life-cycle Model: Its Evolution and Assumptions. In Agresti [2], pages 2–5.

[2] W.W. Agresti, editor. *New Paradigms for Software Development.* IEEE Computer Society Press, 1986.

[3] H.G. Birch and H.S. Rabinowitz. The Negative Effect of Previous Experience on Productive Thinking. In Wason and Johnson-Laird [24], chapter 3.

[4] D. Dörner. *Problemlösen als Informationsverarbeitung.* Kohlhammer Standards Psychologie, Teilgebiet Denkpsychologie. W. Kohlhammer, Stuttgart, 2. Auflage Edition, 1979.

[5] E.M. Dusink. Cognitive Psychology, Software Psychology, Reuse and Software Engineering. Technical report, TU Delft, Delft, the Netherlands, 1991. to appear.

[6] E.M. Dusink. Testing a Software Engineering Method Statistically. Technical report, TWI, TU Delft, Delft, the Netherlands, 1991. to appear.

[7] S. Letovsky. Cognitive Processes in Program Comprehension. In Soloway and Iyengar [23], pages 58–79. (Human/Computer Interaction Series).

[8] D.C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental Models and Software Maintenance. In Soloway and Iyengar [23], pages 80–98. (Human/Computer Interaction Series).

[9] A.S. Luchins and E.H. Luchins. New Experimental Attempts at Preventing Mechanization in Problem Solving. In Wason and Johnson-Laird [24], chapter 6.

[10] Neil Maiden and Alistair Sutcliffe. The Abuse of Re-use: Why Cognitive Aspects of Software Re-usability are Important. In Liesbeth Dusink and Patrick Hall, editors, *Software Re-use, Utrecht 1989: Proceedings of the Software Re-use Workshop, 23-24 November 1989, Utrecht, The Netherlands*, chapter 10. Springer-Verlag, 1991.

[11] R.E. Mayer. Different Problem-Solving Competencies Established in Learning Computer Programming With and Without Meaningful Models. *Journal of Educational Psychology*, (67):725–734, 1975.

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

[13] Georg A. Miller. The Magical Number Seven— Plus or Minus Two: Some Limits on our Capacity for Processing Information. *Psychological Review*, (63):81–97, 1956.

[14] Allen Newell and Herbert A. Simon. *Human Problem Solving.* Prentice-Hall, Engle Wood Cliffs N.J., 1972.

[15] J. Pinto and E. Soloway. Providing the requisite Knowledge Via Software Documentation. In Soloway et al. [22], pages 257–262. special issue of the ACM/SIGCHI Bulletin.

[16] G. Polya. *How to Solve It.* Garden City N.Y., 1957.

[17] G. Polya. *Mathematical Discovery*, volume II: On Understanding, Learning and Teaching Problem-Solving. Wiley, New York N.Y., 1968.

[18] R.S. Rist. Planning in Programming: Definition, Demonstration, and Development. In Soloway and Iyengar [23], pages 28–47. (Human/Computer Interaction Series).

[19] A.R. Rohr. *Kreative Prozesse und Methoden der Problemlösung.* Beltz Monographien. Beltz, Weinheim, 1975.

[20] S.B. Sheppard, J.W. Bailey, and E.K. Bailey. An Empirical Evaluation of Software Documentation Formats. Human/Computer Interaction, chapter 6. Ablex, Norwood, New Jersey, 1984.

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

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

[23] E. Soloway and S. Iyengar, editors. *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).

[24] P.C. Wason and P.N. Johnson-Laird, editors. *Thinking and Reasoning, Selected Readings.* Penguin Modern Psychology. Penguin Books, 1968.

[25] W.A. Wickelgren. *How to Solve Problems: Elements of a Theory of Problems and Problem-Solving.* A Series of Books in Psychology. W.H. Freeman and Company, San Francisco, 1974.

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

[27] S. Wiedenbeck. Processes in Computer Program Comprehension. In Soloway and Iyengar [23], pages 48–57. (Human/Computer Interaction Series).

[28] C.-C. Yu and S.P. Robertson. Plan-Based Representations of Pascal and Fortran Code. In Soloway et al. [22], pages 251–256. special issue of the ACM/SIGCHI Bulletin.

8 About the Author

Liesbeth Dusink has been working at Delft University of Technolgy since 1985. She got het MD in Biology from Utrecht University. At the moment she teaches courses on programming, software engineering, and programming support environments. Her research concentrates on software engineering methods which supports compositional reuse and on software measurement and software process measurement.