In M. Petre & D. Budgen (Eds) Proc. Joint Conf. EASE & PPIG 2003 Pages 436-452

Does the Empirical Evidence Support Software Visualisation ?

Pamela O'Shea
Chris Exton

Software Visualisation and Cognition Research Group
Department of Computer Science and Information Systems,
University of Limerick,
Ireland

pamela.oshea@ul.ie
chris.exton@ul.ie

Abstract

Previous experiments and empirical studies in the software comprehension field have been criticised by skeptics, for example [Sheil 1981]. Although it has been twenty-two years since his publication, many issues still need to be addressed to this day. We are left with no definitive catalogue of proof that either confirms or denies the usefulness of Software Visualisation in the field of software engineering. This paper will discuss some empirical studies and experiments from the past, in order to present future researchers and evaluators of Software Visualisation tools with a guideline as to how we can learn from both the good and bad traits of past experiences.

1 Introduction

In the foreword of [Stasko, Domingue, Brown, Price 1998], a common goal for all types of visualisations is identified as *"transforming information into a meaningful, useful visual representation from which a human observer can gain understanding."* It is this *understanding* that must be proven through empirical studies. Many means for testing understanding and comprehension have been developed in the software comprehension field and examples of these can be seen in the surveyed experiments.

Understanding stems from one's own personal *mental model*. The programmers' *mental model* is defined by [Von Mayrhauser 1995] as *a current internal (working) representation of the software under consideration.* Von Mayrhauser discusses both the static and dynamic elements of a mental model. A mental model can even be generated from text based debugging tools, for example, as a programmer becomes more familiar with the code and begins to *chunk* groups of code together [Boehm-Davis et al 1987] etc. *SV* tools should provide many views to support its wide audience of users, these views can be defined through the observations of software engineering practices in empirical studies.

According to [Stasko, Domingue, Brown, Price 1998], this *understanding* can be aided by *SV*, where *Software Visualisation* is defined as *"the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software"*.

It is then reasonable to state that any research (in this case *SV*), which aims to reduce this cognitive load and reduce time spent understanding code during software maintenance, is a worthwhile endeavour. Many studies already *suggest* and some even *conclude* that *SV* is helping in such a way. However, there is still a greater need for more studies to be carried out on larger groups of programmers in industry. This and many other issues will be discussed throughout the paper.

Programmers in industry need to be given serious input into the requirements phase of these *SV* tools, afterall they are going to be the people using these tools in an effort to reduce their cognitive load. It will be their mental models that must be taken into account as well as their behaviours, such data cannot realistically be gathered in the lab, it must be observed *"in the wild"* as it were. It should be noted that no matter how unobtrusive an experimenter tries to make the experiment, it will always be a deviation from their normal working environment and this must be taken into consideration. While it might be an excellent idea to have the participant ask questions of the experimenter in order to gain an understanding of the system, as they would with work colleagues, it is not usual for ones' *colleague* to be a stranger or even take notes!

Having said this, the ideal is not always readily realised in the real-world. For example, it is quite obvious that there is a greater need for experiments

to incorporate greater sample sizes, but this is a difficult recommendation to realise. Firstly there is the problem of recruiting busy professional programmers, and secondly, there is so much data to analyse, especially using *Talk-Aloud* protocols as was done in [Pennington 1987] and [Storey et al 1998].

Software Visualisation in terms of this paper is discussed in Section 2 since it tends to be a broad category and requires further refinement for the remaining discussion.

Section 3, presents a short selection of some empirical studies. Each study is discussed in comparison to the others, both good and bad traits are highlighted. The main theme is what we can learn from previous research in order to empirically evaluate future Software Visualisation tools. Some very noteworthy points have been recorded in previous studies and should be kept in mind for the future. The studies were chosen to cover a range of participant experience. For example in [Pennington 1987], forthy professional programmers were used, this differs from the mixed study carried out by [Boehm-Davis et al 1987], where the participants were eighteen professional and eighteen novice programmers.

Section 4, deals with future issues and discusses the factors that have been learnt through the survey of experiments. Issues such as sample sizes, training time, measurement of spatial aptitude are discussed as well as other features.

2 Software Visualisation Overview

SV depends upon the research carried out in the Software Comprehension field. Both these categories are quite wide, so further refinement and definition is required to clarify the meaning in the context of this paper.

According to [Boehm-Davis et al 1996] *Software Comprehension* consists of *reconstructing the logic, structure, and goals that were used to write a computer program.* Each experiment measures software comprehension in some form to varying degrees, if that is its goal. What is meant by this is that each empirical study will have different requirements and as a result will have different interpretations of when a participant has achieved enough *comprehension* to satisfy the task at hand. For example, there may not be a need to understand every single line of code in the system, instead, it might be desirable to narrow the focus to a certain number of classes/methods or functions. Littman speaks about the programmer being able "to localize parts of the program to which changes can be made", this is referred to as the *As-Needed Strategy* [Littman et al 1986].

Software Visualisation also needs to be defined in terms of this paper. *SV* tends to become a parent category for all types of visualisations and encompasses such categories as *Algorithm Animation*, *Program Visualisation*, *Visual Programming* etc.

Algorithm Animation is *"the process of abstracting a program's data, operations, and semantics, and creating dynamic graphical views of those ab-*

stractions " [Stasko 1990].

Visual programming refers to any system that allows the user to specify a program in a two (or more) dimensional fashion, [Myers 1986].

Program visualization, is where *the program is specified in the conventional, textual manner, and the graphics is used to illustrate some aspects of the program or its run-time execution."*, [Myers 1986].

In relation to the software engineering field, *Software Visualization* (SV) should have the goal of making the professional software engineers' cognitive tasks easier (usually using *Program Visualisation*). However, it was found that there is more need for studies focusing on *Program Visualisation* tools using professional programmers. For example, in the past, the majority of empirical studies were performed on academic participants rather than professional programmers in industry. Thus limiting the type of conclusions that could be drawn as well as running the risk of missing behaviour's that are particular only to professional programmers. As a result, the examination of empirical studies in this paper incorporates many of these studies run on student programmers due to lack of industrial run studies. It is hoped that such studies may help to provide a reference point, that can be used by other researchers to devise their own empirical study.

3 A Short Selection of Empirical Studies

3.1 Pennington's Study : *Comprehension Strategies in Programming*

Of all the studies surveyed, [Pennington 1987] had the largest number of participants with a total of forty professional programmers. An allocation of two and a half hours to the experiment also makes it the longest. Although [Cunniffe and Taylor 1987] only used program segments, Pennington's two hundred line program can be considered to be quite small in comparison to the seventeen-hundred line program used in [Storey et al 1998].

The strength of this study can be seen through the quality of participants, forthy professional programmers from industry were observed in order to find any differences in comprehension strategies between programmers with high and low levels of comprehension. Half of these programmers were asked to *Think-Aloud.*

Experimental data was gathered using a controlling program which recorded all of the programmers' responses, explanations and response times, as well as recording which line of the program was in the center of the screen. Complimenting this, programmers were allowed to take notes or draw diagrams while studying the program.

Program comprehension was studied using the following categories: *Operations*, *Control Flow*, *Data Flow*, *State*, and *Function.*

Van Dijk and Kintsch's *textbase* and *situation model* are referred to as the *program model* and *domain model* in this paper. *Program Model* is defined as,

"*a representation that highlights procedural program relations in the language of programs*". *Domain Model* is defined as, "*a representation that highlights functional relations between program parts that is expressed in the language of the domain world objects*". Pennington suggests from her previous results, that the program model is constructed before the domain model.

In contrast to the other experiments, the two hundred line program was a real-world COBOL program used in production, which was also ported to FORTAN for the experiment. Adopting a *Bottum-Up* approach, the participants were asked to summarise the program in text form after a forty-five minute study of the program. Four levels of detail were recorded: 1. Detailed statements, 2. Program level statements, 3. Domain level statements and 4. Vague statements. Twenty questions were presented in a quiz format on screen to examine comprehension (cloze-test). Thirty minutes was allocated to the modification tasks, in which all participants were unfamiliar with the modification task.

Being the only experiment surveyed that examined comprehension strategies, the results were categorised into comprehension groups. It was observed that the programmers who achieved best comprehension levels had used a cross-reference summary strategy (i.e. participants who summarised using program, domain and operation statements evenly). While programmers with low comprehension levels, had used either a program level summary or a domain level summary.

Pennington states that "*In our research, comprehension was strongly related to participants' strategies in constructing a model of the domain*". Interestingly, Pennington goes on to say that "*we must be prepared for a multiplicity of mental representations, even within one head*".

In conclusion, the program used was not extremely complex when compared to the seventeen-hundred line program used in [Storey et al 1998], which was designed to have complex control flow. However, there are quite a number of strong points when compared to the other surveyed experiments. Most importantly, all the programmers were professional software engineers and the code used was a real-world program. This is quite significant, when compared to [Cunniffe and Taylor 1987] for example, where only program segments were used. In general, programmers do not try and recall parts of a program without reference to the source code. This is the one issue that stands out from this experiment, as it is not an everyday programmers task. From an SV tool point of view, support is given for tools that use hypelink styled layouts e.g. PUI [Chan and Munro 1997], as the best comprehenders used a cross-reference summary strategy and this would allow them to navigate more freely. SV tool designers can look to this experiment as evidence for supporting multiple views but it must also be kept in mind that recalling is not an everyday programmer task.

3.2 Cunniffe and Taylor's Study : *Graphical vs. Textual Representation: An Empirical Study of Novices' Program Comprehension*

As with a large number of experiments to date, [Cunniffe and Taylor 1987] was performed using twenty-three student *novice* programmers. The group was divided in two, where group one had learned to program textually using Pascal. In contrast, group two had learned to program graphically using FPL (First Programming Language which was developed at Columbia University). The authors set out to discover, 1. Is one representation better only for certain stages of learning ?, 2. Only for certain aspects of programming ?, or 3. only for certain learners ?

Differing from Pennington's five comprehension categories, participants' comprehension was judged by how quickly and accurately the questions were answered. Of all the surveyed studies that used source code, Cunniffe and Taylor had the shortest, using only program segments and not full programs. Each segment had three questions, based on each of the following types [Atwood and Ramsey 1978] *Type I*: recognition of specific simple structures. *Type II*: recognition of flow of control and input/output. *Type III*: evaluation of flow of control, output, or variable values.

An important issue to note here, and one that is not often seen in experiments is that training was provided to the students well in advance since they learned FPL in class. This difference can be seen when compared to studies such as [Boehm-Davis et al 1987], where the participants only received a half hour training.

Both the visual and verbal aptitudes were measured for all participants [Ekstrom et al 1976]. The *Paper Folding Test (VZ-2)* measured visual aptitude, while *The Advanced Vocabulary Test II (V5)* measured verbal aptitude. Amongst not only the surveyed experiments but others, it is quite rare to see the spatial aptitude measured.

Eight Different program segments were designed and unlike Pennington's experiment, the participants were familiar with the task. Each segment was coded twice, once in Pascal and once in FPL. Both versions were similar except for variable names in order to reduce training effects and repetition. All of the questions could be answered from the current display and did not require a larger context.

An online system recorded the reaction times and gave detailed instructions. Similar to Pennington, participants were instructed to answer quickly and accurately. The mean reaction time for FPL was approximately five seconds faster than the Pascal reaction time. The largest difference between FPL and Pascal was seen in answers to the TYPE III questions. Cunniffe and Taylor did not find this surprising since these questions required evaluation of input and determination of output.

A positive correlation between accuracy and spatial aptitude was found, no matter which representation was used! Regardless of the participants' visual aptitude, the graphical representation of the program was comprehended

more quickly. On the downside, it should be noted that the authors mentioned the fact that they placed emphasis on the reaction times as a measure of comprehension and therefore devised questions that they knew could be answered. As a result, constrained the conclusions that could be made about the effect of the graphical representation on comprehension accuracy.

In conclusion, the strongest point about this experiment was the fact that the participants' spatial aptitude was measured. Interesting findings were made where accuracy was linked to spatial aptitude. Of all the surveyed experiments, Cunniffe and Taylor emphasised training the most. Training is an important issue, especially when it comes to evaluating complex SV tools. The lesson here is that training should be emphasised more in order to increase confidence in the results. Two options are available to the SV evaluator here, either training can be provided prior to the experiment (e.g. a couple of weeks) to allow the participant to become familiar with the tool over a longer period, or the SV tool can be integrated into a well known environment that is already familiar to the developer. The program segments used are not real programs and no coding was asked of the participants. This is somewhat disappointing when compared to studies like [Boehm-Davis et al 1987] where three different design types were employed or in [Storey et al 1998] where a seventeen-hundred line program was used.

3.3 Boehm-Davis et al Study : *Mental Representations of Programs for Student and Professional Programmers*

The value of this study comes from its not often found mix of participants' qualifications. Thirty-six programmers (eighteen professional and eighteen students) were asked to make either simple (make a change in 1 location) or complex (many locations) modifications to three different programs. Each program used a different design i.e. in-line code, functional decomposition or object-orientated.

[Boehm-Davis et al 1987] set out to examine programmers' cognitive representations of software. The only surveyed experiment to supply each participant with supplementary materials: a program overview, a data dictionary, a program listing and listings of both current and expected output from the program. Modifications to be performed were supplied at the start of the experiment and not as required, which is something to always consider when giving maintenance tasks during an experiment.

Little training time was given, a half an hour training was allocated where each participant was given a sample problem to solve. The three programs were then presented to the programmer in a random order (using a different problem for each program type).

Similar to Pennington and Cunniffe & Taylor, a computer was used to record responses (i.e. each call for an editor command etc.). In contrast to the other means of judging programmers' comprehension, the experiment

assistant helped to gather the contents and structure of the programmers' mental model for all three programs (related to Buschke's 2D grid procedure [Buschke 1977]). The programmer had to recall as many components of the program as possible, each component was then written on a separate large index card. The relationships between these components were then specified by wiring the relationships on small index cards, which could then be arranged to show the programmers' mental model. Five variables were used to reflect the programmers mental model: Number of program segments/chunks, Number of relationships, Depth of structure, Width of structure, and Connectedness of structure.

The differences between student and professional programmer were fewer than one would have thought, but as suggested in the paper, these students tended to be very good. The main difference was seen in debugging time, where students took 6.9 minutes longer on average. It was observed that there are two criteria for ease of maintainability: 1. Ease of finding specified information and 2. Ease of recognising relevant program structures. Both the mental model and the information gathering process are critical aspects of the maintenance performace task.

In conclusion, the three design types used in the programs as well as the mix of both professional and novice programs makes a solid foundation for these experimental results. This type of study would be interesting to replicate in order to explore the differences between novice and professional programmers more clearly. From an SV tool point of view, the lessons learned is that navigation must be fluid, (e.g. the SHriMP tool) allowing the developer to find that information required, this is the first criteria for ease of maintenance "*ease of finding specified information*". The chunks of code that the developer builds in their mental model should be allowed to be reflected on screen as well, this is second criteria for ease of maintenance "*ease of recognising relevant program structures*".

3.4 Petre and Blackwell : *A Glimpse of Expert Programmers' Mental Imagery*

The most unusual of the surveyed studies [Petre and Blackwell 1997], with an aim to find out how closely the mental images of experts correspond to external representations. Interestingly, there is a reference to [Hitch et al 1995], where it was concluded that "*verbalization overshadows insight*". Petre and Blackwell explain that requiring people to talk can inhibit insights through imagery. Perhaps this can be seen as a disadvantage with talk aloud protocols in certain test conditions. Although it should be kept in mind that [Chi et al 1989] showed that self-explainers perform better in problem solving.

In contrast to all the other studies, this was not a controlled laboratory experiment but consisted of observational studies and interviews. Although using ten participants makes this the smallest sample sized experiment surveyed, the participants are the most experienced all being experts from both

industry and academia with ten or more years programming experience.

Unlike the participants of Boehm-Davis et al, participants here could design a solution to one of four problems or to take a problem of their choice. Programmers were told to imagine themselves free of coding restrictions and it should be noted that they did not have to implement any solutions whatsoever. The four types of programs were: 1. Noughts and crosses, 2. Academic timetabler, 3. Lexicon for sub-anagram solver, 4. Pinball path predictor.

Differing from the other means of gathering comprehension information, participants here were prompted with questions during the programming tasks whenever the participant showed signs of deep thought. These questions focused on what the participant was using as their mental image, and what it looked like, in order to solve the problem. For example, "*What colour is it?*" and "*What's there that you can't see?*". Further examples of questions can be seen on page 114. Similar to Pennington's experiment, notes were allowed to be made and later examined.

With no other surveyed experiment having such data, the imagery described by the expert programmers was detailed, for example, "*text with animation*". Significantly, all experts described sound as part of their imagery. Other images were greater than four dimensions, and all described interaction. One said "*It's like describing all the dimensions of a problem in 2D, and the third dimension you're putting closeness to a solution*". There was also very strong spatial imagery (e.g. landscapes). "*..it's on the horizon, so I can keep an eye on it, but I don't really need to know...*". All of the described images were dynamic, but participant to control so that the rate could be varied, or the image could be frozen and some even permitted the events to be reversed. The authors detail that the "*experts chose where to put their attention at any given moment, and different regions of the imagery were described as comming in and out of focus*". All of the imagery could accommodate incompleteness. It is interesting that all of the experts reported using more than 4 dimensions, the extra dimensions were additional information such as overlaid data flows, or links to external representations. Also the experts talked about labelling entities in the imagery.

It is improtant to note that the imagery described here is for construction and not debugging. Two descriptive comments were : "*..the possibilities of debugging at bottom level from here are zero*" and "*In debugging, you only do it mentally for the difficult ones: intermittent, incomplete capture of the stimulus...*". Petre and Blackwell note that programmers, like designers, believe that much of design in non-verbal.

In summary, the common elements were: 1. multiplicity of modalities, 2. stoppable dynamism, 3. variable selection, 4. provisionality and incompleteness, 5. adjustable granularity, 6. extra dimensions, 7. simultaneous multiple images. Petre and Blackwell go on to say that "*experts have a tendency to create a visualisation for a particular problem (e.g. a specific data structure) even if it will never be useful for another problem*" i.e. a custom visualisation.

To conclude, this paper is a *goldmine* for designers of SV tools. It is

striking to note that all of the experts described sound as part of their visualisation, which were often dynamic and greater than four dimensions. Also, evidence is provided for the ability to label elements of a visualisation. Support is found here for the repetition of the spatial tests used in [Cunniffe and Taylor 1987] for future experiments, as many experts described spatial imagery. However much of the data helps in the designing of SV tools, it is important to keep in mind that no code was written and that these descriptions were only designs of how the developer would go about solving the problem and not how the developer would go about debugging. There is a lot of room here for future study.

3.5 Storey et al : *How Do Program Understanding Tools Affect How Programmers Understand Programs ?*

Thirty university student programmers (five graduate students and twenty-five senior undergraduate students) were the participants used here [Storey et al 1998]. The aims were to study, 1. the factors affecting the students choice of comprehension strategy, 2. to observe if the three tested tools aid in the comprehension (Rigi, SHriMP, SNiFF+), 3. to devise a means to characterise the more effective tools, 4. and to provide feedback for developers of comprehension tools.

Taking the same length of time as [Boehm-Davis et al 1987], the two hour experiment consisted of the following time limited phases: 1. Orientation 5 mins, 2. Training Tasks 20 mins, 3. Practice Tasks 20 mins, 4. Formal Tasks 50 mins, 5. Post-Study Questionnaire 15 min and 6. Post-Study Interview and Debriefing 10 min.

Emphasis was placed on training in three of the stages. Firstly, during the Orientation, where basic features of the tool were taught. Secondly during the Training, where a limited set of tool features were demonstrated. Finally, during the Practice Sessions, where the participant completed some tasks in order to become familiar with the tool. This training *Hangman* program written in C was larger than Penningtons' program, consisting of twelve files and three hundred lines.

As part of the Formal tasks, the participant was videotaped, and asked to *Think-Aloud*, as was done in Pennington's experiment. Worth noting, was the fact that two programs were employed, one for the training and the other for testing which was a *Monopoly* program (1700 loc, 17 files, with complex control flow).

The task questions are listed in the paper and can be referred to as needed. As with so many of the reviewed experiments, the tasks did not have to be implemented.

The Questionnaire had fifteen randomly ordered questions (five sets of three). A popular questionnaire design was adopted where the questions in a set were subtle rewordings of each other to prevent the chance of misinter-

pretation or an erroneous answer. The Interview and Debriefing, asked the opinions of the participant in order to gather information that the questionnaire could not.

The Questionnaire answers were rated on a scale of 1 to 5, from strongly disagree, agree, disagree, neutral, agree, to strongly agree. It was found that for "*pleasantness of use and confidence in results*" the results were not statistically significant. For the "*ability to generate results*", Rigi was rated worse that SHriMP and SNiFF+. For the "*ability to find dependencies*", Rigi was rated better than SHriMP and SNiFF+. Also, no significant differences were found between the SHriMP and SNiFF+ tools.

Storey et al note two main biases in the experiment, firstly, it was found that the experimenter forgot to show an essential feature of a tool and this significantly affected the comprehension strategies used. Secondly, the experimenters were also designers of Rigi and SHriMP.

In conclusion, the biases are strong enough to affect the results. However three tools are compared and some useful information can be learned for future comparative studies. For example, the issue of training that was mentioned in the conclusion of Section 3.2 has re-emerged here. Evidence such as this supports the argument for a greater emphasis to be placed on training during the evaluation of SV tools. Again, while the programs used were appropriate, a study which requires the participants to write or modify code would also be valuable, especially when it comes to evaluating future tools.

4 Conclusion and Future Issues to Address

In conclusion, there is mixed evidence for the effectiveness of *Software Visualisation* and not all of this evidence is in directly related fields of study i.e. much evidence can be gathered from the software comprehension field. This points to the fact that many more experiments focused on *Program Visualisation* need to be run. Many issues need to be addressed in future studies and these will be discussed in turn.

The training issue was highlighted in Cunniffe and Taylor's study and again in the study by Storey et al. SV tools can be quite complex to master, the decision to spend time learning a new tool can pay off in the future when the user becomes proficient with all the features. This must be kept in mind during the evaluation of such tools. As stated in Section 3.2 there are two choices. The training can be provided prior to the experiment (e.g. a couple of weeks) to allow the participant to become familiar with the tool over a longer period, or the tool can be integrated into a well known environment that is already familiar to the developer.

Sample size is quite a controversal issue, while it is obvious that larger sizes are needed, it is not so straightforward in practice. Busy software professionals are both difficult to find and recruit in large numbers. As well as this, the data from *Talk-Aloud* protocols is extremely voluminous

to manage, generating huge amounts of post experimental work for each participant. One possible avenue of light here, is to research the possibilities of experimental replication. Since replication is a large area in its own right, reference may be made to Daly's thesis [Daly 1996] for further examination.

Of all the surveyed experiments, [Cunniffe and Taylor 1987] was the only study to measure spatial aptitude. This could prove to be a factor in *Program Visualisation* experiments. Especially considering the results from Cunniffe and Taylor's study where accuracy was linked to spatial aptitude, making this is an area for future research.

A need for future experiments to include all their details is also evident, for example, the amount of time spent on the experiment is unknown in [Cunniffe and Taylor 1987]. This helps future researchers to design their own empirical study based on current and previous research, but this is made somewhat harder when details are missing, especially if a replication experiment needs to be run.

Learning affects were addressed by [Cunniffe and Taylor 1987] where both versions of the source code were similar except for variable names in order to reduce training effects and repetition. These affects are also addressed by [Storey et al 1998] where the questionnaire was constructed of groupings with questions of different rewordings. These learning affects are an important point to take into consideration during the design of future experiments and should be minimised as well as possible.

The design of the test programs are also important to examine, for example, the Boehm-Davis et al study used three design types i.e. in-line code, functional decomposition and object orientation. The difference is clear when compared to Cunniffe and Taylor's program segments. Future experiments to evaluate *Program Visualisation* tools need to consider using programs that are representational of current practices, for example, a test program containing design patterns may be considered during the evaluation of such tools.

Basili speaks about the tell-tale signs that show a field is maturing [Basili 2002]. Maturity of a field is seen when the "*level of sophistication of the goals of an experiment increase*", when "*understanding interesting things about the discipline becomes apparent*", and when a "*pattern of knowledge*" can be built from a series of experiments. Since this paper cannot be an exhaustive reference of the experiments performed to date, a number of varying examples were selected. Future experiments can build upon past experiences and use the current empirical body of knowledge to evaluate tools with the aid of professional software engineers.

References

[Atwood and Ramsey 1978] Atwood, M. E., Ramsey, H. R., *Cognitive Structures on the Comprehension and Memory of Computer Programs: An Investigation of Computer Program Debugging.* Tech. Rep. TR-78-A21. Alexandria, VA: U.S. Army Research Institute, 1978.

[Basili 2002] Basili, V., *Experimentation in software engineering.* talk given at 5th Workshop on NSF-CNPq Readers Project, Salvador, Brazil, January 2002.

[Boehm-Davis et al 1996] Boehm-Davis, D. A., Fox, J., and Philips, B., *Techniques for exploring software comprehension*, Empirical studies of programmers: Sixth Workshop, pages 3-37, 1996.

[Boehm-Davis et al 1987] Holt, R. W., Boehm-Davis, D.A., Schultz, A.C., *Mental Representations of Programs for Student and Professional Programmers*, Empirical Studies of Programmers: Second Workshop, pages 33-46, 1987.

[Buschke 1977] Buschke, H., *Two-dimensional recall: Immediate identification of clusters in episodic and semantic memory..* Journal of Verbal Learning and Verbal Behaviour, 12, 201-206, 1977.

[Chan and Munro 1997] Chan, P., Munro, M., *PUI: A Tool to Support Program Understanding* Proceedings of the fifth International Workshop on Program Comprehension (IWPC '97), pages 192-198, IEEE Computer Society, 1997.

[Chi et al 1989] Chi, M.T.H., Bassok, M., Lewis, M.W., Reimann, P., Glaser, R., *Self explanations: how students study and use examples in learning to solve problems.* Cognitive Science, 13, 145-182, 1989.

[Cunniffe and Taylor 1987] Cunniffe, N., Taylor, R., P., *Graphical vs. Textual Representation: An Empirical Study of Novices' Program Comprehension.* Empirical Studies of Programmers: Second Workshop, pages 114-131, 1987.

[Daly 1996] Daly, J. W., *Replication and a Multi-Method Approach to Empirical Software Engineering*, PhD Thesis, www.cis.strath.ac.uk/research/efocs/abstracts.htmljd$_t$*hesis*, 1996.

[Ekstrom et al 1976] Ekstrom, R. B., French, J. W., Harman, H. H., *Manual for Kit of Factor-Referenced Cognitive Tests.* Princeton, NJ: Educational Testing Services, 1976.

[Hitch et al 1995] Hitch, G.J., Brandimonte, M.A., Walker, P., *Two types of representation in visual memory: evidence from the effects of stimulus contrast on image combination.* Memory and Cognition, 23, 147-154, 1995.

[Littman et al 1986] Littman, D.C., Pinto, J., Letovsky, S., Soloway, E., *Mental Models and Software Maintenance*, Empirical Studies Of Programmers: First Workshop, pages 80-98, 1986.

[Myers 1986] Myers, B. A., *Visual programming, programming by example and program visualization: a taxonomy.* Proceedings of the 1988 IEEE Workshop on Visual Languages, pages 192-198, 1986.

[Pennington 1987] Pennington, N., *Comprehension Strategies in Programming.* Empirical Studies of Programmers: Second Workshop, pages 100-113, 1987.

[Petre and Blackwell 1997] Petre, M., Blackwell, A. F., *A Glimpse of Expert Programmers' Mental Imagery.* Empirical Studies of Programmers: Seventh Workshop, pages 109-123, 1997.

[Sheil 1981] Sheil, B.A., *The Psychological Study of Programming*, ACM Computing Surveys, Vol 13, Number 1, pages 101-120, 1981.

[Stasko, Domingue, Brown, Price 1998] Stasko, J., Domingue, J., Brown, M.H., and Price, B.A., *Software Visualization: Programming as a Multimedia Experience*, ISBN 0-262-19395-7, MIT Press, 1998.

[Stasko 1990] Stasko, J.T., *Tango: A Framework and System for Algorithm Animation* IEEE Computer 23, pages 27-39, 1990.

[Storey et al 1998] Storey, M. AD., Wong K., Muller H. A., *How Do Program Understanding Tools Affect How Programmers Understand Programs ?.* Science of Computer Programming Journal, Vol 36, Issues 2-3, pages 183-207, 2000 (paper from 1998)

[Von Mayrhauser 1995] Von Mayrhauserm, A., and Vans, A.M., *Program Understanding: Models and Experiments*, Advances in Computers, Vol 40, Academic Press, 1995.