In G. Kadoda (Ed). Proc. PPIG 13 Pages 147-159

Long Term Comprehension of Software Systems : A Methodology for Study

C. R. Douce
Feedback Instruments
Crowborough, East Sussex, UK.
Chrisd@fdbk.co.uk

Keywords: POP-II.B. *Maintenance* POP-V.B. *Research Methodology*

Abstract

Studies of software comprehension often use short-term recall as a way to study comprehension. Experiments range from broad descriptions of program purpose to tests that require subject to 'fill in the gaps'. There is little doubt that human memory is a very important issue when it comes to understanding how software systems work. This paper addresses the topic of 'long term' comprehension, namely, the retention of programming knowledge over a period of months, years or even decades. Long-term retention of programming knowledge is thought to be an area within the psychology of programming literature that has received surprisingly little attention. To stimulate debate, methodological issues that may affect the study of long-term comprehension are detailed. Finally, a planned experiment based upon related psychology of programming studies is outlined.

Introduction

Computing is filled with interesting and often informative 'folklore'. One possibly exaggerated rumour, and one that feels as if it must be 'true', begins in a large city, perhaps in the US. A key software developer in the financial sector is close to retirement age. This individual has worked for the very important organisation for a very long time and has held a very important role. The managers realize how important this individual is, or more precisely, , how important his or her knowledge is to the day to day running of the corporation. Software engineering courses emphasise the issue of documentation to students by asking them to consider what may happen if an important developer were to be suddenly 'knocked down by a bus' one morning. The managers of our illustrious company decided not leave anything to chance. Every morning, our knowledgeable 'techy' was collected by a big limousine, driven by a company chauffeur. Other embellishments to this story would have emphasised the length of the limousine, its bullet-proof glass and the astounding luxury of its seats to emphasise how important this key individual was. The journey to work and home occurred for several months until another company employee gradually acquired knowledge about the intricate workings of the organisations computer systems or any critical development was complete. The accident prevention safeguards if we again return to embellishing this thoughtful story would have cost the company obscene amounts of money.

When an employee leaves a technical department in a company that is involved with the development, deployment and maintenance of computer systems, a lot of knowledge of the workings of the systems within that company leaves with the employee. This is, of course, good news for the software developer who dreams of chauffeur driven limousines as a way to commute to work, but is not good for the organisation that has to fight developer flight with incentives and wages. Software is, of course, incredibly knowledge intensive. It requires knowledge of platform architectures, problem domains and programming languages. Luckily, these are three things that are transferable and, ultimately, replaceable. What cannot easily be replaced is knowledge about a particular software system, their architecture, object-structure, problems and design nuances. Such knowledge can take months, if not years to acquire and is, of course, expensive for a corporation to develop.

The following section discusses the issue of long-term programming memory. This is followed with a very brief discussion of program and software knowledge, discussing literature relevant to the psychology of programming where several relevant studies are discussed. The fourth section

addresses the issue of methodology and tries to grapple with the thorny issue of subjects and experimental materials. Materials, especially when used in conjunction with understanding how long-term memory and comprehension is considered to be an issue of great importance, and is believed to be one that should receive significant attention.

Since the aim of this paper is primarily to stimulate debate amongst the psychology of programming community, there is little in a way of a conclusion, other than the common oft repeated call of, 'we need to do more research in this area'. However, the penultimate section describes a study that is currently being prepared and is intended to be executed towards the end of this year.

Long Term Comprehension of Software

Long term comprehension of software is considered to be knowledge of a program or a series of programs over a period of months, years or decades. During a software re-engineering project, there is nothing nicer than being able to talk with the original designer. During such a conversation, ideas that may take weeks to understand without help may be able to be grasped in an afternoon of discussion and drawing. Using the original designer as a source of information not only draws on 'code comprehension' but also 'design comprehension'. Design comprehension is very important, but in this paper 'long-term comprehension' is specifically associated to source code comprehension. It is true that they are very closely related, since it is obvious that code can be, or ought to be, a realisation of a given design.

Little is known about the impact that long-term comprehension may have on programmer performance, specifically in the maintenance of existing software. A programmer, for example, writes a software product for an engineering control domain. Five years later, luckily, the programmer still works for the same organisation. New hardware is to be used in a control system, and changes must be made to ensure that the software is still saleable. Instead of using a newly employed programmer, the original programmer is recalled from a more senior position to undertake the development work. Studies of programmer strategy makes up an important part of psychology of programming research. It is currently unclear what effect existing programming knowledge may have on strategy. Will a 'recomprehension strategy' be comparable or substantially different to an equally competent programmer undertaking the comprehension of the same system from scratch?

From a software engineering perspective, it is not known whether using the original programmer may have any immediate effect on the products final reliability than if a 'naive' programmer had been utilised. Of course, trying to study this idea is plagued with a host of methodological difficulties, such as individual differences, both of the domain and of the programmer.

Another issue is whether there is a dominant a form of knowledge or representation of a program or software system that could be more easily recalled than other types of knowledge. Take our software expert who is chauffeured to the office every morning. This expert may be faced with code that he or she was involved with developing approximately ten years ago. Some fragments of the program code may strike our lucky developer more readily than others. Indeed, we could even hypothesise that our developer may use certain types of long-term slices of forgotten code, rather than others. The issue of representation can again be brought back to the issue of software design. Modern software design practices emphasise the usage of multiple complementing views. Little is know whether some views are more easily recalled than others, although it is very tempting to draw introspective but unconfirmed conclusions.

Knowing more about how the long-term memory of a programmer works, and what is immediately seen when a programmer faced with vaguely familiar code is considered important to several groups of people. A software development manager would like to quantify the value of the invisible. For the cognitive psychologist, code could represent an interesting experimental material. For the software engineer, it provides the tool designer another avenue to consider. Current software tools constantly focus upon the present and the past, but perhaps they should also consider the future, when things are not so clear.

Programmer Knowledge

From the outset, programmer knowledge, and implicitly, programmer memory has always been of interest to researchers who have a fascination with the psychology of programming. One of the earliest models of programmer memory concerns a distinction between 'stable' semantic knowledge about programming concepts, and more 'fragile' syntactic knowledge that has to be learned by rote (Shneiderman and Mayer 1979). A later proposition by Soloway hypothesised that programming knowledge can comprise of plan knowledge and discourse knowledge (Soloway and Ehrlich 1984). Plans are considered to be common programming concepts, such as searching, sorting and how to form other useful algorithms. Discourse knowledge relates to the knowledge of how programs are to be written. Programming plans are closely associated with research related to working memory where significant abstractions, considered to be *chunks*, can be held. Chunks are associated to stable patterns that are held in long-term memory. Regarding computer programming, it has been hypothesised that 'plans' can be recognised through programming beacons, which can be detected through the application of a comprehension strategy (Brooks 1983; Wiedenbeck 1986).

Software engineering has recently developed the notion of a 'design pattern', akin to patterns used in architecture. A pattern can be described as a core to the solution of a problem that occurs over and over again within a particular problem domain (Gamma, Helm et al. 1995). Instead of re-inventing a solution, an outline can be utilised and can be given a label allowing a considered solution to be easily discussed amongst groups of programmers. The notion of a 'programming plan' appears to quite close to the notion of a design pattern. Both represent abstractions and can be interpreted as ways to comprehend software.

Pennington, taking a slightly different approach, performed an influential study where several different representations of a single program are studied (Pennington 1987). A question was posed as to whether certain views of 'abstractions' of a program may dominate. Four abstractions were considered. The *functional* abstraction concerns the overall 'goals' of the program - what it is intended to perform. The second abstraction can be considered to be the *data flow* abstraction. Within a program, data passes through a series of transformations during its life. The data flow abstraction represents what may happen to such data. The *control flow* is the sequence of execution, or the order in which program actions occur. Finally, the *conditionalized* or *conditional* abstraction relates to actions that are performed when a particular set of conditions are met. Pennington concludes that, for her experiments 'the natural representation of programs is procedural'. She then goes on to cautiously state, 'at least for the programs written in traditional programming languages'. Different programming notions can emphasis differing 'abstractions'.

Theories and models are important, since they present a picture that allows us to make sense of observable events. Models of memory are particularly revealing. The multi-store model of memory has provided us with the notion that memory can be divided into a short-term temporary store, a long term store, and modality specific sensory stores that decay rapidly while passing senses to our 'attention'. This model has given way to a more sophisticated model of *working memory* which takes account of both experimental psychology studies, and evidence obtained from cognitive neuropsychological cases (Baddeley 1997). Long-term memory has the ability to store many different items, over substantial periods of time. Facts are retained, such as names of the capital city of Italy, and knowing that a dog is a type of mammal. Long-term memory can hold semantic memory, autobiographical memory, and implicit knowledge. Psychologists continue to debate and conduct experiments to further understand structure of long-term memory to determine whether, for example, there can exist a category of memory called episodic memory, (Schacter and Tulving 1994) where knowledge of events and situations are held.

Attention has generally focussed on the programmers knowledge and how it is structured, rather than looking at programmer as an individual. What is meant is that the 'structures' the programmer uses are considered to be of great importance, not necessarily the fact that a programmer may have the potential to associate development activities with actions or episodic events that may take place outside the world of the text editor or debugger. In some cases, conversations could potentially be

used to aid comprehension. Sections of code may trigger memories of a development 'accident' which may have dramatically offended the aesthetic sensibilities of another programmer. Many of these events go together to make-up the autobiography of a software developer. The suggestion that the recall of events external to software may aid in its comprehension is pure conjecture.

Methodology

The focus of our attention is the programmers long-term memory. The main issue is not whether we need to study it, for its study is bound tell us something about the activity of programming, but *how* to study it. We have described, very broadly, some categories of information a programmer could feasibly remember when faced with a complicated program that had been viewed several years ago. The key problem being how we can get to know what a programmer knows, and when we know this we can begin to determine what the programmer may remember successfully, and what he or she may have forgotten.

Source code is the raw material of the programmer. In the majority of cases, it the material that the programmer uses to build software. Source code is what the programmer knows intimately. It can tell us about the design of a system, some details regarding its history, and often provide us with a some, albeit limited, information about who crafted it. Viewing source code allows us to comprehend what the original programmer comprehended. In doing so, we can come armed with a battery of questions or recollection devices to aid our study. Code is, in some ways, like a diary. Diary keeping is a recognised method of memory research, where several studies of this kind are described by Groeger (Groeger 1997). Subjects, often themselves, wrote down events on cards or in a diary that were later used stimulus material. Issues affecting recall concern how many times an event had been previously recalled, and how pleasant or unpleasant a particular event had been. Software engineers, no doubt, have the ability to recall how pleasant or unpleasant a particular software development experience once was. Although, quite certainly, source code will remain silent on this issue, source code positively shouts the relationships that exist between its component parts. Functions call other functions which are related to problem domains. Conditional statements are related to other conditional statements. Variables are used to build other variables which are then given to functions, or become a part of larger data structures. Source code can provide us with an enormous number of facts about data flow, control flow and conditional execution of instructions. Using source code, any knowledge that a 'retiring' programmer may give us about how a particular program may function can be easily verified.

In trying to understand what a programmer knows and what a programmer has forgotten or may need to view to facilitate re-comprehension, there are two 'ideals'; an ideal programmer and an ideal program. Beginning with the ideal program, firstly it must have only been developed with one programmer. Without modern source control and version management software, it may be impossible for an experimenter to determine who developed what, and when. Software development amongst groups of programmers is essential. The 'perfect' program, however, is one that has been developed by a lone programmer and one that solves a fairly short well-defined problem, requiring little specialist domain knowledge. 'Real' software relies heavily on domain knowledge. Software developed for the self-referential arena of 'computing' may utilise easily accessible domain knowledge of the programmer.

The ideal programmer is similarly difficult to find. By its nature, the effect of studying long-term programming memory could have an immediate effect on a programmers recall effectiveness. The ideal subject is one who has not engaged with the 'ideal' source material for some length of time and has not taken up any development work in another organisation where the domain or problem are similar. Given the specialised nature of software development, finding a perfect combination of these two 'ideals' is practically impossible.

In some cases, a diary study can only yield interesting results if parts of the diary are studied in isolation. Reading a diary from cover to cover after many years will invalidate an experiment

designed to test accuracy of autobiographical knowledge. Similarly, providing a listing or a related graphical representation to a programmer will inspire silent 'software related' recollections, which may be precisely what we wish to study. It is proposed that any stimulus fragments or slices of code is removed out of context from the original software. Using only fragments, an experimenter could ask specifically tailored questions. An experimenter could ask what functions could be called from 'this' function to determine control-flow knowledge. Similar questions may be asked about conditional statements, 'what conditions must be met to ensure that this fragment is executed?', for example. Small cards or a display screen could be utilised to administer the fragments. Answers could be later checked against the known 'execution truth'.

The use of fragments or code slices to facilitate software comprehension is by no means a new idea. Weiser proposed that a mechanised system could produce a slice that is used to represent a subset of its behaviour (Weiser 1981). A program slice is then used to comprehend how a particular section of code functions, facilitating understanding. The slice is specified by a defining 'slice criterion'. Weiser describes a case where a particular variable may be of interest, its data-flow abstraction, to a programmer, and slices could be provided with the intention of aiding programmer comprehension.

Probe questions can extend beyond the four abstraction categories outlined within the previous section. It is not known, for example, whether events during the time of a software development can be easily triggered by cards containing several lines of code. Subjects could be asked whether they can remember anything else in connection with a 'slice'. As suggested earlier, a small set of instructions may inspire the recollection of the installation of a new software tool or development environment. Similarly, a conversation between a co-worker could be recalled relating, perhaps, to the projects specification or requirements. Such recall episodes will have to be categorised using a scheme that should be clearly defined and be evidently reusable.

Here, we return to the issue of strategy. Program comprehension strategy is considered to be an issue of substantial importance. One of the considered deficiencies of knowledge-oriented models of comprehension is that they do not take account of how knowledge is utilised (Davies 1993). Another consideration is whether programmers, when faced with vaguely familiar code, adopt any form of discernible 'recall' strategy. Such a consideration is incredibly tentative and is clearly open to further assessment and debate.

Another source of information that could be utilised includes formal software documentation that describes changes that a product undergoes during its lifetime. Such documentation could include problem reports, change requests and a whole series of records that ensures that any changes made to a software system are traceable. Documents can contain important information such as dates, times, details of necessary software tools and individuals who carried out modifications, not only to code, but also to surrounding documentation or graphics. Such information constitutes a sophisticated change 'diary' that could be used to drive an exploration into programmer knowledge. A document can describe a software 'change episode' or 'change event'. It would be interesting to examine the degree and accuracy a programmer may be able to recall documented episodes. In doing so, we may be able to see whether there are any significant differences between episodes that are remembered, and those that are forgotten.

Studying long-term programmer knowledge is somewhat different to the more 'repeatable' studies of program comprehension. An experiment would be more of a 'case study'. Variables will change, possibly dramatically, between cases and agreement between observed phenomena can only be achieved through the publication of comparable studies. The construction of experiments is a labour intensive activity, requiring careful attention to the selection of both materials and subjects. When materials have been selected, thorough analysis is required to determine whether they are suitable. In many cases, they may not be able to be used. Documentation may be incomplete, for example. It may not be possible, with a great degree of certainty, to attribute particular developments to specific developers. When suitable materials have been identified, substantial work is needed, possibly through the application of design recovery and other software engineering tools. Fresh comprehension may have to be undertaken so that the distant understanding can be gauged.

At present, the study of long-term comprehension can only be described as exploratory. The following section attempts to further expand this section on methodology by detailing a proposed study of programmer long-term comprehension.

Proposed Study

Two programmers have been found who worked as engineers for a company designing educational engineering training systems. Both programmers left the organisation some years ago to take up other positions, but have kept in contact with colleagues within several departments. One of the developers worked on several software developments over a ten year period, ranging from process control training systems, to telecommunication training systems. The other programmer developed a small number of instrumentation packages for an electrical machines trainer. Several versions of source code relating to all of the key products are available and permission has been obtained for the code to be used as experimental material.

The proposed study is to take part in three stages. The first stage comprises of a structured interview with each programmer. Its purpose is to explore whether the subject is considered to be acceptable for experiments, and to determine the extent that certain materials could be utilised within later experiments. The interview is to begin with a background of the programmer, and is then to follow with questions regarding the projects that the programmer worked on. Questions are designed to determine the extent of development work, degree of design involvement and the length of the project and approximately how long ago the development took place. Also of relevance, is whether the programmer worked on the project with any other engineers. Other information includes number of programming languages known, whether subject has had formal software training and how long has been involved with software development for. Information relating to documentation practices will also be captured. For completeness, subjective opinions of particular projects should also be recorded, whether a project was enjoyable or frustrating, easy or challenging.

The second part comprises of a series of experiments. Each experiment is dedicated to a particular project that the subject had worked on. A series of cards are produced, where a fragment or program slice derived from the project source code is printed. Slices are divided into four types, relating to the type of probe questions they are intended to give. Distracter cards are interspersed amongst the 'real' cards which contain 'nonsense' fictitious, but grammatical, code. On each card, all possible comments are removed. Identifiers such as variable or function names remain unchanged. All cards are to be numbered, which relate to a set of probe questions that correspond to the selected slice.

Function abstraction probe comprises of questions akin to, 'what does this function do?'. Data flow probes correspond to questions like, 'before this point, how has the data changed?' Control flow questions comprise, 'which function is this function called by?' or visa-versa. Finally, conditional questions correspond to, 'what causes these lines to be executed?'. Before any of these questions are answered, the first question, 'do you recognise this code?' is asked. If not, the card is discarded. The last question, 'can you remember anything else about these lines?' is designed as a general probe to elicit any other information that the stimulus material may inspire. Indeed, this last probe may elicit information relating to any of the previously mentioned abstractions.

During the experiment, all proceedings are to be tape recorded. Following the experiment, results are deemed to be correct if the answer could be substantiated by some evidence found within the source code. If, for example, a function is claimed to called by a function several layers above and not a single layer above, the result would be deemed to be correct. Additional responses to probes should be reported, and all results tabulated, along with any source code anomalies that may cause disruption to the results.

The third and final section comprises of a post-experimental discussion. During the experiments, the slices may have simulated recall of significant facts regarding time spent working on certain development projects. Here, additional questions are asked, for example, whether any

significant development difficulties were faced and whether the development was remembered as one that was pleasurable or challenging.

To aid case-study interpretation, the source materials should be described in some depth. This should begin with a broad description of the problem domain, the age of the source code, how long the software had taken to develop and, importantly, how often and how great were revisions that were made to the system. To obtain an understanding of the size of the software, common software metrics should also be included. If significant revisions were made, any data relating to the size and complexity of various releases should also be included.

Conclusions and Further Work

At present, the only conclusion reached is that there is a substantial body of fascinating memory literature that could potentially be of direct interest to researchers studying the complexities of how programmers can retain an understanding of computer software over significant periods of time. The surface has merely been scratched. There includes a large number of interesting debates that is of direct relevance to this field.

One possible methodology that could be used for the study of long-term program comprehension has been presented. It is also concluded that the proposed experiment, as detailed within the previous section, like many psychology of programming studies are fraught with methodological difficulties. It is not known what the proposed experiment may reveal. It is also not known how any of its results may be ultimately interpreted.

Software developers hardly work in isolation. They work within small groups or unique cultures. Memory about software does not only originate from the individual programmer, but from the collective knowledge of several. Knowing 'facts' about software may be an important issue, but may not be the *most* important issue. It is suggested in the previous section that developers are asked *what* they may know about small fragments of code, and not *where* they may look, or *who* they may ask to find an answer to the question. Individuals are important, but so is the environment in which they work and all the associated organisational structures which are used to support their day-to-day activities.

From a software engineering perspective, it is the day-to-day work that is of greatest interest. Reliable software is needed to be constructed on time and within budget and be able to be malleable enough to be used within a world of changing technologies and user requirements. Perhaps by looking at ourselves in the ways that have been described, using slices as a tool for exploration, can allow us to reflect more clearly upon the activities and mechanisms that software developers use to achieve success.

References

Baddeley, A. (1997). *Human memory : Theory and practice* - revised edition. Hove, Psychology Press.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18, 543-554.

Davies, S. P. (1993). Models and theories of programming strategy. International *Journal of Man-Machine Studies*, 39, 237-267.

Gamma, E., R. Helm, et al. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts, Addison-Wesley.

Groeger, J. A. (1997). *Memory and remembering : everyday memory in context*. Harlow, England, Addison-Wesley Longman.

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19, 295-341.

Schacter, D. L. and E. Tulving (1994). *What are the memory systems of 1994? Memory Systems* 1994. D. L. Schacter and E. Tulving. Cambridge, Massachusetts, The MIT Press**,** 1-38.

Shneiderman, B. and R. E. Mayer (1979). Syntactic/semantic interactions in programmer behavior: a model and experimental results. International *Journal of Computer and Information Sciences*, 8, 219-238.

Soloway, E. and K. Ehrlich (1984). Empirical studies of programming knowledge. *IEEE Transactions of Software Engineering*, SE-10, 5, 595-609.

Weiser, M. (1981). Program slicing. *Fifth International Conference on Software Engineering*, IEEE.

Wiedenbeck, S. (1986). Beacons in computer program comprehension. *International Journal of Man-Machine Studies*, 25, 697-709.