In P. Romero, J. Good, E. Acosta Chaparro & S. Bryant (Eds). Proc. PPIG 17 Pages 215 - 226

Sidebrain: a sidekick for the programmer's brain

John Sturdy

University of Limerick
john.sturdy@ul.ie

Keywords: POP-V.A. goal structure; POP-V.A. attention investment; POP-V.A. short-term memory; POP-II.C. working practices; POP-III.D. editors; POP-V.B. protocol analysis; POP-I.A. distributed teams.

Abstract. It has been observed that the activity of programming is not a linear progression, but a complex hopping between many threads[1]. It appears that remembering all the necessary threads, and directing attention appropriately, may exceed the limits of human working memory[2]. This paper describes work in progress to design and create a tool which models and supports programmers' mental activity by external assistance to their working memory, both for gathering of information and for direction of attention. It presents some initial findings from the use of a prototypical implementation, and puts forward some suggestions for experiments based around the tool.

1 Introduction

Programmers are often seen to note down information on paper or electronically, mostly while debugging but also while developing new code, especially that with many connections with existing code. Three common types of such information are:

- uncompleted tasks to return to after finishing a task
- other tasks to do, thought of while working on current task
- observations made, typically as the end result of a task

Another observation connected with programmers' handling of working information is the common problem of not wishing to leave a piece of work for fear of finding it harder to pick up again after a break; this can cause various forms of stress.

1.1 An initial survey

To verify these ideas, a pilot survey (presented at the end of this paper) was carried out among some experienced programmers. The returns from this confirmed the following observations:

- that programmers' memory limits[2] are routinely exceeeded and external memory is necessary
- that this is typically done both on the computer and on paper
- the branching nature of the thought and behaviour patterns during programming (described as "*opportunistic processing*"[1] in the context of program comprehension)

– that this is one of the drivers of the need for external memory

Typical replies included:

– “Typically branching. Very like web-browsing experience.”
– “If I did need to note a future action I would use notepad or some other electronic method for recording it.”
– “Generally scribble in logbook - bulletpoints, quick doodles. Nothing formal. Am fairly rigorous in terms of filing documents, emails etc. in folder hierarchy”

1.2 Developing a tool to help

These findings, and some work on annotating keystroke logging with the start and end of tasks and subtasks[1], led to the idea of writing a tool to organize working information as:

– a stack of open tasks
– observations (discovered knowledge)
– a queue or pool of pending or suspended tasks

The tool has been named “Sidebrain”, since it is meant to work alongside the programmer’s forebrain. During implementation and early testing it proved to be a useful tool, and it also opened up a number of further research questions, many of which may be explored through the history log that Sidebrain produces.

2 The implementation

The first implementation of Sidebrain was written as an extension to the programmable editor and development environment GNUemacs[3], which is widely used by software developers (by reputation, mostly by experienced developers). It would be possible to implement a similar system in other programmable editors and IDEs, and also to make a standalone implementation.

As part of the implementation it was decided to use an XML[4] based file format for its persistent data, to make it possible to exchange data between implementations written in different languages.

2.1 The data structures

Sidebrain is meant to support the programmer’s working memory, and so must have data structures that represent a simple model of the parts of working memory that programmers find limiting.

The main unit modelled and represented in this software we call the “task” – a simple description of what the programmer is trying to do. This consists of a short

[1] The process of annotating work while it was in progress turned out to make the work annotated more efficient and directed.

piece of text (one line). Tasks can be entered either via the keyboard or, on a suitably equipped system, by voice.

We model the programmer's memory of the tasks they are working on as a stack: while working on one thing, the programmer realises that they are starting a distinct task within it, at the end of which they will return to the previous activity. For example, in the information gathering of debugging, the programmer may be looking for the reason why a routine `a` is not being called, and find that it is only called when the variable `x` is not zero. Finding why `x` is set to zero would then be a sub-task of finding why `a` is not being called.

The tasks are arranged into a stack of open tasks, onto which the programmer can push and pop them. The task stack is displayed continuously, typically in a small window towards the edge of the screen (see figure 1), and is meant to keep the programmer focused on what they are trying to achieve. Subtasks are displayed above their parent tasks, with the current task being displayed at the top of the window.

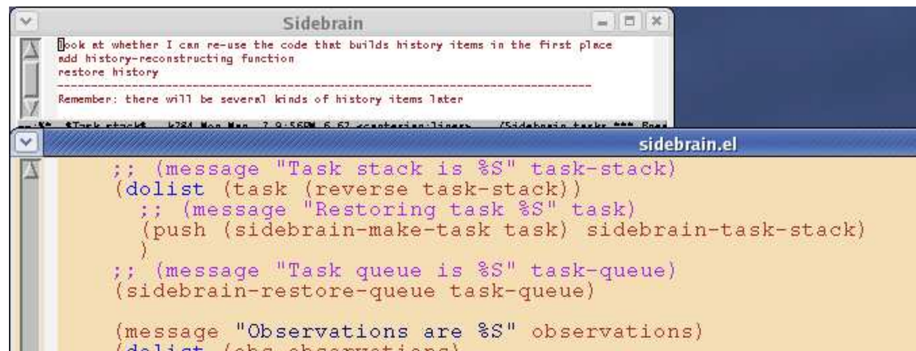

Fig. 1. The task stack display

Attached to the task stack, and displayed just below it, is a collection of observations entered by the programmer as they go about their information gathering.

To support context switches, Sidebrain provides a queue of things that the programmer intends to do, or resume doing, later. The queue is kept as an ordered list, each entry containing:

– a label naming the entry
– a task stack
– a list of observations

By default, the label is derived from the bottom (most ancestral) task of the task stack, thus naming the queue entry by the high-level description of what the programmer is trying to do. Labelling entries allows them to be selected by name in any order, so the queue can also be used as an unordered set or pool of tasks. Further investigation will be needed to see whether the ordering of the tasks is important; for example, it may turn out that they will always be referenced by name, and never picked in order.

For simplicity, the initial implementation includes neither prioritization nor grouping of tasks in the queue. It may be useful to provide these in later versions.

The programmer can view the task queue on demand, but, unlike the task stack, it is not normally continuously displayed. The labels can be used for selecting an entry by name, or the queue can be presented as a window in its own right, to be browsed with cursor movements, scrolling and searches.

Another data structure, which the programmer will not normally need to see, is the task history, into which tasks are transferred when they are completed. This could be used for Personal Software Process[5] monitoring, or for analysis for psychological research. To aid such use, in addition to the text string, each task contains the following data:

- "`begin`" and "`end`" time stamps
- the time for which this task has been top of the stack
- indications of what the programmer was editing at the time, in the form of filename and line number

These additional details are not displayed.

2.2 The operations

Sidebrain is operated through a small selection of commands; some of these ask the programmer to enter some text. For best results, use of these commands should be interleaved closely with commands to the rest of the development environment, particularly the text editor.

Here we describe the operations by name; in the original implementation, they can be accessed by name through GNUemacs' "`M-x`" command entry; they could also be bound to key sequences, or made into menu entries or toolbar buttons, and, where voice input is available, spoken commands.

The programmer pushes new tasks onto the stack with a "`begin task`" command, which prompts the programmer to enter a description of the task. This can be done through the keyboard, but is most conveniently done with voice input, which makes use of Sidebrain similar to talk-aloud, which has been described[6] as a rich source of information about the working of the programmers' mind – useful if using a log of Sidebrain operations for research.

The programmer pops tasks with "`end task`". It is also possible to end several tasks at the same time, by selecting a level on stack (in the stack display window) and telling it to end that task and all its subtasks.

Alternatively, the programmer can abandon a task, using "`abandon task`", which is the same as ending the task, but records it differently in the task history log.

The programmer can add an observation to the observations window using the command "`observation`", which also prompts for a text string. Sometimes a subtask may end with making an observation related to the task (particularly when the task was an information-gathering one), and it might be useful to have a command "`end task with observation`", which could make the task text available when entering the observation string.

When a programmer's work is interrupted, sometimes the interruption needs a new stack of tasks of its own, which in turn may need to be put aside to go back to the original stack, or perhaps to another. To support this, Sidebrain allows the programmer to put the whole stack of tasks aside (along with the associated group of observations), with the command "`suspend task`". (A possible extension here would be to make this be usable for part of the stack, provided the interface is not too complicated.) "`suspend task`" moves the task stack into the reminder queue, from where it can be resumed with the command "`resume task`". All tasks put into the reminder queue are given labels by "`suspend task`" (the programmer is prompted to edit the label should they wish to), and "`resume task`" uses the labels to select the task to resume.

The command "`browse tasks`" brings up a buffer containing the list of of suspended tasks, and the programmer can select from there a task to resume by moving on to it with the cursor and pressing "`enter`".

The programmer can enter a task onto the queue (without having to begin it and suspend it) with the command "`reminder`", and can begin such a task from the reminder queue with the command "`resume task`" or from the tasks browser, just as though it had been started earlier and then suspended.

2.3 Persistence

One inevitable form of interruption to programming is the intrusion of real life into the programmer's world, including some fairly drastic forms such as going home for the night (or, possibly, for the day), or, for the less dedicated programmer, social activities. These cause unfortunate breaks in the programmer's chain (or tree, or graph) of thought, and to address these occurrences when they are a problem[2], Sidebrain provides persistence via a facility to save its data structures to file, for example at the end of the session, and to reload them from file.

The saved data encompasses the complete state of Sidebrain:

- task stack
- observations
- task queue
- history

As stated previously, the saved Sidebrain data is in an XML-based format[4] rather than in GNUemacs' native Lisp format, for easy interchange with other implementations of Sidebrain, or with other systems.

The saved XML data can also be passed from one programmer to another with the handover of programming activities, for example in "*Follow-the-sun*" development[7]. The initial implementation includes commands for sending selected tasks by email (from the task browser), and for parsing an e-mail to extract XML tasks from it and insert them in the task queue.

[2] It is not always a problem; getting away from concentrating on something can be the key to solving it – see also the last question in section 3.3.

2.4 Integration

Sidebrain can read the traditional "`todo`" comments from source files, and pick them up into the task queue, and will also go back to the source file automatically to change the comment from "`todo`" to "`done`" when the tasks concerned are ended – an attempt to make it as natural as possible for a programmer to use this system, and to make it easy for those using the system to collaborate with others who are not using it.

The implementation of Sidebrain for GNUemacs integrates fully with its host environment, sharing copy-and-paste buffers, latest search string, etc.

3 Research questions

As well as being an experimental tool for exploring a variety of questions concerning the working memory of programmers, Sidebrain is designed to be a useful everyday working tool for programmers, and if it succeeds in this it has potential for behavioural studies with high ecological validity, as its history record is then produced as a side-effect of the programmer's normal work.

Some research questions that may be addressed using this tool are listed below, both concerning how effective this model of supporting programmer activity is, and using it as a tool for investigating other issues.

3.1 Validity and usefulness

A form of research planned in this project is to release Sidebrain for general use, and look for feedback from programmers on how well it fits their thought processes[8], and whether to some extent it affects their way of working. Sidebrain is built on a rather simple and mechanistic model of parts of the human memory; anecdotal evidence from conversations amongst programmers suggests that they, perhaps more than other people, tend to introspect on their mental activities in terms such as "stacks"[3], and such a tool as Sidebrain may to some extent encourage and further this. This may be an interesting point to investigate through subjective questionnaires, although the questionnaires themselves may have a similar effect.

Also, some more intensive case studies[9] will be appropriate, with the history sections of the saved XML files being collected and analysed. It should be possible to send the data gathering program to participants, to extract and return only the information required for the experiment, so that the participants do not have to send back the whole record of what they have done.

3.2 Remaining focused on the task

Sometimes, in the complex information-gathering of development or debugging, it is possible to follow a line of investigation into more and more detail, and eventually lose

[3] The author has repeatedly observed programmers using in technical conversation statements such as "push that" (meaning to remember it to come back to later), or "let's pop several levels" (meaning to go back to the topic of discussion from which the current topic had arisen)

track (perhaps being distracted from it by the detail) of the original task. Keeping a stack of open tasks visible on the screen is meant to serve as a useful reminder not only of what to return to, but also of why the programmer is trying to find a particular piece of information, thus helping to keep attention directed to what is relevant.

Research questions in this area needing empirical verification, probably from controlled experiments on complex tasks, include:

- whether being freed from having to remember the things to come back to may free for other use part of the programmers' working memory, and allow more attention to be directed to problem solving
- whether it makes a difference to have the task stack displayed continually, against having it viewable on demand, or having the newly exposed task displayed briefly as a side effect of ending its subtask (or, on a suitably-equipped system, spoken aloud).
- to what extent are parent tasks still relevant when the sub tasks have been completed? (It appears that the search may evolve during information gathering[10]) This can be derived from the use of "`abandon task`" instead of the normal "`end task`", as seen in the history log.

3.3 Keeping hold of other ideas

According to sources including Letovsky[1], and supported by my initial survey, the thought sequences needed for programming and debugging are typically not linear; often, while following one line of thought, something else that is useful, but not connected to the immediate task, may come to mind. When this happens, the programmer can either push what they were doing onto a stack of things to come back to, and follow the new line of thought, or put the new line of thought aside to come back to later, and continue the present task. Sidebrain is designed to help with both of these, with its queue or pool of tasks that are not part of the current stack. Research questions in this area include:

- Is this facility useful and effective?
- Is the programmer's mind is particularly stimulated to work on some particular problem when the attention is notionally directed to some other problem? (This will probably need manually-assisted analysis of the timestamps on reminders entries and the history record, to determine which new tasks are related to the task in hand.) This could be a form of displacement activity[11], perhaps, or possibly simply that stimulation work on one problem stimulates the mind generally? Something more than this tool alone would probably be needed to investigate this.
- Does the provision of a task queue memory remove some of the attention-directing load from the programmer's working memory, allowing more attention to be given to problem-solving?
- How often does the programmer note a task to come back to later by making a reminder for it, compared with how often they make it a subtask and do it immediately? Does this frequency correlate with productivity (or with creativity?), amongst a population of programmers? Does it correlate with cognitive styles[12][13]?

– How many reminders are generated during various kinds of task?
– How long do tasks stay put aside for? And is there a correlation between the length of time "on hold" and the pace or nature of work on resuming them?
– Is it useful to give priorities to tasks in the queue? Re-arranging priorities might be just another opportunity for displacement activity; or setting the priorities might be an unwanted cognitive burden. The priorities might be ignored anyway – this could be analyzed automatically from the logs by comparing the priorities assigned on putting tasks into the queue, with the order in which the tasks are actually taken up.
– Might it be useful to move tasks which have not been active for some time, to the back of the queue (effectively giving them a low priority implicitly)?
– Is it useful to group tasks in the queue?

3.4 Information-gathering behaviour

Sidebrain in real everyday use should provide an opportunity to follow information gathering behaviour on significant problems, with a high level of ecological validity. Particular points which could be investigated here include:

– the typical number of investigative steps between observations, and the variations in this
– the typical number of investigative steps between changes to the code
– the typical number of observations between changes
– correlation between these and productivity, in a population of programmers

3.5 Suspendability and resumeability

Programmers often become engrossed in a task, and do not wish to put it down, which appears to be often for fear of losing their train of thought, and not being able to take it up again (although it may sometimes be from enjoyment of the activity). This appearing to be a common cause of strain and overwork, it seemed potentially beneficial to develop tools to make it easier to handle interruptions (such as telephone calls, questions from other programmers, meals, and going home for the night)[14]. Early experience of using Sidebrain suggests that it may go some way to solving this problem.

This differs from most of the preceding areas, in that it covers longer time spans than working memory usually covers.

The main research question in this area is whether the availability of saved task descriptions and observations makes it easier to get back to work after a significant interruption. This could be measured both through mechanical measurements of productivity, and through subjective surveys.

The action of describing the work being suspended, on putting it aside (if it is done then, which implies lazy "`begin task`" usage), might have effects either on the task being suspended (summarising and clarifying the work that has just been done on it), or on the handling of the interruption (either reducing distraction from thoughts of the task it interrrupted, safe in the knowledge that its state has been saved, or increasing distraction from it by having just summarised it).

3.6 Passing work between programmers

The file format in which Sidebrain stores its task data may also be used for mailing tasks between programmers, in, for example, follow-the-sun working[7]. This goes beyond helping the memory of individual programmers, into being a form of communication, and perhaps a part of the corporate memory[15] of the development team or organization.

Research in this area would compare hand-over using this data, against the techniques currently used.

3.7 The value of externalising decisions

Literate Programming[16] is the practice of writing software and its documentation together, in a combined document which interleaves source code and explanation. The emphasis is on the natural-language explanation, with the source code being embedded in it, and typically being a small part of the overall text.

One way of seeing Sidebrain's annotation of activities through the "`begin task`" command is that it is to the activity of programming as literate programming is to the resulting code; what the programmer produces is an interleaving of code changes and explanation of code changes.

It has been observed that one way of improving understanding of a problem is to explain it to someone else, or even to an inanimate object – sometimes termed the "*Rubber Plant effect*" or "*Wooden Indian effect*" (as a decorative plant or figurine can be a convenient object to explain things to[4]). It is thought that how this works is that to express thoughts in language, conflicts in the thoughts have to be either resolved or at least made explicit.

It is not obvious why it should be helpful for programmers to do this through some medium other than the source code before they write the source code – whether, for example, it is the speed of expression, or whether this occurs mostly where there is a poor match between the problem and the technology available to solve it (and, correspondingly, the source code is not concise and clear).

Marking the beginning and end of tasks makes explicit the activities being carried out, which in the informal experience with this software so far has appeared to increase productivity. Experiments using varied styles of explanation may yield some clues as to why this is helpful, and perhaps lead to some helpful further techniques and tools.

For example, use of Sidebrain may be compared experimentally with the *pair programming* of XP[18]. It may also be interesting to compare talking to different targets while programming – pair programming partners, office plants, people who listen but do not say anything subject-specific, people who listen but do not say anything, and perhaps even people who respond but not in a language understood by the programmer – to explore why such verbalizing helps.

[4] The original plant used in this context (as a substitute for a cardboard cut-out of an expert programmer) was given the name "Dijkstra", in 1982 at Cambridge University Computer Laboratory[17]

4 Ongoing work

The next stages of work on this project are

- designing details of experiments, and of the data collection needed for them
- completion of the first version of the software (including recording of enough information for the proposed experiments), and of the documentation for it.

When it is ready, it will be released as open source software, and announced in suitable places on USENET and the web, with an invitation to register to help with some research, by sending back results of running some analysis functions on one's history files.

In parallel with release to the public, subjects will be recruited for case studies, in organisations with which we have research links, and more detailed log data connected and analysed.

Guided by the results from the initial release, both tool and experiments will be refined further for further rounds of experimentation.

5 Summary

The first round of informal requirements for a programmers' working memory support tool have been gathered, and a prototype of the tool almost completed, and internal test use and demonstrations suggest that it has potential to be genuinely useful for real-world use. A number of experiments based around the tool have been suggested, and public deployment of the tool is expected to lead to further iterations.

Appendix: initial survey questions

This simple qualitative survey was conducted internally, as a pilot study, in the author's Department, among academic computer scientists who claimed experience of working on large projects. Although the results were useful, they also made clear the need to rephrase some of these questions. The questions for this initial version of the survey came from the author's own experience of large-scale software development and maintenance.

1. When looking for a piece of information necessary to make a decision about an action take, do you find that your search is typically linear, or typically branching?
2. When looking for information necessary to take an action, do you typically take that as your next action, or do you typically realize that some other action (related or unrelated) is needed, and do that first? Or, if you notice the need for such an action, do you make a note to go back and do it later? And if so, how do you make the note (paper? file of such notes? add a comment to the code?)
3. If, for question 2, you find yourself doing some other action first, do you sometimes find it hard to remember what you were originally trying to do, and perhaps even lose track of the original action?

4. Do you use any systematic strategy for keeping track of what you are trying to do? If so, what is that strategy?
5. What kinds of information do you gather, and how do you store them?
6. If your answer to Q4 was “no”, is this because of the lack of a the specific tool or system for it, or would you not use such a thing anyway? whether your answer to Q4 was “yes” or “no”, do you regard the extra typing as a problem, and would you expect to find voice input useful for this?
7. If voice input were available in a suitable form, would you expect it to make storing such information significantly easier?

Acknowledgements

This research has been supported by the Science Foundation Ireland Investigator Programme, B4-STEP (Building a Bi-Directional Bridge Between Software ThEory and Practice).

The author wishes to thank colleagues for feedback on drafts of this work.

References

1. Letovsky, S.I.: Cognitive processes in programmer comprehension. In: Empirical Studies of Programmers, Ablex Publishing Corp. (1986) 58–79
2. Miller, G.A.: The magical number seven, plus or minus two. The Psychological Review **63** (1956) 81–97
3. Stallman, R.M.: GNU Emacs Manual. GNU Press (2002)
4. W3C: Xml. (Web address http://www.w3.org/XML/)
5. Humphrey, W.S.: Introduction to the Personal Software Process. Addison-Wesley (1996)
6. Russo, J., Johnson, E., Stephens, D.: The validity of verbal protocols. Memory and Cognition (1989)
7. Espinosa, J.A., Carmel, E.: Modelling coordination costs due to time separation in global software teams. In: International Conference on Software Engineering. (2003)
8. Wohlin, C., Höst, M., Henningsson, K.: Empirical research methods in software engineering. In: Empirical methods and Studies in Software Engineering – Experiences from ESERNET. (2003) 7–23
9. Kitchenham, B., Pickard, L., Pfleeger, S.L.: Case studies for method and tool evaluation. IEEE Software (1995) 52–62
10. Bates, M.J.: The design of browsing and berrypicking techniques for the online search interface. Online Review **13** (1989) 407–431
11. Potts, C., Catledge, L.: Collaborative conceptual design: a large software project case study. Computer Supported Cooperative Work **5** (1996) 415–445 Section 6.3, Process obsession as a displacement activity.
12. Ford, N., Ford, R.: Towards a cognitive theory of information accessing: an empirical study. Inf. Process. Manage. **29** (1993) 569–585
13. Mancy, R., Reid, N.: Aspects of cognitive style and programming. In: Annual Workshop of the Psychology of Programming Interest Group. Number 16, PPIG (2004) 1–9
14. Stewart, R.: Managers and their jobs. Macmillan (1967)
15. Garvin, D.A.: Building a learning organization. Harvard Business Review on Knowledge Management (1998) 47–80

16. Knuth, D.E.: Literate Programming. CSLI, Stanford (1992)
17. Wray, S.: Rubber plant effect. personal communication (2005)
18. Williams, L., Kessler, R.R., Cunningham, W., Jeffries, R.: Strengthening the case for pair programming. IEEE Softw. **17** (2000) 19–25