In P. Romero, J. Good, E. Acosta Chaparro & S. Bryant (Eds). Proc. PPIG 18 Pages 83 - 96

Abstraction levels in editing programs

John C G Sturdy

University of Limerick
john.sturdy@ul.ie

1 Introduction

Traditionally, text editors for program text have operated largely at the level of lines and characters, and their users have made their changes largely character by character, with "cut and paste" and "find and replace" as their only more powerful actions.

Some editor users may think of changes to source code in those terms, too, but it is also possible that some, perhaps the more experienced programmers, think at a higher level of abstraction, such as syntax trees, and would be able to work more efficiently with tools that match the way they think.

Some such tools have now begun to appear, typically described as "refactoring tools"; some of them are integrated into editors and Integrated Development Environments (IDEs) such as Eclipse, NetBeans, IntelliJ, and Visual Studio; and some are separate command-line programs.

It may also be the case that the provision of such tools will help to train or educate programmers to think of their work in a more abstract way; this effect may appear naturally as experienced programmers go about their work, or could be done deliberately in the training of novice programmers.

This research sets out to explore some editing tools that work at higher levels of abstraction than lines and characters, investigating first their design, and then moving on to empirical research to find their usefulness and effects.

2 Survey

Card, Moran and Newell[1] compare navigation using traditional cursor keys, keys that produce movements in terms of the text units (paragraph, line, word), joystick and mouse. They found the mouse was fastest, followed by joystick, then character cursor keys, with the text movement keys being the slowest. Their experiment was for some word processing tasks; one of the tools presented in the present research is equivalent to the one they found slowest, but in the context of editing source code, which, being more strongly structured, may give more of an advantage to movements based on the text contents. Also, it is possible that using the text contents to guide the movements and editing operations may reduce the number of small corrections at the end of a sequence of keystrokes. Despite the advantages claimed for the mouse by usability experts and by novice users, many experienced computer users, however, largely prefer the keyboard over the mouse

when both are available for the same operation, apparently for a combination of reasons including convenience (avoiding moving the hands from one input device to another) and comfort (many people, particularly RSI sufferers, find the mouse less comfortable to use than the keyboard).

Questions in this area addressed by this research include:

- Whether these findings apply in the context of program editing
- Why the keyboard input is preferred over the mouse despite the claims of "experts"
- Why the text-based navigation is slower, and whether this is true under all circumstances

There have been several attempts to introduce "Syntax editors" or "Structure editors", working with specific programming languages, and editing the parse tree directly, and redisplaying it to the user after each change[2][3][4][5]. These never became popular; users typically found them unnatural, constraining them to do things in a certain sequence, not necessarily the one they would normally have done. However, this does not rule out the possibility of using information about the the language to make a productive interpretation of the user's command input.

A feature generally known as "syntax colouring"[6][7] is now widely made available in programmers' text editors. This displays programming language keywords, comments, string liberals, and other distinct parts of a program in distinct ways, typically including coloured text, coloured backgrounds, and underlining. The popularity of these suggests that this information is significant to programmers, at least for program comprehension. We hope to apply similar information to interpret user input to the editor optimally.

More recently, the concept of "Refactoring" (re-arranging source code without changing its meaning) has been introduced[8]. Refactoring tools are typically language-specific, although much of the logic of the refactoring operations is the same across a wide range of languages.

Independently from explicit refactoring operations, some points about program editing that have long stood out to the author (as a programmer with over a decade of commercial experience in several programming languages) include the following:

- Many sequences of navigation commands are aimed to reach a few specific kinds of points of interest in the program text.
- There are many routine program editing operations normally carried out by stereotyped sequences of human actions, and which are therefore prime candidates for automation, a typical example being to take an expression, replace it with a variable name, and at the appropriate place insert a definition of that variable, pasting in the original expression as its initial value, and then adding a further use of that variable. Some of these sequences are those recognized as refactoring transformations; others are not refactoring, as they change the meaning of the program, for example, wrapping a conditional around a block of code. They are, nonetheless, very predictable.

Both of these are very stereotyped forms of behaviour, and, over years of observation, some particular patterns of editing behaviour stood out as being candidates for automation. Interestingly, once some of these had been automated and the new commands put into regular use, further stereotyped patterns became apparent (perhaps indicating a shift in thinking about editing).

3 Design of the experimental system

3.1 Requirements and principles

To avoid the problems that prevented the general uptake of syntax editors, the system must integrate well with conventional editing, allowing a mixture of the new operations and traditional ones.

In operating on language-based units of selection, it must use a consistent model of navigation, selection, and action. It must give feedback on the type of selection in use.

There should be an element of 'DWIM" ("Do What I Mean") where necessary, but without sacrificing the consistency mentioned above.

The system should make it easy to achieve the intended change with a minimum of command input, and this ease of use should be available through a variety of input methods and devices, including those often chosen because of disabilities – for example, voice input. These may become used more widely in the future.

As well as minimizing the amount of input, the design should minimize the rate of making mistakes (both those which are detected and corrected immediately, such as a deletion or insertion that is undone immediately, and the more subtle ones which show up later, for example when compiling or running the program). In particular:

- Error-prone operations where the desired result is easily predictable are prime candidates for automation. Typical example are wrapping a control construct around some statements, or removing an existing such constructed; misplacing the end of the control construct – such as selecting which closing brace in the C language – seems to be a common kind of slip in altering existing source code.
- It should be clear what the effect of command will be; the highlighting of the selection (see below) helps with this.

Although it is possible for such a system to use nonstandard interface hardware (such as voice input, or specially adapted keyboards to compensate for disabilities), it should also be usable through standard hardware, such as a keyboard.

3.2 Design decisions

We have implemented the experimental system as a collection of extra commands for the popular programmable editor, GNUemacs[9]. Care was taken to avoid

interference with the normal operation of the editor, so the extra commands are available as an enhancement to, rather than a replacement for, the normal facilities.

Central to the overall design is the extension of the idea of the "cursor", from pointing to a single character in the text being edited, to pointing at a syntactically or semantically significant unit or units of text. To emphasise this, the selected text is drawn highlighted with a different background colour, the colour used also providing feedback on what the current type of unit is. The cursor keys move the selection highlight in steps of the current type of unit; for example, if the current unit is `statement`, a whole statement (or several whole statements, if the selection has been extended) will be highlighted, and the cursor keys will move by whole statements.

The selection cursor can be extended to cover multiple selection units, by using the arrow keys with a modifier key (much like using the Shift key with the arrow keys on some common word-processing software).

The selection cursor is not necessarily contiguous; for example, navigating by depth of brackets leaves the selection cursor split into two pieces, one covering an opening bracket, and the other covering the corresponding closing bracket.

Editing commands, such as Convert to Function, or Make Conditional, or Delete, act on the current selection.

Choosing different types of unit to navigate by is done by using the cursor keys while holding down a modifier key, as explained in more detail below.

The new high-level editing commands use key sequences starting with some of the existing editing keys (such as Insert and Delete), and also uses some of the editing keys with modifiers.

3.3 Navigation

The long-established on-screen representation for files of program text is as a series of lines of characters. In this, each line is made of a series of characters, and it is possible to move from one line to the next either directly, or by moving (with character-by-character movements) off the end of the previous line. This establishes the idea of major and minor dimensions, and the relationship between them, the major dimension being the line number, and the minor dimension being the character number within the line. Modern computer keyboards typically provide two pairs of "cursor keys" (the left and right arrow keys for the minor dimension, and the up and down arrow keys for the major dimension), for moving in these two dimensions.

We generalise this model to allow flexible selection of a pair of dimensions, major and minor, from a variety of possibilities, some of them already built into our host editor, GNUemacs, and some of them added for this research. The built-in movement commands include movement:

- by characters, lines, and pages
- by words, sentences, and paragraphs

– by bracketed expressions[1], depth of brackets, and by function definitions

The added movement commands include movement:

– by statements and parts of statements (head, body, tail, framework[2])
– by table cells and rows in markup languages

Introspective consideration of the way movement commands are used showed that they are typically used in groups, as listed above: the user will navigate for a while by cartesian movements, or for a while by movements based on natural language structure, or for a while by those based on program structure, but will switch between these groups relatively rarely. Indeed, almost all movements within a particular type of file may well be from the same group or a limited range of groups; for example, editing a C program will rarely require movement by words, sentences and paragraphs (apart, perhaps, from when moving inside embedded string literals and constants).

Within a group, the finest scale of movements are rarely mixed directly with the coarsest ones: for example, the user may move by pages (screenfuls) to find the right part of a file, then by lines onto the right line, and then by characters to the exact place for editing[3]; but they would rarely go directly from page movements to character movements.

This leads to the idea of two kinds of changes between different forms of movement around the file:

– Switching between different kind of movement (cartesian, text, program structure);
– Zooming between different scales of the same kind of movement.

Combining these two gives us a two-level tree of kinds of movement; alternatively, this could be seen as a two-dimensional grid, as follows:

Cartesian:			pages	lines	chars
Structural:		defuns	depth	exprs	chars
Text:	paragraphs	sentences	phrases	words	chars
Structured text:		block-depth	blocks	words	chars
Tables:			rows	cells	chars
Program:	defuns	statements	statement-parts	exprs	chars

At any time, we have a pair of major and minor dimensions, which are adjacent in one row of our grid of dimensions, for example “sentences” and “phrases” in the “structured-text” row. The vertical cursor keys are taken over from the normal GNUemacs key assignments to perform our major movements, and the

[1] The implementation of movement over bracketed expressions was extended to cover the paired tags used as bracketing constructs in markup languages.

[2] For example, the head of an `if` statement is the condition, the body is the conditional code, and the tail is the `else` case if there is one; the framework is the keyword `if`, along with any brackets or further keywords required, such as `else`.

[3] unless the page-level or line-level movements happen to hit exactly the right point, which will not be possible for many points of interest within the code

horizontal cursor keys for the minor movements. When we are using cartesian style movements, these keys perform very nearly their traditional actions.

Since the available movement dimensions are conveniently described on a two-dimensional grid, choosing a pair of dimensions can be achieved conveniently by using the cursor keys (with a modifier). While this is happening, the grid of available dimensions is displayed as in the table above, with the selected minor dimension highlighted. This is redrawn after each operation that chooses a dimension, and removed at the start of any command. Thus, it remains visible while the user is navigating the space of possible dimensions, but it disappears when the user starts to use any other commands.

To provide feedback both on the current selection and on the form of navigation and selection commands in use, a cursor covering the current selection is drawn, by changing the background colour of the selected text. The normal system cursor usually appears at the start of the selected text. The extended cursor changes colour (as does the normal cursor) to indicate the current dimension. It is redrawn at the end of each of our movement or editing commands, and is removed whenever the user sends any input to GNUemacs, such as a keystroke. Thus, the display reverts to the normal GNUemacs display whenever the user does anything outside our additional command set.

Along with the enlarged selection cursor, the colour changes give the user feedback on the selected form of movement, while maintaining feedback on what text is currently selected. Specific confirmation is given in the status line that the editor continuously displays.

The system can remember a different current dimension for each of GNUemacs' "major modes", which are configurations of editing commands for particular types of text, such as c-mode, html-mode, lisp-mode etc.

Since source code often has pieces of natural language text embedded in it, in the forms of string literals and comments, a separate set of movement dimensions is remembered for such embedded text, and the system switches automatically between them.

To investigate why navigation using semantically-based units was found to be slower (by Card, Moran and Newell[1]), a hybrid form of navigation is included, in which the arrow keys move "normally", that is, by single characters and lines, but the selection cursor is drawn as above, for the current selection unit that the cursor is within. This form of navigation comes from the author's introspective observation that it is hard to predict what will be selected after holding an auto-repeating cursor key down; the progress of the selection is not steady in terms of position on the screen, when the cursor keys are moving by semantically-based units.

3.4 Stereotyped operations

Many sequences of editing operations by programmers follow some well-defined patterns, aiming for certain common goals such as reusing a value by putting it into a variable, or reusing a block of code by turning it into a named procedure, or making a group of statements be executed only under a certain condition.

Such operations typically involve syntactically or semantically significant areas of the text, such as those selected by the navigation commands described above.

One of the questions to be investigated in this research is what programmers are thinking of while doing such stereotyped operations. It could be, for example, that some programmers will think of an operation as "Make these statements conditional on the value of *a*", while others will think of the same operation as "Put an `If (`*a*`)` and an open brace *here*, and put a close brace *there*".

Those of these commands which do not change the meaning of the program come under the description "refactoring", and are available in a variety of editors and IDEs.

Obviously, choosing a command to do such an operation requires knowing that such a command is available. This implies either having learnt of the command, or guessing that a particular operation might be implemented as a single command, and searching for it. It may be possible for the editor to recognize a sequence of manual edits as performing something that it also implements as a command, and to use this as a pedagogical vector by informing the user that this could have been done more easily. (However, recognizing such a sequence would not be easy.)

To investigate how editor users think of editing operations, we made available a collection of commands to do such operations atomically, and will observe how much they are used compared to more laborious alternatives.

The operations provided include the following:

Convert selection to variable A common form of behaviour in editing existing code (and also sometimes in creating code on the fly) is to arrange for the re-use of the result of calculation, by assigning the result to a newly created variable, using the variable where the original calculation was, and then using the variable again elsewhere. This is a very stereotyped form of behaviour, with just two decisions to be made (or just two parameters to be given to a macro command) once the calculation concerned has been selected:

- the name of the variable
- the scope for variable is to have

The type of the variable (for programming languages with static type systems) can often be deduced from little more than a trivial syntactic examination of the surrounding text, so this does not need to be specified by the user. A good candidate for the scope can also be found automatically; for a start, the scope cannot be wider than that of any variables referred to in the calculation. It is, of course, appropriate to ask the user for a name for the variable, although, in fact, any unique name would suffice. (It may, later, be possible to suggest a suitable name for the variable by automatic determination of its rôle[10].)

Not only is this a very predictable pattern of behaviour when done manually, it is also a somewhat tedious one, involving moving the editor cursor away from your point of real interest in the code, and then coming back to it.

This action is a form of refactoring; it does not alter the semantics of the program as a whole.

Convert selection to global variable This is similar to creating a local variable, but finding a place for the variable declaration is different.

Convert selection to function This is similar to using an expression to create a variable, but has the added interest of creating a parameter list, containing all the free variables in that piece of code. As with creating a variable, making a command for this lets the user keep their attention on the point of use, rather than having to move away to the point of declaration, which is likely to be less significant in their understanding of the program.

The new command for this will refuse to convert a selection which does not make sense as the body of a function; for example, if the level of bracket nesting is different at the start and end of the text, or if inside the text the level of nesting drops below that at the start and end of it; or if the selection begins or ends part way through a statement in the programming language concerned.

This action is a form of refactoring; it does not alter the semantics of the program as a whole.

Alter selected text to another value of the same kind The "begin altering" command temporarily redefines the cursor keys to bring different possible values into the selection. The range of possible values is displayed in the same way as the range of possible navigation dimensions. The minor dimension keys (left and right arrows) select different values of the same kind, and the major dimension keys switch between different kinds of value. For example, if the selection was originally a reference to a local variable, the left and right arrows will choose from all the local variables in scope at that point; the up and down arrows will move to different kinds of value, such as global variables and defined constants. The "end altering" command accepts the current value, and returns the arrow keys to navigating the text.

Surround selection with call

Remove function call around selection These are simpler operations than those above, but doing them manually still involves splitting the user's attention between two points in the program.

Unify selected statements

Comment-out selection These operations are not particularly fiddly, but they benefit from acting atomically on a visible selection, rather than the user having to remember the other end of the selection while working on one end of it.

Make selection conditional

Make selection iterative

Remove control construct around selection These operations wrap the selected code with a control construct, or remove such a control construct. They have an advantage over entering or deleting the start and end of control construct manually, as they act atomically on a visible selection, thus making it easier to be sure that the closing part of the construct (such as a closing brace, or the keyword `end`) is being inserted at the right place.

The command to make the selection conditional examines the selection to see whether it is already the body of a conditional statement without an `else`

case; in this case, it adds another term to the existing conditional statement. This involves some fairly shallow analysis of the selected text (deeper than a regular expression match, but not full parsing, which may not even be possible, as the code being edited might not currently be syntactically well-formed).
As with the previous two operations, these benefit from acting atomically on a visible selection. They are more complex patterns of behaviour than the previous two, while still involving a very set pattern of inserting text.

Programmers often do some kind of tidying up after an edit, such as adjusting indentation or other whitespace, and this group of operations typically end with reindenting the affected area; our new commands do this automatically.

3.5 Language-independent definitions

Many of the operations described above are similar across a range of programming languages. The operations are implemented in two layers:

Language-specific support For each language supported, there is a collection of functions for handling code fragments in that language; a typical such function is one to return the names (and types, if the language is statically typed) of all the local variables in scope at a particular point.

Language-independent logic The editing operations themselves are written to work in any language, using the language-specific support. For example, the operation of converting a region of code to a function is shared between languages, calling language-specific functions to extract variable references, variables in scope, to insert the syntax needed for a function definition and for a function call, but using language-independent code to compare the variable references and variable bindings to find which variables must be passed in to the new function as parameters..

Although in practice this is unlikely to be possible to a significant extent, it should in principle be possible to make equivalent changes to equivalent programs written in different languages, using the same commands.

4 Initial observations

The author has made considerable use of the system and some predecessors from which it grew, finding it usable and useful although frequently mixing in traditional editing commands – gradually less so as these atavastic regressions systematically revealed what remained to be done to achieve the aim of accurate edits with a minimum of "fiddling around".

Further possible high-level commands became apparent once the system was in use; this suggests that using the system changes the user's way of thinking

about editing, and so may support the hypothesis that such a system could be used to train programmers to think of their editing more abstractly[4].

The system has shown itself useful for exploration of code structure as well as for editing – in fact, perhaps more so, as the easy highlighting of structure elements appears to the ease the comprehension of complex program sections.

Although, in description, it sounds complicated, the system of major and minor dimensions, with re-use of the same keys but with modifiers to move around a grid of dimensions, has proved practical and natural, with fairly quick user adaptation. This must be seen in the context of the system being designed for experienced full-time programmers, and not for casual users.

The author, having a keyboard-related disability, uses the system regularly through nonstandard hardware, including pedals and voice input, and finds it is natural to use through a mixture of these, with conventional keyboard input where necessary. A blind programmer, working with speech output only, is investigating the usefulness of the system in that context; there, it is concise output (as well as not having to adjust the exact position) that is more important than concise input.

5 Research questions

This project is designed to investigate various questions about how people edit programs. Some of this can be supported by automatic data gathering built into the tools, and some of it by observation, questionnaire and interview.

- Given a rich choice of navigation and editing commands closer to programming language structure than characters and lines, which such commands do programmers use, and which do they ignore? For navigation, is this influenced by the behaviour of auto-repeat keys? Which commands do users most often undo?
- How do variations between programmers in the choice of editing commands correlate with other characteristics of the programmers, such as experience and background?
- When, and why, do users revert to more laborious means of editing when the editor can do much of that work for them with far fewer commands?
- What can be deduced from this data, about the cognitive processes involved in editing software?
- Does the use of syntax directed commands reduce the time taken, the number of keystrokes, or the number of exploratory and retracted (undone) actions?
- What are the implications and effects of editing (and of thinking about editing) at a higher level of abstraction?

6 Experiment design

Having developed and experimented with the system as a programmer, the author then moved onto the next stage, of empirical verification. Sometime before

[4] Possibly a weak Sapir-Whorf effect?

starting to recruit participants, the author started to participate in potentially relevant online communities. Suitable logging software was written part-way through the project, and left in habitual use to establish its stability.

The experiment concerns the use of a system in ordinary everyday work, and so has to be designed as a field study, using volunteers. Two approaches were available to the author, who decided to take both of them, for completeness.

Broad recruitment from online communities Natural places to look for people interested in experiments on editors included newsgroups and mailing lists about editors, such as comp.emacs.

Targeted recruitment from specific contacts The author's University has contacts with companies in the area, who are sometimes interested in participating in experiments.

Both of these are in progress at the time of writing.

Three forms of experimental data could be gathered:

- Watching and recording subjects as they work.
- Subjective feedback from participants, gained through interview or questionnaire.
  A slightly less subjective form of feedback to be gained from interview or questionnaire is which features the users miss when going back to a GNUemacs without these extensions.
- Log files from monitoring software offered to participants.

Information available directly from participants includes perceived ease of use and usefulness, and observations on whether it seems to increase productivity, or reduce frustration.

Information available from log files includes levels and patterns of usage of the new commands, and how much the user reverts to commands outside the new system.

More specifically, the following can be logged:

- the amount that the commands inside and outside this system are used
- for each editing action, whether it is within this system, and whether the navigation commands leading up to it are within the system, outside it, or a mixture with the start or end outside the system
- the amount that the commands outside this system are used while the system is making them available directly on keys

Users will be asked about the following aspects of their computing experience:

- Length of experience with programming
- Length of experience of GNUemacs
- Length of experience of with the experimental system
- Programming languages used in the monitored period
- Programming languages known
- Programming languages preferred

– Experience of writing tools manipulating programs as data, e.g. compilers and interpreters
– Level and subject of highest qualification

To encourage participation, the results have to be sent back by the participants in the form of e-mail messages, so that they can see everything that has been gathered about their editing.

7 Analysis

Analysis of the results will start with informal manual inspection of log files for any obvious unusual patterns. Specific statistical tests will be done on the proportions of commands used of each type, and correlation of these with time using the system, and with factors from questionnaires about experience and training.

Users of different editors and IDEs probably have different profiles within the overall population of programmers; GNUemacs has a reputation for a steep learning curve, and hence for being largely an experts' tool. This must be taken into account in the analysis, as it may be that GNUemacs users are preselected for their ability to learn additional tools and techniques.

8 Possible results: the hypotheses

The author's main hypothesis is that the way programmers think about changes to programs varies considerably from one programmer to another, and this may correlate with some surveyable characteristics of the programmers will. In particular, those with considerable experience of non-imperative programming languages, or of compiler writing, or with mathematical background, may be more likely to make good use of editing operations at a more abstract level than the traditional ones.

The author expects that use of such editing tools will influence the way their users think of editing code.

A minor hypothesis is that the slowness of using semantically-based navigation, as observed by Card, Moran and Newell[1], is connected to the difficulty in predicting how long to hold down a repeating key to get to a point already identified by eye on the screen.

9 Possible further work

It may be possible to use editing tools such as the ones described here as a training aid to help programmers to think of their programs at a higher level of abstraction then they did before.

The change in the nature of the command stream, making it closer to a declaration of the high-level intention of the programmer, along with the information

that is gathered to tell what operations are appropriate at any time (thus allowing a higher-level description of the state of the system), may be a step towards conversational computing, using speech both for input and for output. The same characteristics may also prove desirable for editing on devices without space for a conventional keyboard and large screen display, such as many handheld devices.

It may be possible to recognise automatically repeated patterns of behaviour by the user, and to suggest making them into commands.

The higher-level navigation and editing commands may make possible new forms of input for controlling editors; in particular, the smaller number of commands needed may make it easier to use voice commands, and the reduced need to type text may make it possible to edit using non-keyboard input devices, such as joysticks. Both of these may be useful in "ubiquitous computing".

In turn, in conjunction with developments in softawre comprehension tools, such as Chive[11], these may lead away from the dependence on a two-dimensional, "page"-like presentation of source code, in turn leading to further changes in how source code is understood and handled.

It would be interesting to investigate the effects of introducing novice programmers to these editing tools, perhaps in their first experience of editing source code.

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).

Software availability

The experimental software is published under the General Public Licence, and is available for download from `http://emacs-versor.sourceforge.net/`

References

1. Card, S.K., Moran, T.P., Newell, A.: The Psychology of Human-Computer Interaction. Lawrence Erlbaum Associates (1983)
2. Allison, L.: Syntax directed program editing. Software Practice and Experience (1983)
3. Zelkowits, M.V.: A small contribution to editing with a syntax directed editor. SIGPLAN Not. **19** (1984) 1–6
4. Garlan, D.B., Miller, P.L.: Gnome: An introductory programming environment based on a family of structure editors. In: SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments, New York, NY, USA, ACM Press (1984) 65–72
5. Kaiser, G.E., Feiler, P.H., Jalili, F., Schlichter, J.H.: A retrospective on dose: an interpretive approach to structure editor generation. Softw. Pract. Exper. **18** (1988) 733–748

6. Baecker, R.M., Marcus, A.: Human factors and typography for more readable programs. ACM Press, New York, NY, USA (1989)
7. Spinellis, D.: Dear editor. IEEE Software **22** (2005) 14–15
8. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
9. Stallman, R.M.: GNU Emacs Manual. GNU Press (2002)
10. Sajaniemi J., N.P.R.: Rôles of variables in experts' programming knowledge. In Romero, P., Good, J., Bryant, S., Chaparro, E.A., eds.: Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group. (2005) 145–159
11. Cleary, B.; Exton, C.: Chive - a program source visualisation framework. In: 12th IEEE International Workshop on Program Comprehension, 2004. Proceedings. (2004) 268–269