Structure Editors: Old Hat or Future Vision?

Andreas Gomolka and Bernhard Humm

Hochschule Darmstadt – University of Applied Sciences
Haardtring 100, 64295 Darmstadt, Germany
andreas.gomolka@gmail.com, bernhard.humm@h-da.de

Abstract. Structure editors emphasise a natural representation of the underlying tree structure of a program, often using a clearly identifiable 1-to-1 mapping between syntax tree elements and on-screen artefacts. This paper presents layout and behaviour principles for structure editors and a new structure editor for Lisp. The evaluation of the editor's usability reveals an interesting mismatch. Whereas by far most participants of a questionnaire intuitively favour the structure editor to the textual editor, objective improvements are measurable, yet not significant.

Keywords: Programming, Structure editor, Evaluation, Lisp, Eclipse.

1 Introduction

Structure editors have fascinated designers of development environments for decades [1, 2, 3, 4]. The idea is simple and convincing. The elements of the syntax tree of a program are mapped to on-screen artefacts and can be edited directly.

The basis for this is the awareness that programs are more than just text [5]. A programmer designing a piece of code thinks in structures: classes, methods, blocks, loops, conditions, etc. Using a textual program editor he or she has to codify those syntactically using parentheses such as '{...}', '(...)', '[...]' or using keywords such as 'begin ... end'. The compiler then parses the syntactic elements and re-creates the structures in the form of an abstract or concrete syntax tree – the same structures which the programmer originally had in mind. This just seems inefficient and not intuitive.

Structure editors fill this gap: What the programmer thinks is what he or she sees in the editor. Surprisingly enough, structure editors, although around for decades, have never become mainstream. So somehow, there has to be a catch in this quite simple and straight forward idea. In this paper, we try to find out whether it may be possible to avoid the drawbacks of former implementations and whether structure editors are maybe more than an "old hat" – perhaps even a "future vision" of programming environments?

To be able to answer this question, we analyse the requirements of a usable structure editor and describe layout and behaviour principles for structure editors. Based on this, we present a new structure editor for Lisp and an evaluation of the editor's usability based on a questionnaire – with interesting results.

L.A. Maciaszek and K. Zhang (Eds.): ENASE 2011, CCIS 275, pp. 82–97, 2013.

The remainder of the paper is structured as follows: Section 2 describes layout and behaviour principles for structure editors. In Section 3, we present a new structure editor for Lisp via samples and screenshots and give some insights into its implementation. Section 4 describes how we evaluated the usability of the editor and in Section 5 we position our work in relation to other approaches. Section 6 concludes the paper with a critical discussion.

2 Layout and Behaviour Principles

A structure editor should improve the readability and comprehensibility of the code whilst not compromising useful features of textual editors. To this end, we postulate the following layout and behaviour principles for structure editors:

1. Focus on the Net Code. The code layout should support the programmer in focussing on the net program code, i.e., keywords, identifiers, and literals. The structure of the code should be visualized in a clear but discrete manner. A look into the related literature reveals that there is no overall agreement, which kind of representation fits this intention. Dimitriev, for example, states that programmers always translate program text to tree structures in their mind [6] and argues that editors should emphasise this view. In contrast, Edwards claims that tree structures are not satisfying to display conditionals and therefore proposes to visualize programs using tables [7]. We think that the representation should emphasise the structure of the program, but also enable the programmer to recognise the original code. Therefore, we propose, similar to the approach of Ko and Myers [4], to replace syntactic elements for structuring the code (e.g., parentheses for block structures, separators like semicolons, and delimiters like double quotes for string literals) by graphical elements.

2. Do not Restrain the Programmer. The editor should help, but not unnecessarily restrain the programmer. For an editor it is only possible to visualize the structure of the program correctly if it does not contain any syntactical errors. Some former approaches handled this problem by preventing the creation of syntactical errors at all [1, 2]. This had the effect that simple operations which change the structure of the program became quite complex, e.g., removing a parenthesis and inserting it somewhere else. It is essential for the usability of a structure editor how it handles this problem.

3. Keep the Layout Compact. Apart from editing, a programmer uses an editor also for reading and understanding a piece of code. The structured representation should support the programmer in quickly getting an overview of the whole program. Therefore, the structured representation should be as compact as the plain text representation.

4. Keep Common Look and Feel. The behaviour of the structure editor should be as similar as possible to the look and feel of widely used editors. Examples are shortcuts, colouring, and behaviour during typing. This facilitates getting accustomed with it for experienced programmers.

5. Do not Introduce New Dependencies. A structure editor is just one of many more tools to work with a program. The textual form of a program makes it easy to change between different editors. This independence should not be dismissed without a good reason. Thus, a structure editor should not necessitate changes to the programming language or the way programs are stored.

6. Make the Layout Configurable. Where possible, the programmer should be able to configure the presentation of the code. For example, colours that are used in the layout should be configurable.

7. Leave the Choice to the Programmer. Some programming tasks might be easier to achieve with a simple structure editor, some with an advanced structure editor and yet others with a textual editor. Therefore, the programmer should be able to freely and easily swap between different editors respectively editor modes.

3 A Structure Editor for Lisp

This section presents a new structure editor for the programming language *Lisp* that was developed as a research prototype. It follows the principles we proposed above.

3.1 Why Lisp?

The main reason why we decided to build the research prototype for Lisp – or, to be more precise, *Common Lisp* – is Lisp's uniform syntax. Lisp data is expressed as a so called *S-expressions* [8]. The term S-expression means *symbolic expression* and includes symbols and nested lists. As there is no syntactical difference between data and code, a Lisp program also consists of S-expressions. This simplicity and uniformity and the ability to treat Lisp code as data make it particularly easy to develop a structure editor for Lisp.

Also, in a different research context, we use Lisp as a base language for developing *domain-specific languages* (DSLs) in the context of *language-oriented programming* [9]. A structure editor may be particularly useful for developing programs using DSLs that are based on Lisp.

3.2 Code Presentation

The structure editor is based on the Eclipse plug-in *CUSP* [10]. CUSP already provides an environment for developing Lisp programs using Eclipse including a *Navigator View* for browsing Lisp projects, a *REPL* (*Read-Eval-Print-Loop*) and an *Outline* of the currently displayed Lisp file. The new structure editor has been integrated into this environment as an additional *Editor Window* (see Fig. 1).

The Editor Window consists of two separate representations of the code. Besides the structured representation, we also provide a textual one. The user is able to switch between these two using the tabs at the lower left corner of the editor window.

The following Figures 2-4 demonstrate the different possibilities of viewing the code that are provided. All three figures show the same snippet of Lisp code defining a new function called "hello-world" which prints a string n times.

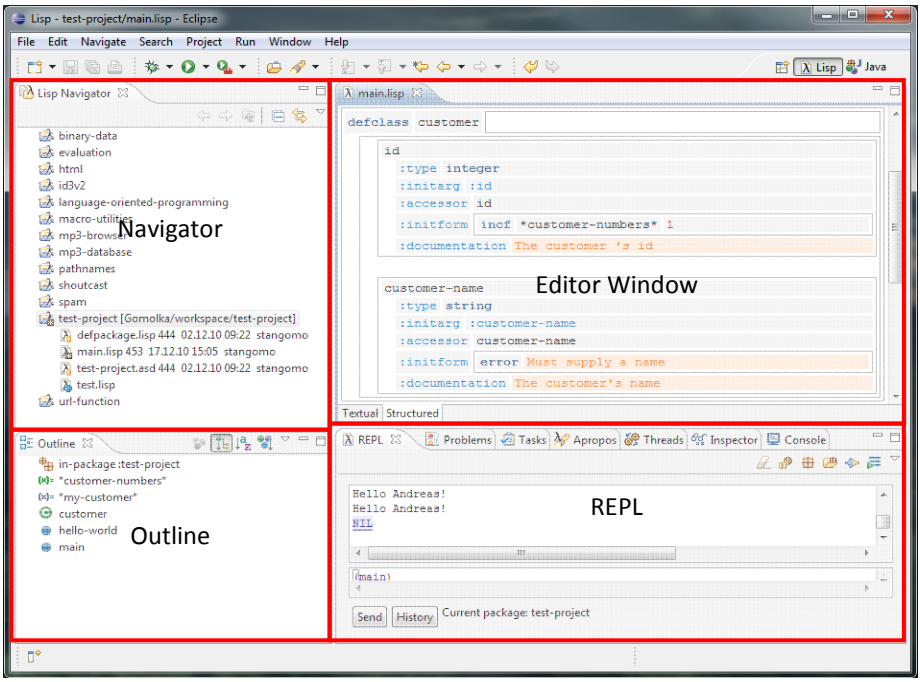

Fig. 1. Overview of the structure editor GUI

```
;;; This is a comment
(defun hello-world (name frequency)
  (dotimes (i frequency)
    (format t "Hello ~a!~%" name)))
```

Fig. 2. Textual representation

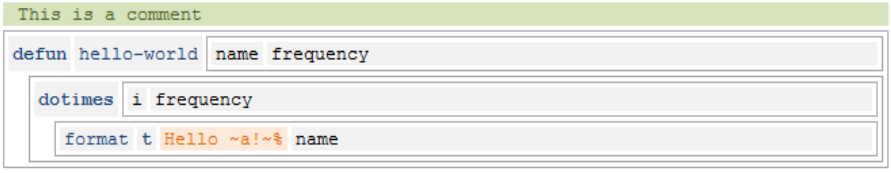

Fig. 3. Default structured representation

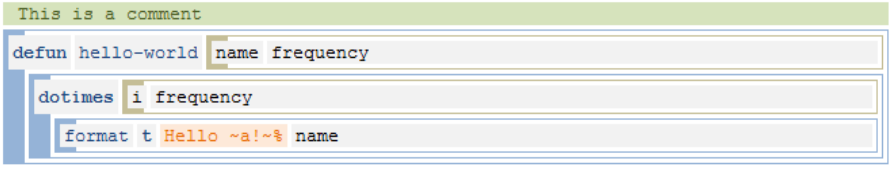

Fig. 4. Coloured structured representation

Fig. 2 shows the snippet using the textual representation. The structure of the code is visualized by the indention of the lines and the individual symbol types (e.g., keywords, string literals, comments, etc.) are indicated by different colours.

Fig. 3 shows the same snippet displayed in the structure editor. All parentheses are replaced by grey boxes which visualize the block structure. Also, the double quotes delimiting the string literals are hidden and expressed by the light orange background. Similarly, the leading semicolons marking the beginning of a comment are hidden and the comment is indicated by the light green background. All this removes syntactic delimiters from the code and accentuates the net code, which satisfies Principle 2 in Section 2.

A slightly different representation of the same code snippet is shown in Fig. 4. There, in addition, coloured bars are displayed at the left side of each box and the boxes themselves are also coloured. The colours indicate whether a block contains a call to a function or macro (e.g., “defun”) or just an ordinary list (e.g., the parameter list of the function “hello-world”).

According to Principle 70, the programmer may decide which representation to use and enable or disable the additional information expressed by those colours via the preferences menu. Furthermore, all colours to be used (background and foreground) can be configured (Principle 6).

3.3 Editing

The code can be edited directly in the nested block structure. There are no additional commands or shortcuts necessary compared to editing the code in the textual representation. As shown in Fig. 5, typing an opening parenthesis will open a new box. Typing a closing parenthesis will close the current box and move the caret outside. In each step, the layout rearranges itself according to the changes. This satisfies our claim to enable the programmer doing the same typing as using a textual editor (Principle 4).

The caret can be moved around using the mouse or the keyboard. The arrow keys will move it one character to the left or right or one line up or down. Using <Tab> respectively <Shift><Tab>, it is moved one field forward backward. <Pos1> will place the caret at the beginning of the first field of the current line and <End> at the end of the last field.

The programmer may decide about line breaks or blank lines. They will be inserted by typing <Return>. Each new line is inserted to the current block. Line indention is calculated automatically depending on the context of the current block, because this is part of the block structure.

The structure editor provides code completion which also shows additional information about the selected symbol as shown in Fig. 6. As common in Eclipse, this is invoked using <Ctrl><Space>.

Common actions like undo/redo or cut, copy and paste may be called via the “Edit” menu or by using the usual shortcuts, for example <Ctrl><C> for “copy” (Principle 4).

1. Empty row:
2. Inserting "(":
3. Inserting the string "some":
4. Inserting <Space>:
5. Inserting the string "code":
6. Inserting "(":
7. Inserting ")":
8. Inserting ")":
9. Inserting <Return>:

Fig. 5. Behaviour during typing

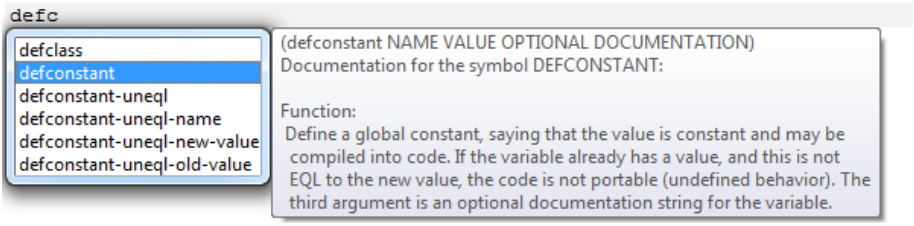

Fig. 6. Code completion

3.4 Implementation

The implementation of the new Editor Window containing the structure editor is based on the Graphical Editing Framework (GEF) provided by Eclipse [11].

GEF applies the Model-View-Controller (MVC) design pattern that explicitly separates the data structures themselves and the way they are displayed in the user interface. GEF is designed in a generic way so that any kind of model can be used. In our case, the model is the syntax tree that was parsed from the Lisp code. We extended the parser that came with CUSP to enrich the individual tree elements (e.g., to distinguish between different kinds of symbols like function names and keyword symbols).

Each model element is mapped to a figure which visualizes the different type of expression or symbol. Each change which is done using the structured representation in the user interface is reflected to the model. In some cases, an operation causes more than one change. For example, typing an opening parenthesis changes the whole structure because the following elements have to be moved into a newly created box. These modifications are performed in the model and afterwards the affected elements of the view are adjusted accordingly.

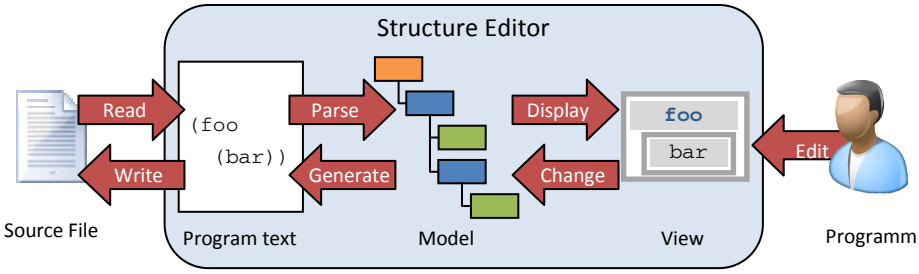

Fig. 7. Changing the model using the structure editor

Fig. 7 displays the whole process of editing a piece of code using the structure editor. First, the text is read from the source file and directly parsed to get the corresponding syntax tree. This is mapped to the figures that represent the individual elements of the tree. As mentioned before, each change which is done by the programmer is reflected back to the model. The corresponding Lisp code is not touched until the user saves the current document or changes to the textual representation. This means, using the structure editor, the programmer directly works on the syntax tree of the program.

The editor takes care of performing editing operations only if they result in a valid syntax tree. For example, it is not possible to paste code that contains unbalanced parentheses. If this is necessary, the programmer may circumvent this restriction (Principle 2) by switching to the textual representation to fix the appearing parsing errors. The code that was edited using the structure editor will not contain any structural parsing errors at all.

The following numbers give an impression of the extent of the implementation of the editor. The first one describes the newly created part of the plug-in (including some code that was taken from GEF samples) and the second one also incorporates the code of the already existing CUSP-plug-in.

Lines of code (structure editor):	8,290
Lines of code (entire plug-in):	25,571

4 Evaluation

In order to evaluate the usability of the structure editor in comparison to a common textual editor we conducted a survey.

4.1 Survey Preparation

Following Dumas and Redish, we presume that “usability means that the people who use the product can do so quickly and easily to accomplish their own task” [12, p.4].

We defined that the task we analyze by this evaluation is to understand the meaning and the structure of a piece of Lisp code – in other words: how the structure editor supports the readability and comprehensibility of the code. Considering that the users just need to read a piece of code, we decided to conduct a survey in terms of examining screenshots of the editor.

In literature, there are many metrics for analyzing the usability of software such as *effectiveness*, *efficiency*, *measures of learning*, and *subjective usability* [13], [14]. We focused on measuring the efficiency and a subjective rating of the usability.

To this end, three questionnaires were composed. Two of them show screenshots showing a piece of Lisp code and ten multiple-choice questions related to the meaning of the displayed code. We produced two versions of each questionnaire: one containing a screenshot of the code in textual representation and one containing a screenshot of the structure editor. This made the results comparable. In the third questionnaire, the participants were asked to rate how they experienced code reading in the two different representations and to give statements about things they liked or disliked in the screenshots of the structure editor.

4.2 Conducting the Survey

We conducted the survey with two different groups of participants. The first group consisted of second semester Bachelors' students (37 people). They had not known Lisp before. The second group was a team of Masters' students (13 people) who were engaged in a development project using Lisp and Prolog.

Both groups were randomly (according to their last names) divided into two groups and each group got one version of the first questionnaire. After exactly five minutes the students were told to stop working and to mark how far they got in answering the questions. For the second questionnaire, the groups were swapped: the group that worked on a questionnaire containing screenshots of the textual editor first then got the ones containing screenshots of the structure editor and vice versa. Again, the students had five minutes time to answer the questions. Finally, the students answered the third questionnaire.

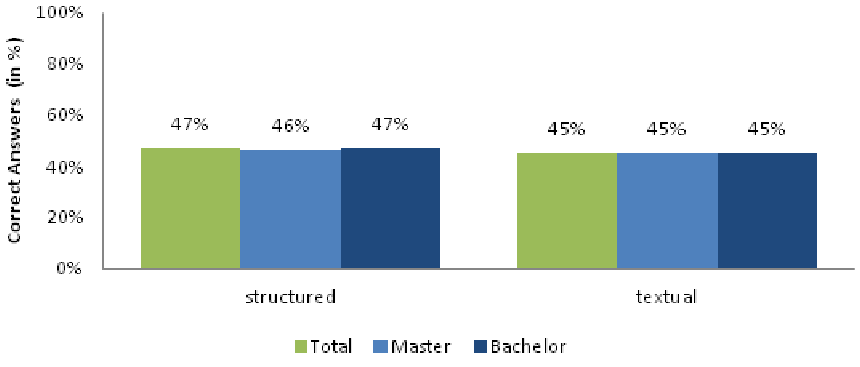

Fig. 8. Efficiency results

4.3 Results

As explained in Section 4.2, the first two questionnaires contained questions for comparing the efficiency in reading and understanding code in the two different representations.

Fig. 8 shows the cumulated results of this part of the survey in terms of the percentage of correct answers. As one can see, the results using the structure editor are slightly better (2%) but there is no significant difference.

We also examined how many questions the students managed to answer in the rather short period of five minutes. Fig. 9 shows the results. The students working with the structure editor did a bit better but, again, the difference is not significant.

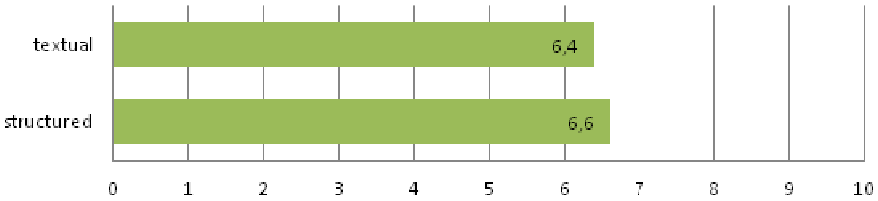

Fig. 9. Number of finished questions

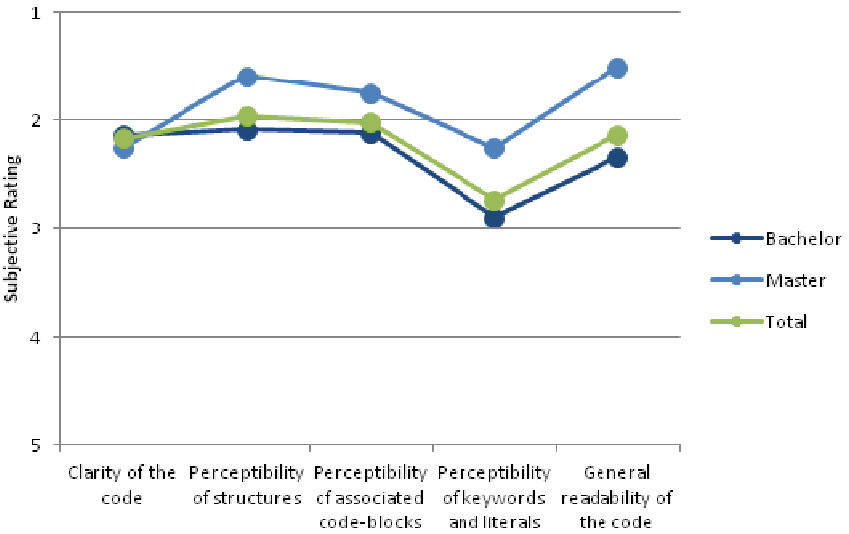

Fig. 10. Subjective rating of the structure editor

In the third questionnaire, the participants were asked to rate the structure editor compared to the textual editor regarding:

- Clarity of code
- Perceptibility of structures
- Perceptibility of associated code blocks

- Perceptibility of keywords and literals
- General readability of the code

The rating was possible within a range from “significantly better” (1) to “significantly worse” (5). A value of 3 means “no difference”. Fig. 10 shows the result. All ratings are in the positive half of the spectrum. Most ratings are close to 2 which means “better”. The Master students who were already experienced in working in Lisp gave better rates than the Bachelor students.

Most of the statements the participants gave about what they liked regarding the structure editor pointed in a similar direction. Several people wrote something like “code is clearly arranged” or “the structure is clearly visible”. However, a few people contrarily stated that they were confused by the structured representation of the code.

In general, the diagram indicates the subjective feeling of the participants that the structure editor helps them reading and understanding the code better.

As a last question, we asked the participants whether they would use such a structure editor if there was one for their favourite programming language. Fig. 11 shows that the majority (61% in total, 82% of the Master students) would at least give it a try. Students that voted negatively argued that they got used to their current editor and do not want to spend time in learning how to use a different one.

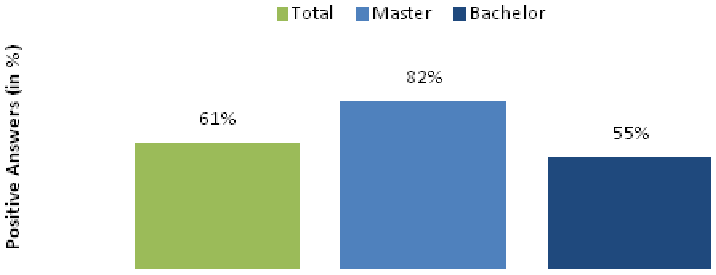

Fig. 11. Question "If there were a structure editor for your favourite programming language - would you use it?"

The evaluation does not reveal significant benefits of the structure editor as one may have expected. Nevertheless, it shows an interesting mismatch between the subjective ratings of the participants in the third questionnaire and the actual results from the first two questionnaires. This will be discussed in Section 6.1.

5 Related Work

We are not the first ones thinking about visualising the structure of a program in the editor and directly working on the syntax tree that was created from the code. In this section we present other approaches that were developed to achieve these goals.

5.1 Early Structure Editors

The idea of an editor which visualizes the structure of the underlying code is not new. In 1971, Wilfred J. Hansen presented a system called “Emily” [1] which was, in fact,

a structure editor for PL/I. The basic idea was to create a program by recursive replacement of placeholders according to their role in the Backus-Naur Form (BNF) notation of the programming language. The structure of the program and even of every command was fixed by structures of placeholders. Emily physically stored the whole program in a hierarchical structure that supported descending into sub-structures along the hierarchy. From the programmer's point of view, it was technically not possible to create programs that contained syntactical errors.

Other systems that follow a similar approach are "MENTOR" [15] and the "Cornell Program Synthesizer" [5]. Particularly in the Lisp community, programmers were fascinated by the idea of working directly on the structure of the code instead of a textual representation. An example of a structure editor for Lisp is Interlisp-D [16].

However, these early structure editors that are mostly summarized as *syntax-directed editors* could not satisfy the expectations and did not become widely accepted. Looking at these ancient examples which ran on terminals, restricted the programmers in several ways (violating Principle 2) and were quite tedious to use compared to a textual editor, this seems comprehensible. But what about newer systems based on the same idea?

5.2 Program Tree Editor

A more recent example of a structure editor of a different flavour is the "Program Tree Editor" [17]. This system visualises a piece of code written in a common programming language as a tree, similar to a file browser. It supports C, C++, C#, Java, Java Script, J#, XML, XHTML. Each tree node represents a structure from the underlying code and can be contracted and expanded. The tree structure is created upon opening a file containing source code and is translated back to the textual representation when a file is saved.

The user navigates through the tree using the keyboard and is able to edit the individual nodes directly in the tree. Nodes can be added or removed without the need of a mouse. Features like auto completion are provided.

This type of editor literally implements working on the underlying tree structure of the code. However, we question that this kind of visualization is particularly useful. We believe that programmers do not think in such file browser-like tree structures when they program. They more likely think in block structures. This is why we designed the GUI of our structure editor in a different way, consisting of nested boxes that emphasise the structure in a more discrete, but nevertheless clear way.

5.3 Subtext

A totally different approach of representing the structure of a program is presented as a system called *Subtext* [7]. Subtext is not based on an existing programming language. Instead, it introduces its own programming language that is not based on a textual representation of code any longer but stores its code in a database.

Using Subtext, the programmer composes programs from combining so called *schematic tables* which the author of the system describes as "a cross between

decision tables and data flow graphs" and which are intended to replace all kinds of conditional constructs. The basic idea behind such schematic tables is to visualize the structure of the program in two-dimensional way. The horizontal axis contains the different cases of a conditional statement ("deciding") and the vertical axis determines what happens if the individual cases become active ("doing").

Subtext seems to be quite an interesting approach for visualizing decision structures such as nested case statements. The greatest drawback appears to be its lack of compatibility. Subtext cannot be used to visualize the structure of already existing programs written in a common programming language (Principle 5).

5.4 A Structure Editor for C#

The most similar approach to our structure editor we are aware of is a *Structured Editor for C#* [18]. We regard it as the most capable editor of the ones we compared. This editor also represents the structure of the code in a discreet way by coloured bars at the beginning of each line. The actual bounds of a code block are shown as soon as one clicks on it using the mouse. All syntactic delimiters like curly brackets and semicolons are hidden, because they are not needed any longer.

One difference is, that the programmer is forced to change his way of typing. The delimiters are not only hidden, they are also not typed at all. For example, for entering the body of a C# class, the programmer has to press the <Return> key instead of typing a curly bracket. Our philosophy is that the programmer may type exactly the same code with the textual and the structural editor in order to minimize the learning curve and to easily switch between editors (Principle 4).

5.5 Structure Editors and Language-Oriented Programming

All structure editors that were mentioned so far try to be an alternative or extension to the textual editors that are normally used to read and edit programs. In different ways they visualize the structure of a program. In the context of language-oriented programming (LOP) [6, 9] there is one more step of abstraction where structure editors can be useful.

One main idea of LOP is to enable domain experts to contribute more directly to the programmatic solution of a problem by using a suitable Domain-Specific Language (DSL). Such a DSL may be an extension to an existing programming language (*internal DSL*) or a new language (*external DSL*), not necessarily a textual one. In the latter case, the program is created using a special kind of structure editor (sometimes called a *projecting editor*) that also performs a mapping from the DSL to an executable program. In this case, the structure editor is more than just an alternative view on the program – it is actually part of the language workbench.

A language workbench that provides a complete development environment for external DSLs is the "Meta-Programming-System" [6] by JetBrains, which includes an "editor language" to create structure editors for each newly developed DSL. Another example is the system called "Intentional Software" that was proposed by Simonyi [19].

6 Conclusions

In this paper we described layout and behaviour principles for structure editors and presented a new structure editor for Lisp. We also presented an evaluation of our editor. Taking this into account, we now try to answer the question: Are structure editors an old hat or a future vision?

6.1 An Interesting Mismatch

Structure editors have been around for decades. However, they have not succeeded in replacing classical textual program editors. We think that this is an interesting mismatch: on the one hand, the concept of displaying the underlying structure of a program and directly working on the syntax tree is intuitively attractive. On the other hand, this kind of editor has gained low acceptance in practice so far.

The results of our survey revealed this mismatch, too. The majority of the participants had the intuitive feeling that the structure editor was superior to the textual editor. However, the quantitative results showed no significant improvement. All this seems to suggest that structure editors are rather “old hat” than “future vision”.

Certainly, structure editors are no silver bullet for software engineering [20]. Understanding the concepts of a programming language or paradigm is far more difficult than coping with a particular syntax. For example, a programming novice who has understood the concept of classes, inheritance, and polymorphism will not have a major problem in getting acquainted with different syntaxes, be it curly or other parentheses or, instead, boxes in a structure editor. Insofar, one should not expect an extraordinary measurable improvement in usability.

The structured representation even has a drawback which most of the former implementations of structure editors were not really able to handle: Structure editors require a syntactically correct program to be able to determine the structure of the code and to display the structured representation. So, modifications that are quite small but lead to a change of the structure (e.g. moving a bracket from one line to another) are not possible without the support of the editor. So even though the idea of a structure editor itself is quite simple – the implementation is not.

Also, many programmers are reluctant to change their way of programming. Our survey confirmed this opinion. Some participants conceded that the structure editor might be useful, but they got accustomed to their favourite IDE and do not want to change tools without really having to. The benefit seems to be too small for most programmers.

This is why we integrated the structure editor into a popular IDE like Eclipse and also provided the textual editor as part of the plug-in. As a result, the programmers may just use the structured representation where this seems helpful – and perhaps find out that this applies in more cases than expected.

6.2 Structure Editors Are Still Useful

Taking into account the arguments from the previous section, structure editors most likely will not be able to replace textual editors that are embedded in powerful IDEs. However, we still feel that they can be useful.

A first step is, as just mentioned, to plug structure editors into an IDE like Eclipse and offer programmers the possibility to use it as an alternative to the textual editor. For example, the programmer may use the structured representation for reading and understanding the code because it provides a better overview. To edit the code, he or she then may switch to the textual perspective. Anyway, this kind of use would not justify describing structure editors as "future vision" of programming environments.

However, we see a growing field of special-purpose programming issues where, indeed, structure editors could provide a significant improvement: configuring an application, defining business rules, designing the layout of a GUI, specifying a business process, etc. For all those special-purpose programming issues, DSLs are becoming more and more popular. The ever-increasing number of XML dialects is an indication for this. We feel that structure editors are particularly useful for programming DSLs, or, in general, for Language-Oriented Programming (see Section 5.5).

Fig. 12 provides an example of a code snippet in some XML dialect in textual form and in the structure editor. The representation in the structure editor is by far more clearly arranged than in the textual XML syntax. This is particularly useful for novices or rare users of this particular XML dialect.

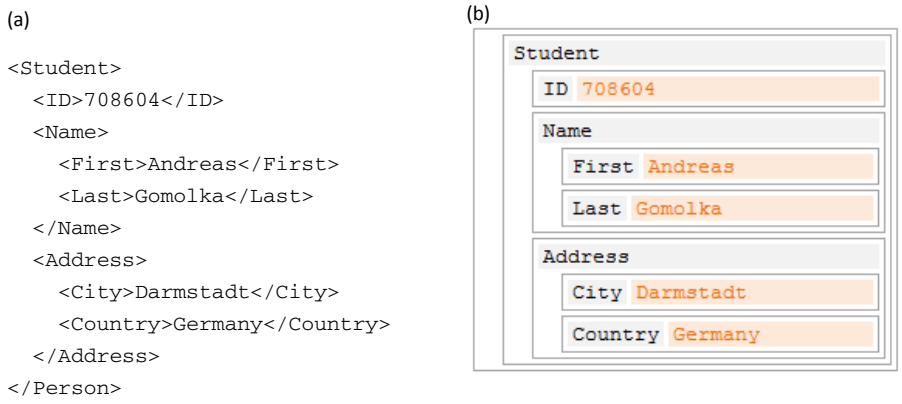

Fig. 12. A snippet of XML code in textual (a) and structured representation (b)

Conway et al. [21] and Myers et al. [22] have shown in comprehensive analyses that structure editors and graphical editors are most useful for programming novices, e.g., children. Since users of special-purpose DLSs are usually rare users and often novices we are confident that structure editors may be most useful in this context: a future vision for DSL editors.

6.3 Future Work

As future work, we plan to extend our evaluation of the structure editor towards its use in Language-Oriented Programming. In addition to readability and understandability of code we will examine the effects of the editor on the learning curve for DSLs as well as the effectiveness and efficiency of programming.

A program editor is a tool and no silver bullet. In the end, it is a matter of taste which kind of editor a programmer feels most appropriate for achieving a task – and this is a case for structure editors.

References

1. Hansen, W.J.: User engineering principles for interactive systems. In: Proceedings of the 1971 Fall Joint Conference (AFIPS 1971), pp. 523–532 (1971)
2. Borras, P., Clement, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B., Pascual, V.: Centaur: the system. In: Proceedings of the Third ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pp. 14–24 (1988)
3. Ballance, R.A., Graham, S.L., van de Vanter, M.L.: The Pan language-based editing system. ACM Transactions on Software Engineering Methodology (TOSEM) 1, 95–127 (1992)
4. Ko, A.J., Myers, B.A.: Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 2006), pp. 387–396 (2006)
5. Teitelbaum, T., Reps, T.: The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. Communications of the ACM 24, 563–573 (1981)
6. Dimitriev, S.: Language Oriented Programming: The Next Programming Paradigm. onBoard 1 (2004)
7. Edwards, J.: No Ifs, Ands, or Buts - Uncovering the Simplicity of Conditionals. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications, OOPSLA 2007 (2007)
8. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, Part I. Communications of the ACM 3, 184–195 (1960)
9. Humm, B.G., Engelschall, R.S.: Language-oriented Programming via DSL Stacking. In: Proceedings of the 5th International Conference on Software and Data Technologies (ICSOFT 2010), pp. 279–287 (2010)
10. Jasko, T., Ritchey, T.: CUSP. A Lisp plugin for Eclipse, `http://www.bitfauna.com/projects/cusp/`
11. The Eclipse Foundation: GEF and Draw2d Plug-in Developer Guide, `http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.gef.doc.isv/guide.html`
12. Dumas, J.S., Redish, J.C.: A practical guide to usability testing. Intellect Books, Exeter (1999)
13. Bevan, N.: Measuring usability as quality of use. Software Qual. J. 4, 115–130 (1995)
14. Schalles, C., Rebstock, M., Creagh, J.: Ein generischer Ansatz zur Messung der Benutzerfreundlichkeit von Modellierungssprachen. In: Engels, G., Karagiannis, D., Mayr, H.C. (eds.) Modellierung 2010, Klagenfurt, Austria, März 24-26, vol. 161, pp. 15–30. GI, Bonn (2010)
15. Donzeau-Gouge, V., Huet, G., Kahn, G., Lang, B.: Programming Environments based on Structured Editors: The MENTOR Experience. Institut National de Recherche d'Information et d'Automatique (INRIA), Rocquencourt (1980)
16. Burton, R.R., Masinter, L.M., Bobrow, D.G., Haugeland, W.S., Kaplan, R.M., Sheil, B.A.: Overview and status of DoradoLisp. In: Proceedings of the 1980 ACM Conference on LISP and Functional Programming (LFP 1980), pp. 243–247 (1980)

17. Yurov, A.: Program Tree Editor, http://www.programtree.com/
18. Osenkov, K.: Designing, implementing and integrating a structured C# code editor. Brandenburg University of Technology, Cottbus (2007)
19. Simonyi, C., Christerson, M., Clifford, S.: Intentional software. In: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 2006 (2006)
20. Brooks, F.P.: No Silver Bullet: Essence and Accidents of Software Engineering. IEEE Computer 20, 10–19 (1987)
21. Conway, M., Audia, S., Burnette, T., Cosgrove, D., Chistiansen, K.: Alice: lessons learned from building a 3D system for novices. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 2000), pp. 486–493 (2000)
22. Myers, B.A., Pane, J.F., Ko, A.: Natural programming languages and environments. Communications of the ACM 47, 47–52 (2004)