Context-Based Model for Concern Markup of a Source Code

[1] *M.S. Malevannyy <mmxforever@mail.ru>*
[2] *S.S. Mikhalkovich <miks@sfedu.ru>*
[1] *Don State Technical University,*
162, Socialisticheskaja st., Rostov-on-Don, 344022, Russia
[2] *Southern Federal University,*
8A, Mil'chakova, Rostov-on-Don, 344090, Russia

Abstract. In this paper we describe our approach to representing concerns in an interface of an IDE to make navigation across crosscutting concerns faster and easier. Concerns are represented as a tree of an arbitrary structure, each node of the tree can be bound to a fragment of code. It allows one to quickly locate fragments in the source code and makes switching between software development tasks easier. We describe a model which specifies data structures used to store the information about these code fragments and algorithms used to find the code fragment in original or modified source code. The model describes the information about code fragments as a set of contexts. Another important feature of the model is language independency. The model supports different programming, mark-up, DSL-languages and any structured text, such as a documentation. Main goal is to keep concern tree consistent with evolving source code. Search algorithm is designed to work with a modified source code, where the code fragment may change. The model is implemented as a tool, which supports different programming languages and integrates into different editors and integrated development environments. Source code analysis is performed by a set of lightweight parsers. In case of significant changes if the code fragment may be not found automatically the tool helps a programmer to find one by suggesting possible places in the source code based on the stored information.

Ключевые слова: Concerns; Separation of Concerns; Program Comprehension; Integrated Development Environments

DOI: 10.15514/ISPRAS-2016-28(2)-4

For citation: Malevannyy M.S., Mikhalkovich S.S. Context-Based Model for Concern Markup of a Source Code. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 2, 2016, pp. 63-78. DOI: 10.15514/ISPRAS-2016-28(2)-4

1. Introduction

During software development and maintenance developers usually work with several code fragments related to their current task or *concern*. Most concerns are crosscutting[1], which means that code related to it tends to be scattered across a number of files, or different places within one file. Repeated navigation between these code fragments requires a considerable time and effort[2]. These fragments form a "working set". Switching to another task requires investigating the source code and locating all fragments relevant to the task. Returning to the task after working on another one may take significant time.

A number of techniques address to this problem, such as Aspect-Oriented Programming[3], Feature-Oriented Programming[4][5][6], Delta-Oriented Programming[7], Subject-Oriented Programming[8] and others. Most of them are intended to explicitly separate concerns into a number of modules and provide different mechanisms of composition of these modules. It often requires significant changes in the source code to use one of these techniques.

Other methods provide support of concerns by adding new tools to an IDE, such as virtual files[9][10][11] colour markup[12] without changing the source code. These tools are often designed for only one IDE and depend on its infrastructure and thus are limited to only few languages, supported by the IDE. Another common limitation is low tolerance of changes in the source code. When the code is modified some code fragments may be lost.

Many of these tools are limited to only one programming language, while large software projects are often developed in several languages, including DSL-languages and markup languages, and code fragments related to a concern may be scattered across files in different languages.

We are currently developing an approach[13] intended to mitigate the problems of navigation across the code and switching between different tasks. The approach doesn't require any changes to the source code. It defines a notion of a concern as a tree-like structure, consisting of sub-concerns and code fragments. Similarly to ConcernMapper[14] it displays a concern tree in an IDE as a toolbox and allows one to quickly locate fragments in the source code. Unlike most other tools it and may be used in different IDEs and allows one to work with code in different languages. Another goal is robustness, which allows working with the code being actively developed keeping concern tree consistent with the code.

2. Model

We present a model our approach is based on. It uses lightweight parsers to analyze source text and to create parse tree, which will be used later. The model defines the data being stored in the concern tree. And finally, it defines algorithms to search the code fragments in a modified source code.

2.1. Lightweight parsing

The model is common for different languages. To minimize dependency on IDE infrastructure we use lightweight parsing to analyze the source code and build parse tree, which contains information about significant entities in the code. Lightweight parsers can recover from errors and produce parse tree for code with errors or incomplete code, which is important while the code is being modified.

Adding support of another programming language requires development of a lightweight parser for this language. Lightweight parsers are simple and easy to develop using our DSL language LightParse. For most languages it takes only about 10-30 lines of text to express important language features and produce a lightweight parser. The parser is able to analyze source code and build a simple parse tree with only nodes, corresponding to these language features. Any other parts of source code (e.g. method bodies) are skipped. Saving information about an entity in the source code is available for all entities returned by the parser. The more detailed parse tree the parser produces — the more entities can be saved in the concern tree, however development of the parser may require more time.

Lightweight parsers produce a lightweight parse tree. Nodes of the tree have type, name and location in the source code. Node name consists of several tokens; one of them may be marked as important. For example method name consist not only of one identifier — name, which is marked as important, but also includes parameter names and types, access modifiers, return value type and so on.

An example of a lightweight parser is given in subsection 2.3. Lightweight parsing is described in our paper[15] in more detail. The paper provides examples of lightweight parser grammar. More examples may be found in GitHub repository of the tool[1] (files with extension ".lp").

2.2. Data

The approach is not limited to any specific programming language and therefore the information in the concern tree should be sufficient to support different languages. Also, we assume that the source code may change and the concern tree should possibly store some redundant data to find the code fragment after the code has changed.

Each code fragment in the concern tree stores next 5 items:

- Type.
- Header context. It may include entity name and any number of additional tokens.
- Outer context. It includes names and types of all parent nodes from the immediate parent to the root of the parse tree.

[1] https://github.com/MikhailoMMX/AspectMarkup/tree/master/Parsers

- Horizontal context. It consists of two subsets of names and types of preceding and subsequent sibling nodes.
- Inner context. It includes a subset of subnodes of current code fragment.

These items form *Context* of the node. Except for type, any other item may be empty.

Type is used to filter non-relevant nodes when searching for the code fragments. If a concern tree item is bound to a method only methods should be considered, other nodes, e.g. classes, fields may be ignored.

Header context represents entity name and several additional tokens. In the following C# code example

```
public void visit(TreeNode t)
public void visit(Expression e)
```

both methods are named `visit`, but have different parameter types and names. Header context makes possible distinguishing overloaded methods and other entities with same names. Header context is represented as a list of tokens, where one token may be marked as important and it is considered as the **name** of the entity. Header context as well as name may be empty.

Outer context stores enclosing entities for the code fragment, such as classes and namespaces. In many languages there may be variables and methods with exactly same names, but defined in different classes or namespaces. An example is the implementation of one interface by different classes. In this case it's necessary to save not only the name of the entity, but also the name of enclosing entities. In the following example

```
namespace N
{
    class C1 : IVisitor
    {
        public void visit(IVisitor v) { }
    }
    class C2 : IVisitor
    {
        public void visit(IVisitor v) { }
    }
}
```

both methods have same names and header contexts, but are defined in different classes. For example, outer context for the first method will include name and type of class `C1` and namespace `N`. Outer context for an entity is a list of Header contexts and Types for each enclosing entity starting from the immediate parent to the topmost entity in the source file.

Header context and outer context are sufficient for most programming languages, where all names are unique, at least in a certain scope. However, there is another class of languages, such as Yacc (grammar definition language), or markup languages, such

as XML. In these languages there may be two entities with same name in same scope. Without additional information binding concern tree nodes to such entities is ambiguous. To handle these cases two different kinds of context were added to the model.

Horizontal context keeps nearest neighbors before and after the node. It consists of two sets of pairs (Header context + Type), one for preceding entities and one for subsequent entities. Following example is an excerpt from ANSI C grammar[16]:

```
selection_statement
    : IF '(' expression ')'
        statement ELSE statement
    ...
    ;
```

There are two occurrences of `statement` in a subrule of a rule `selection_statement`. Their horizontal contexts are different: token `ELSE` and another non-terminal `statement` are located *after* the first occurrence of `statement` and *before* the second one. This information makes it possible to distinguish similar entities by their location among their neighbor entities.

It could have been achieved by saving an *index* of the entity. For example, first `statement` gets index 1 and second one gets index 2, but saving indexes is less tolerant to changes in the source text. Adding or removing entities in the beginning of a subrule invalidates indexes of all subsequent entities, but has almost no effect on horizontal context.

Inner context is intended to store subnodes of an entity. In some cases an entity can have empty name and may be distinguished from another one only by its content. For example, variable declaration sections in such language as PascalABC.NET[17] are unnamed, but they have different variables:

```
var
    X, Y : Double;
var
    Name, Address : string;
    Age : integer;
```

In this example, there are two sections. It may be necessary to bind a concern tree node to a whole section. Horizontal context cannot be reliable in this case because it keeps only type and name, which is empty — changing their order will lead to incorrect result of the search. Inner context is a set of Header contexts and Types for some subnodes. In the example above saving only one subnode (i.e. variable name) is enough to distinguish these sections. Amount of subnodes to be saved as the inner context may vary.

Inner context for leaves of a parse tree may contain lines of source code. This may apply if the entity spans multiple lines in the source code (e.g. methods).

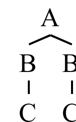
Inner and horizontal contexts may be empty if the entity has no neighbor nodes or subnodes. Otherwise. it may be not necessary to store all neighbors or subnodes. Usually, a small amount of unique nodes is enough to distinguish similar entities. In many languages horizontal and inner contexts are a redundant information. However, using horizontal and inner contexts increases reliability of the search even with a code on a programming languages that normally don't need these two kinds of context. When the code has changed this information may be useful.

Let T is a parse tree node. $Context(T) = (Name_T, Type_T, N_T, O_T, H_T, I_T)$ is a tuple of node Name, Type and its Header, Outer, Horizontal and Inner contexts described above. When a binding to the node T is added to the concern tree, $Context(T)$ is saved.

Name and *Type* are strings. Header context $N_T = (S_1, S_2, \dots S_n$ is a list of strings. Outer context $O_T = ((N_1, T_1), (N_2, T_2), \dots (N_n, T_n))$ is a list of pairs, where N_i is a Header Context and T_i is a type of an enclosing entity. Inner Context $I_T = \{(N_i, T_i)\}$ is a set of pairs: header contexts and type of an entity. And Horizontal context $H_T = (\{(N_i, T_i)\}, \{(N_j, T_j)\})$ is a pair of sets of header contexts and types of entities.

2.3. Additional markup

Our approach is focused on finding code fragments without using any modifications of source code. Additional markup, such as comments with special keywords clutters the code if used frequently. However, in some cases it might be feasible to mark some places in the code with comments. First scenario is binding to code fragments in a file, which contains a lot of very similar entities. Some XML files may have such structure. In this example:



There are two nodes *C*, with equal contexts. Despite being subnodes of different parent nodes, their outer contexts are equal, because both parent nodes have same name. To handle this case it might require to save horizontal context for each parent node, which is not implemented in the model.

Another scenario is binding to code fragments in frequently modified code, where entities may undergo significant changes.

This kind of markup requires a lightweight parser, which builds parse tree based on comments. Comments may define points and spans in the source code.

```
// ConcernBegin Serialization
...
// Concern SomePoint
```

```
...
// ConcernEnd Serialization
```

The code above shows an example of a markup with comments. Concern *Serialization* is a span and *SomePoint* is a single line marked with a comment.

Lightweight parser for this markup is simple and may work with source code in many languages. The only modification it may require to adapt the parser to a different language is changing comment start symbols. Here is a grammar of the lightweight parser written in LightParse:

```
%Extension "*"
Token Tk [[:IsLetterOrDigit:]_]*|
         [[:IsPunctuation:][:IsSymbol:]]
Token NewLine \r|\n|\r\n
Rule Program :   [#Comment|Other]*
Rule Comment :   "//" @CTk? @Tk+
Rule CTk:        @"ConcernBegin"
                 | @"ConcernEnd"
                 | @"Concern"
Rule Other :     Tk
                 | NewLine
                 | #error
```

3. Algorithms

There are two aspects of working with the concern tree: adding a node to the tree and searching the code fragment, related to the node. Both actions require a parse tree, which is provided by a lightweight parser. In the following part of the section we take into consideration only a subset of parse tree nodes whose type is equal to the type of an entity being saved or the one being searched. Given the *T* is a parse tree node to be saved in the concern tree, we consider a set $Tree = \{T_i \mid Type_{T_i} = Type_T\}$.

Next step is calculating a distance between *T* and every item $T_i \in Tree$.

3.1. Calculating distances

Distance two tree nodes is a vector of distances between each component of a context for a given pair of nodes.

$$Distance(T, T_i) = \overline{D_i} = (DName, DType, DN, DO, DH, DI)$$

where:

$$DType = \begin{cases} 1, if\ Type_T \neq Type_{T_i} \\ 0, if\ Type_T = Type_{T_i} \end{cases}$$

Distance for other part of context is calculated with functions *LDistance* and *SDistance*, described further below:

- $DName = LDistance(Name_T, Name_{T_i})$
- $DN = LDistance(N_T, N_{T_i})$
- $DO = LDistance(O_T, O_{T_i})$
- $DH = LDistance(H_T, H_{T_i})$
- $DI = LDistance(I_T, I_{T_i})$

Zero in each component of a vector $\overline{D}$ means equality of corresponding parts of contexts of *T* and T_i. The higher these values — the less similar two parts of contexts are.

Calculating the distance for Name, Header context and outer context is based on a Levenshtein metric [18]. Levenshtein distance for two strings reflects the number of edits (insertions, deletions and substitutions) required to change one string into the other. *Names* of entities are just strings, however Header contexts are lists of strings. Levenshtein distance in this case is calculated similarly, but each edit is a deletion, insertion or substitution of a token. Weight of a substitution in this case depends on similarity of tokens and ranges between 0 (tokens are equal) to 2 (weight of insertion + weight of deletion) if two tokens have maximum possible edit distance between them. Distance between two outer contexts is calculated similarly. Each item of an outer context is a pair (Type, Header Context) and the weight of substitution depends on distance between to header contexts.

Calculation of edit distance is performed by overloaded functions *LDistance*.

Horizontal and inner contexts contain a subset of nodes and the distance is calculated as a number of subnodes present in *T* and absent in T_i.

Calculation of distance between sets is performed by function *SDistance:*

$$SDistance(I, I_i) = |I \setminus I_i|$$

$$SDistance(H, H_i) = |H_L \setminus H_{i_L}| + |H_R \setminus H_{i_R}|$$

3.2. Saving information

Name, Type, Header and Outer contexts are required parts of a context and are saved always. Inner and Horizontal contexts are optional in some cases. To determine should they be saved or not and how much nodes they should contain we are looking for other nodes in the parse tree with similar Header Contexts.

Given the T is the parse tree node to be saved we define two sets of parse tree nodes:

$$TreeL = \{T_i \mid O_{T_i} = O_T\}$$

$$TreeG = \{T_i \mid O_{T_i} \neq O_T\}$$

In other words, one subset consists of all neighbour nodes for *T* (**L**ocal scope) and other one - of all other nodes (**G**lobal scope).

After that, we calculate two values: *NearL* and *NearG*.

$$NearL = LDistance(N_T, N_{T_i}) : T_i \in TreeL;\ \forall T_j \in TreeL, LDistance(N_T, N_{T_j}) \geq LDistance(N_T, N_{T_i})$$

In other words, we find a distance between header contexts of *T* and the most similar node *within* the scope of a node *T*.

$$NearG = LDistance(N_T, N_{T_i}) : T_i \in TreeG;\ \forall T_j \in TreeG, LDistance(N_T, N_{T_j}) \geq LDistance(N_T, N_{T_i})$$

similar to *NearL*, but *outside* of the scope of *T*.

When $NearG > 0$, $NearL > 0$ there are no other nodes with same header. In this case Inner and Horizontal contexts are optional and may be omitted. If NearG = 0, NearL > 0 there are similar nodes with different outer context. Again, saving Inner and Horizontal contexts is optional, but may improve search results if the source file is modified. In case of $NearL = 0$ saving inner and horizontal context is required.

The values *NearL* and *NearG* are saved within the concern tree and will be used for the search.

3.3. Searching

A node in the concern tree keeps Context of some node *T*.

$$Context(T) = (Name_T, Type_T, N_T, O_T, H_T, I_T)$$

After some modifications were applied to the source file, target node may change as well. In some cases target node may be absent in the parse tree, if the code fragment related to the concern was removed. We do not address this case in our research and the tool is designed to always try to find target node or suggest a list of most similar entities.

The search begins with parsing a file and calculating edit distance $\overline{D_i} = Distance(T, T_i)\ \forall T_i \in Tree$

Next step — checking if there is only one node in the tree, which is similar to the target node and therefore considered as the result of the search. It depends on values *NearG* and *NearL*.

If $NearL>0$, then there was only one entity in the source file with Header context H_T. In this case if there is only one node T_i with similar Header context in the tree — it is returned as the result:

$$Result = T_i \in Tree : LDistance(N_T, N_{T_i}) < \frac{Min(NearG, NearL)}{2};$$
$$\forall T_j \neq T_i\ LDistance(N_T, N_{T_j}) > \frac{Min(NearG, NearL)}{2}$$

If $NearL = 0$, then there were other entities in the source tree, but only in the same scope as *T*. In addition to the condition above we can return T_i if it has minimal distance for Header, Inner and Horizontal contexts among all other nodes:

$$Result = T_i \in Tree : \forall T_j \neq T_i : LDistance(N_T, N_{T_i}) \leq LDistance(N_T, N_{T_j})$$
$$SDistance(I_T, I_{T_i}) \ll SDistance(I_T, I_{T_j})$$
$$SDistance(H_T, H_{T_i}) \ll SDistance(H_T, H_{T_j})$$

These conditions are correct if $NearG > 0$. Otherwise there were other entities in the source file with same Header Context outside of the scope of *T*. In this case we add requirements $LDistance(O_T, O_{T_i}) = 0$ and $LDistance(O_T, O_{T_j}) = 0$ to both conditions.

If there are no exactly one node T_i, which satisfies the requirements above we consider the search result as ambiguous and cannot return only one node as the result. It may occur when the source code was modified significantly, the target entity was changed or removed and there are 0 or 2 or more nodes in the parse tree, similar to the target node. In this case the set of all nodes is sorted according to the product of $\overline{D_i} \cdot \overline{W}$, where vector $\overline{W}$ defines weights of parts of contexts.

3.4. Complexity

Wagner-Fischer algorithm[19] is used to calculate edit distances. It has a time complexity of $O(NM)$ where *N* and *M* are lengths of two strings. Calculating edit distance of Header Contexts requires calculating edit distance between two strings at each step. For simplicity, we assume that all tokens and all header contexts have similar length. It gives a time complexity of $O(N^2M^2)$, where *N* is the length of Header contexts (in tokens) and *M* is length of tokens.

Calculating edit distance between two Outer Contexts has a time complexity of $O(N^2M^2K^2)$, where *K* is a length of Outer Context (depth of the parse tree).

In most cases values *N*, *M* and *K* are relatively small. Length of separate tokens usually ranges between 1 and 10–15, longer identifiers are rare. Header Context contains usually not more than 10–15 tokens. Outer context in case of most programming languages contains 1–3 items (e.g. a namespace and a class).

Calculating edit distance is performed for each item in set *Tree*.

Other operations have a time complexity between $O(N)$ (calculating *NearG* and *NearL*, finding exact match) and $O(N \log N)$ (sorting), where *N* is a number of items in set *Tree*.

4. Tool

he tool[2] based on the model was designed to be easily integrated into different integrated developer environments and text editors, such as Microsoft Visual Studio and Notepad++.

---

[2] Available at https://github.com/MikhailoMMX/AspectMarkup

4.1. Architecture

The tool is separated into 3 main parts:

- A collection of lightweight parsers and a parser generator. A parser analyzes source files written in a specific language and provides a parse tree which is then used by the core. To make development of new parsers easier a DSL-language {\em LightParse} was implemented along with an utility which generates lex/yacc and C\# code of the parser from an input LightParse file.
- Core. It implements the model with algorithms. It loads and runs parsers to get a parse tree when it's necessary for saving or searching for a code fragment. A visual component with user interface ready to be integrated into different IDEs is also implemented.
- A collection of plug-ins for integrated development environments or text editors. Since the tool relies on lightweight parsers rather than on a specific IDE, and the visual part of the tool along with algorithms is provided by the core, the tool can be very easily integrated into different IDEs. A plug-in for an IDE should only display the UI component and implement simple interface, which defines 10 methods, such as getting and setting cursor position, accessing the text of currently open files and event handlers for opening and closing the IDE.

At this moment implemented lightweight parsers include: C#, Lex and Yacc, Java, XML, PascalABC.NET and a parser for our own language LightParse. Plug-ins for Microsoft Visual Studio, Notepad++ and PascalABC.NET[20] are developed and the tool is also integrated into a grammar editor Yacc MC.

4.2. Functionality

The tool adds a concern tree to the interface of a IDE. Concern tree may have arbitrary structure and is created by a developer. Each tree node has title and optional description and subnodes. Description length is not limited. It's displayed as a tooltip and may be edited in a separate window.

Each node may be bound to a fragment of code. In this case the node is marked with an arrow. Double click performs navigation to the code fragment if the code fragment may be identified unambiguously. Otherwise, the tool suggests several most similar code fragments. Each code fragment may be navigated to in one click and if the code fragment is found, double click updates the information in the concern tree, so next navigation will not require any additional actions.

A reverse search is also possible. The tool can find a node in the concern tree by cursor position in a current file. Along with the descriptions for tree nodes it may be used to extract some long comments from the code into the concern tree and still be able to easily find and read them.

There are several scenarios of using the concern tree. First, it may be used to maintain a "working set" of fragments, related to a current task. Concern tree is relatively small and finding the node in the tree may be much faster than finding the code fragment in one of currently open files manually.

Concern tree significantly simplifies re-creating working set when returning to a task. Instead of recalling class and method names, performing cross-reference search it's only necessary to expand a subnode in the concern tree related to the task.

Concern tree is very helpful when a new developer starts working with unfamiliar project. Concern tree resembles a table of contents, it's easy to find concerns in it and each concern contains all code fragments related to it with descriptions. Reading description and navigating across the code helps to understand how the code is organized and how it works.

The functionality, concern tree examples and the tool usage scenarios were presented at CEE-SEC(R) 2015 Conference[3].

5. Conclusion

We propose an approach to working with crosscutting concerns. Concerns are organized in a tree-like structure and tree nodes are bound to code fragments scattered across the project. Concern tree is added to the interface of IDE as a toolbox. Concern tree simplifies navigating across scattered fragments and is helpful for investigating and re-investigating a concern. We describe a model our approach is based on. A metrics of distance between entities in a code is defined. A description of data, stored in a concern tree is given. Algorithms of identifying a minimal amount of data to store and searching an entity in a modified source code are provided.

The model is implemented in a tool, which supports different programming languages and integrates into different editors end integrated development environments. It performs either navigation to a saved code fragment if it can be determined precisely, or shows most similar code fragments otherwise. The concern markup tool is used in development of PascalABC.NET and the tool itself.

At this moment some features of the model are not implemented yet, such as horizontal context.

We are currently collecting statistical data and enhancing algorithms to better handle most frequent changes in the source code. Some parameters, such as weights of operations need adjustments.

References

[1]. M. Eaddy, A. Aho, and G. C. Murphy, “Identifying, assigning, and quantifying crosscutting concerns” in Proceedings of the First International Workshop on Assessment

[3] http://2015.secr.ru/lang/ru/program/submitted-presentations/aspect-markup-of-a-source-code-for-quick-navigating-a-project

of Contemporary Modularization Techniques, ser. ACoM ’07. Washington, DC,USA: IEEE Computer Society, 2007, p. 2. DOI: 10.1109/ACOM.2007.4.

[2]. A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, “An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks” IEEE Trans. Softw. Eng., vol. 32, no. 12, pp. 971–987, Dec. 2006. DOI: 10.1109/TSE.2006.116.

[3]. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, “An overview of AspectJ” in Proceedings of the 15thEuropean Conference on Object-Oriented Programming, ser. ECOOP’01. London, UK, UK: Springer-Verlag, 2001, pp. 327–353. (http://dl.acm.org/citation.cfm?id=646158.680006)

[4]. D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin, “The genvoca model of software-system generators” IEEE Softw., vol. 11, no. 5, pp. 89–94, Sep. 1994. DOI: /10.1109/52.311067.

[5]. D. Batory, J. N. Sarvela, and A. Rauschmayer, “Scaling step-wise refinement” in Proceedings of the 25th International Conference on Software Engineering, ser. ICSE ’03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 187–197. (http://dl.acm.org/citation.cfm?id=776816.776839).

[6]. S. Apel, C. Kastner, and C. Lengauer, “Featurehouse: Language independent, automated software composition” in Proceedings of the31st International Conference on Software Engineering, ser. ICSE ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 221–231. DOI: 10.1109/ICSE.2009.5070523.

[7]. I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella, “Delta-oriented programming of software product lines” in Proceedings of the 14th International Conference on Software Product Lines: Going Beyond, ser. SPLC’10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 77–91. (http://dl.acm.org/citation.cfm?id=1885639.1885647).

[8]. W. Harrison and H. Ossher, “Subject-oriented programming: A critique of pure objects” in Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications, ser. OOPSLA ’93. New York, NY, USA: ACM, 1993, pp. 411–428. DOI: 10.1145/165854.165932.

[9]. M. C. Chu-Carroll, J. Wright, and A. T. T. Ying, “Visual separation of concerns through multidimensional program storage” in Proceedings of the 2nd International Conference on Aspect-oriented Software Development, ser. AOSD ’03. New York, NY, USA: ACM, 2003, pp. 188–197. DOI: 10.1145/643603.643623.

[10]. A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr., “Code bubbles: A working set-based interface for code understanding and maintenance” in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ser. CHI ’10. New York, NY, USA: ACM, 2010, pp. 2503–2512. DOI: 10.1145/1753326.1753706.

[11]. S. Chiba, M. Horie, K. Kanazawa, F. Takeyama, and Y. Teramoto, “Do we really need to extend syntax for advanced modularity?” in Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development, ser. AOSD ’12. New York, NY, USA: ACM, 2012, pp. 95–106. DOI: 10.1145/2162049.2162061.

[12]. C. Kästner, S. Apel, and M. Kuhlemann, “Granularity in software product lines” in Proceedings of the 30th International Conference on Software Engineering, ser. ICSE ’08. New York, NY, USA: ACM, 2008, pp. 311–320. DOI: 10.1145/1368088.1368131.

[13]. M. Malevannyy and S. Mikhalkovich, [Implementation of support of aspects in integrated development environments], in Sovremennye informatsionnye tekhnologii: tendentsii i

perspektivy razvitiya: materialy konferentsii [Modern information technologies: tendencies and perspectives of evolution], 2015, pp. 351–353 (in Russian).

[14]. M. P. Robillard and F. Weigand-Warr, “Concernmapper: Simple view-based separation of scattered concerns” in Proceedings of the 2005OOPSLA Workshop on Eclipse Technology eXchange, ser. eclipse ’05.New York, NY, USA: ACM, 2005, pp. 65–69. DOI: 10.1145/1117696.1117710.

[15]. M. Malevannyy, [Lightweight parsing and its application in development environment]. Informatizatsiya i svyaz’ [Informatization and communication], vol. 3, pp. 89–94, 2015, (in Russian).

[16]. ANSI C grammar. (http://www.quut.com/c/ANSIC-grammar-y.html)

[17]. PascalABC.NET. (in Russian). http://pascalabc.net/

[18]. V. Levenshtein, “Binary codes capable of correcting deletions, insertions and reversals” Soviet Physics – Doklady, vol. 10, no. 8, pp. 707–710,1965, (in Russian).

[19]. R. A. Wagner and M. J. Fischer, “The string-to-string correction problem” J. ACM, vol. 21, no. 1, pp. 168–173, Jan. 1974. DOI: 10.1145/321796.321811.

[20]. I. V. Bondarev, Y. V. Belyakova, and S. S. Mikhalkovich, [System pascalabc.net 10 years of evolution], in ”XX Nauchnaya konferentsiya Sovremennye informatsionnye tekhnologii: tendentsii i perspektivy razvitiya. Materialy konferentsii [XX Scientific conference Modern information technologies: tendencies and perspectives of evolution¨], 2013, pp.69–71, (in Russian).

Контекстно-ориентированная модель для разметки сквозной функциональности в исходном коде

[1] *М.С. Малеванный < mmxforever@mail.ru>*
[2] *С.С. Михалкович <miks@sfedu.ru>*
[1] *Донской Государственный Технический Университет,*
344022, Россия, г. Ростов-на-Дону, ул. Социалистическая, д. 162.
[2] *Южный Федеральный Университет,*
344090, Россия, Ростов-на-Дону, Мильчакова, д. 8А.

Аннотация. В данной статье описывается подход к упрощению работы со сквозной функциональностью в исходном коде за счет добавления к среде разработки средств разметки сквозной функциональности. Разметка представлены в виде дерева, отдельные узлы которого могут быть привязаны к блокам кода, обеспечивая быструю навигацию по фрагментам кода, реализующим сквозную функциональность. Привязка узлов дерева к коду осуществляется за счет сохранения в дереве набора информации о фрагментах кода. Сохраняемая информация содержит имя и тип фрагмента кода, а также несколько видов контекстов, которые позволяют однозначно найти фрагмент в коде. Эти контексты позволяют в рамках одной модели работать с кодом на различных языках, как программирования, так и языках разметки, DSL-языках, а также с любым структурированным текстом, например, документацией. Реализация алгоритмов поиска фрагмента по сохраненной информации учитывает возможность внесения изменений в код в процессе разработки, что обеспечивает *устойчивость* привязки. При небольших изменениях исходного кода фрагмент может быть найден автоматически. В случае более серьезных изменений реализован полуавтоматический поиск при минимальном участии

программиста. Исходный код анализируется легковесными парсерами, не полагаясь на инфраструктуру среды разработки. За счет этого достигается возможность работать с широким спектром языков, а также интеграция инструмента в различные среды разработки с минимальными усилиями. В статье представлена модель хранения данных, алгоритмы поиска, а также обзор инструмента, реализующего данную модель.

Ключевые слова: разделение ответственностей; аспекты; языки программирования; среды разработки

DOI: 10.15514/ISPRAS-2016-28(2)-4

Для цитирования: Малеванный М.С., Михалкович С.С. Контекстно-ориентированная модель для разметки сквозной функциональности в исходном коде. Труды ИСП РАН, том 28, вып. 2, 2016 г., стр.63-78 (на английском). DOI: 10.15514/ISPRAS-2016-28(2)-4

Список литературы

[1]. M. Eaddy, A. Aho, and G. C. Murphy, "Identifying, assigning, and quantifying crosscutting concerns" in Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques, ser. ACoM '07. Washington, DC, USA: IEEE Computer Society, 2007, p. 2. DOI: 10.1109/ACOM.2007.4.

[2]. A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks" IEEE Trans. Softw. Eng., vol. 32, no. 12, pp. 971–987, Dec. 2006. DOI: 10.1109/TSE.2006.116.

[3]. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ" in Proceedings of the 15thEuropean Conference on Object-Oriented Programming, ser. ECOOP'01. London, UK, UK: Springer-Verlag, 2001, pp. 327–353. (http://dl.acm.org/citation.cfm?id=646158.680006)

[4]. D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin, "The genvoca model of software-system generators" IEEE Softw., vol. 11, no. 5, pp. 89–94, Sep. 1994. DOI: /10.1109/52.311067.

[5]. D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement" in Proceedings of the 25th International Conference on Software Engineering, ser. ICSE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 187–197. (http://dl.acm.org/citation.cfm?id=776816.776839).

[6]. S. Apel, C. Kastner, and C. Lengauer, "Featurehouse: Language independent, automated software composition" in Proceedings of the31st International Conference on Software Engineering, ser. ICSE '09.Washington, DC, USA: IEEE Computer Society, 2009, pp. 221–231. DOI: 10.1109/ICSE.2009.5070523.

[7]. I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella, "Delta-oriented programming of software product lines" in Proceedings of the 14th International Conference on Software Product Lines: Going Beyond, ser. SPLC'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 77–91. (http://dl.acm.org/citation.cfm?id=1885639.1885647).

[8]. W. Harrison and H. Ossher, "Subject-oriented programming: A critique of pure objects" in Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications, ser. OOPSLA '93. New York, NY, USA: ACM, 1993, pp. 411–428. DOI: 10.1145/165854.165932.

[9]. M. C. Chu-Carroll, J. Wright, and A. T. T. Ying, "Visual separation of concerns through multidimensional program storage" in Proceedings of the 2nd International Conference on Aspect-oriented Software Development, ser. AOSD '03. New York, NY, USA: ACM, 2003, pp. 188–197. DOI: 10.1145/643603.643623.

[10]. A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr., "Code bubbles: A working set-based interface for code understanding and maintenance" in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ser. CHI '10. New York, NY, USA: ACM, 2010, pp. 2503–2512. DOI: 10.1145/1753326.1753706.

[11]. S. Chiba, M. Horie, K. Kanazawa, F. Takeyama, and Y. Teramoto, "Do we really need to extend syntax for advanced modularity?" in Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development, ser. AOSD '12. New York, NY, USA: ACM, 2012, pp. 95–106. DOI: 10.1145/2162049.2162061.

[12]. C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines" in Proceedings of the 30th International Conference on Software Engineering, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 311–320. DOI: 10.1145/1368088.1368131.

[13]. М.С. Малеванный, С.С. Михалкович, Реализация поддержки аспектов программного кода в интегрированных средах разработки. Современные информационные технологии: тенденции и перспективы развития, 2015, стр. 351–353.

[14]. M. P. Robillard and F. Weigand-Warr, "Concernmapper: Simple view-based separation of scattered concerns" in Proceedings of the 2005OOPSLA Workshop on Eclipse Technology eXchange, ser. eclipse '05.New York, NY, USA: ACM, 2005, pp. 65–69. DOI: 10.1145/1117696.1117710.

[15]. М.С. Малеванный, Легковесный парсинг и его использование для функций среды разработки. Информатизация и связь, том 3, стр. 89–94, 2015.

[16]. ANSI C grammar. (http://www.quut.com/c/ANSIC-grammar-y.html)

[17]. PascalABC.NET. http://pascalabc.net/

[18]. В. И. Левенштейн. Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады Академий Наук СССР, 1965. 163.4, стр. 845–848.

[19]. R. A. Wagner and M. J. Fischer, "The string-to-string correction problem" J. ACM, vol. 21, no. 1, pp. 168–173, Jan. 1974. DOI: 10.1145/321796.321811.

[20]. Бондарев И. В., Белякова Ю. В., Михалкович С. С. Система программирования PascalABC.NET — 10 лет развития // XX Научная конференция «Современные информационные технологии: тенденции и перспективы развития». Материалы конференции. Ростов н/Д, 2013. С. 69–71.