2012 IEEE 36th International Conference on Computer Software and Applications

# XMLSnippet: A Coding Assistant for XML Configuration Snippet Recommendation

Sheng Huang, Yanghua Xiao*, Yiqi Lu, Wei Wang
School of Computer Science
Fudan University
Shanghai, China
{shhuang, shawyh, 10210240030,
weiwang1}@fudan.edu.cn

Yu Wang
IBM China Research Lab
Shanghai, China
wangyush@cn.ibm.com

***Abstract*—Framework-based applications are quite popularly used in current commercial applications. Framework-based applications are often controlled by XML configuration files. However, most of these frameworks are complex or not well documented, which poses a great challenge for programmers to correctly utilize them. To overcome these difficulties, we propose a new method to recommend XML configuration snippets automatically through mining tree patterns from the application repository with the aim of assisting the programmer to generate proper XML configurations during the product phase. This method is further realized into a new tool XMLSnippet. In this paper, we will systematically present the core techniques of the proposed method, including closed frequent tree pattern mining, prefix tree based indexing and three types of queries (type query, structural query and context query) for online coding assistance. Experimental results show that these techniques are efficient for recommending reusable XML configuration snippets and consequently improve the productivity of programmers effectively with the assistance of the tool when they program with XML-based frameworks.**

***Keywords- XML; Code Reuse ; Code Generation***

## I. INTRODUCTION

In today's software development, developers often utilize XML based frameworks to simplify implementation or make the programs more structural. Developers in general need to develop XML configuration files and client code to implement framework-based programs. See Fig. 1 as an example. To implement the typical login logic by Struts (one widely used framework to implement MVC in Java Web Applications), we need to build the XML configuration (Fig. 1(b)) and the client code (Fig 1(a)). Generally speaking, in this kind of framework- based programming, the imported framework packages (Jars for Java based frameworks) and the XML configuration files work together with client code to control the logic of the application.

Despite the benefits of framework-based programming, correctly and effectively utilizing the frameworks is still a great challenge. In reality, defects are easily introduced into the software due to the following reasons. (1) Documentation is always insufficient or is badly produced for a multitude of reasons. (2) It is usually the case that programmers improperly edit the XML configuration files while the IDE (Integrated Development Environment) gives no prompt messages. These facts lead to incorrect and inefficient usage of the frameworks in the coding phase.

Automatic code recommendation is one of promising solutions to reduce the probability of introducing defects caused by the complexity of frameworks or misunderstanding of poor documentation. In framework-based applications, code recommendation for XML configuration file building is expected to generate reusable XML substructure, sub elements/attributes, or samples to input element/attribute values so as to assist building XML configuration files for target application.

By automatic code recommendation, we can assist coding in the following target scenarios. For example, when using Struts, a freshman may often meet the following problems: "I know I need to use Struts, but I do not know the basic elements that I need to input "; "I know I need to use the <action mapping> element to manage the action mapping of UI logic, but I do not know the proper attributes and sub-elements combination that I need to use"; "I know I need to use some attributes to decorate an action element, but I don't know what value to input as the attribute value", etc.

However, to the best of our knowledge, no existing academic work addressed the problem of recommending code for XML configuration file building, which is a distinctive and important part of programming with frameworks. Most of previous academic works [2-6, 8, 9] focus on recommending language-level sample code through simple association rule, sequence associations, etc. Their solutions cannot be applied for code recommendation in framework-based programming due to the structural complexity of XML configuration files, and semantic complexity of application context, etc.

Neither existing open source/commercial industrial tools can be directly applied for code assistance scenarios required in framework-based programming. Tools like Eclipse [2], Eclipse XML Editor [20], and Visual Studio 2010 [18] could only generate XML snippets based on templates or XML Schema. None of them can recommend XML code based on the application context or the frequently used elements/attributes combination

Automatic code recommendation for framework-based programming now is possible due to the availability of (1)

* correspondence author: Yanghua Xiao (shawyh@fudan.edu.cn)

0730-3157/12 $26.00 © 2012 IEEE
DOI 10.1109/COMPSAC.2012.42

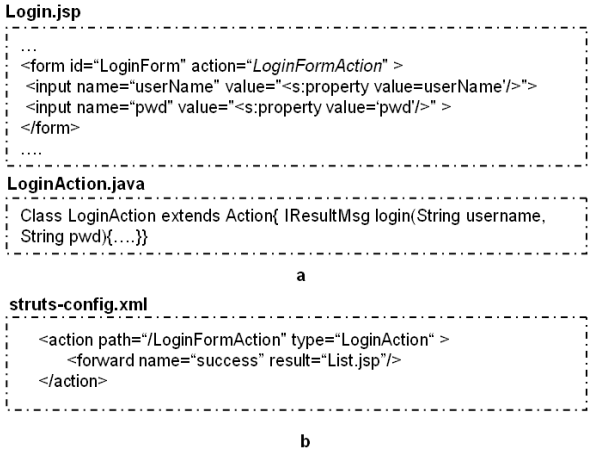


Figure 1. Book Store Sample Code for configuration

massive open-source framework-based application repositories and (2) well-developed XML data mining and query solutions.

In this paper, we propose a novel method to recommend XML configuration code snippets. We realize the method in a new tool named XMLSnippet, which mines closed frequent tree patterns from XML configuration files in application repositories and automatically recommends appropriate patterns through reusable XML snippets for those frameworks requiring XML format configuration files. We have also delivered XMLSnippet as an Eclipse [17] plug-in tool that can be invoked from within Eclipse. We also conduct a series of initial experiments to show that our tool could significantly shorten the development cycle for both experienced and inexperienced programmers.

The rest of the paper is organized as follows. Section 2 describes the background and sample usage scenario of our tool. Section 3 gives a detailed anatomy of the implementation of this tool together with the formal query model and mining model. Experimental evaluation of XMLSnippet is given in Section 4. We present related work in Section 5 and conclude the paper in Section 6.

## II. DESIGN OBJECTIVES

In this section we introduce the background of sample code recommendation and the objective usage scenarios of this tool by a typical example in framework-based programming.

### A. Background

The popularly used IDE Eclipse is designed to prompt reusable code samples with hot key "Alt+/". For example, if you enter "for" in the IDE and push "Alt+/", the IDE will list all popularly used candidate code samples/snippets to do repetition computing. Such kind of code reuse is realized by hard coded programming rules. The Eclipse IDE provides plug-in points to manually set the programming rules in path of "windows | Preference | Java | Editor | Templates"… If we realize XML snippet recommendation by hard coding programming rules as used in Eclipse, it will be time-consuming to prepare all the programming rules, and

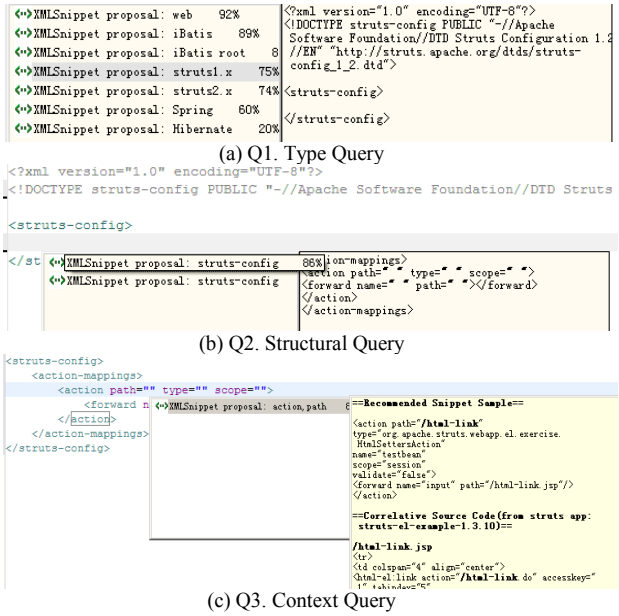


Figure 2. Sample usage scenario of XMLSnippet

consequently the framework providers in general do not provide such a facility.

Besides code recommendation in IDEs like Eclipse, there are also many research works [2-6, 8, 9] to recommend language level sample codes for reusing code. In contrast, this paper will only focus on structural XML configuration snippets recommendation as a complement to existing works. One way for XML snippet recommendation is directly generating sub elements/attributes of a certain element based on the DTD [19] schema of the XML files, like Visual Studio 2010 [18] and Eclipse XML Editor [20]. In fact, each XML configuration file always has a DTD schema (or can be generated) that gives all possible sub elements of a certain element. However, DTD based recommending may confuse programmers because sub elements/attributes always have equal probability to be selected and consequently the sub elements or attributes will be recommended to programmers without discrimination. To solve this problem and extend these tools, we need resort to sophisticated experts of a specific framework to help us summarize frequently used subtrees full of semantic composed of certain elements and attributes so that we can recommend the frequently used subtrees. However, such summarization is framework-dependent and need to be done case by case.

Compared to the schema-based XML snippet recommendation, the approach proposed in this paper is expert-independent and more efficient since it can automatically extract subtrees with ranked priority from existing application repositories. Recommending XML snippets by tree pattern mining also enables our approach to be applied on any framework based applications that rely on XML configuration files.

### B. Sample Usage Scenarios

We use the Struts [13] framework as a running example (shown in Fig. 1) to illustrate the usage scenarios of

XMLSnippet. At the Presentation tier, Struts implements MVC by the following steps: 1) the user submits a Form through a JSP page; 2) a Controller class of Struts passes the Form to a specific Action class, which is guided by a configuration file named struts-config.xml; 3) the Action class is executed and returns a result string (e.g. "success", "fail"); 4) the Controller directs the control flow to the next JSP page as specified in the same configuration file. In this example (as shown in Fig. 1(a)), Login.jsp defines a LoginForm which will be submitted to the server when a user logins. The struts-config.xml file has an <action …> element that relates this request to a Java class LoginAction, and also a <forward …> element that directs the "success" login result to the next JSP page List.jsp, where ordinary users can view existing books, and a bookstore administrator can update or delete existing book items or create new books. In the above example, without any tools, a programmer needs to manual edit elements, values or even the structure of XML configuration files, which is time-consuming and error-prone.

Above problems caused by manually XML editing has motivated us to develop XML snippet recommendation tools. Such tools generally are designed with the purpose to reuse existing codes to mine effective patterns to be recommended. Generally speaking, there are three levels of code reuse/generation: (L1) samples for programmers to learn from, which could not be reused directly by copy-paste operation; (L2) copy-paste with necessary manual modification; (L3) directly reusable samples without modification. In the following part of this section, we will show the detailed steps to create the configuration of struts-config.xml through three types of queries proposed in this paper to generate reusable XML snippets.

- Type query (*TQ*). The query to get the recommendation of one specific type of framework as the initial content of an empty XML document. The snippets returned by *TQ* belong to L3.
- Structural query (*SQ*). The query to get the subtree recommendation according to the position of the current cursor in the XML document under editing. *SQ* belongs to L2 since the snippets of basic tree patterns returned by *SQ* consist only of elements or attribute names and the programmer still needs to input attribute values.
- Context query (*CQ*). The query to get the reference sample code to help programmers to input the context sensitive attribute/element values. Obviously *CQ* belongs to L1 code reuse.

Suppose the programmer is a new learner of Struts, he needs to search the Web to find some specifications or samples to write such kinds of configuration snippets. The learning curve may be too long before he could be proficient. With XMLSnippet, he only needs to know what kinds of framework he plans to use and some basic knowledge of the framework. Continue our running sample. Suppose a programmer needs to create an XML document in Eclipse IDE. He just needs to trigger the hot key "Alt+/" in the empty XML document to retrieve recommended tree patterns of different frameworks by *TQ*. The initial recommendations are the high-level tree patterns of most frequently used frameworks in the code repositories. As Fig. 2(a) shows, XMLSnippet prompts with the frequent tree patterns. Each of these patterns is a certain framework. The percentage after the framework name shows the occurrence frequency of this type of framework in the application repositories. Here 75% of applications used Struts 1.x as a MVC framework.

After the programmer moves to Struts 1.x, the reusable detailed content popups on the right (see Fig. 2(a)). Then, an empty line is preserved to remind programmers to input sub-elements. Once the programmer moves the cursor to the empty line, he could ask XMLSnippet for the proposal of subtrees by *SQ*. As Fig. 2(b) shows, the tool will also list the subtree candidates ranked by occurrence frequency. Each proposal shows the popularly used subtree representing atomic logic controlled by Struts. Generally speaking, the hotspots are highly recommended since they are more frequently used, having a high confidence of safety [6]. The programmer could raise such a query any time after specifying the framework type by *TQ*.

Notice that the sample XML snippets proposed by the tool do not include the attribute value or element value because these values always depend on the context of the project under development. As illustrated in Fig. 2(a), the hotkey in the attribute value blank will trigger *CQ* to propose <sample XML fragments, Source code> pairs ranked by the frequency of the pattern extracted from the sample XML fragments. The programmer can select the proper XML snippets from the popup sample snippets list by moving the cursor up and down. The attribute value of the sample XML snippet and its corresponding occurrence in the client source code is simultaneously highlighted (see the attribute value "/html-link" in Fig. 2(c) as an example). By navigating correlative source code fragments, the programmer knows that he needs to input the html form action as the value path attribute, and action class to handle the logic of the form, etc. Finally, he may build the complete XML fragment (shown in Fig. 1(b)) to control the logic of login by struts. Then, the programmer could add other elements similarly with the assistance of XMLSnippet by hot key "Alt+/".

In summary, XMLSnippet is expected to provide the above three types of query to support different levels of code reuse with the following characteristics:

- Generality. It should be able to be applied on any framework-based applications that rely on XML configuration files.
- Effectiveness. The proposed code samples or XML snippets are useful and meaningful to shorten production time cost.
- Usability. It should be helpful for programmers with different skill levels and background.
- Efficiency. The tool should be efficient enough to support online coding suggestions.

From the sample usage scenario and the desired characteristics of XMLSnippet, we may imagine that the tool could help programmers by saving time of typing all the elements, reducing the chance of an incorrect combination of attributes, and finding the accurate attribute/element value.

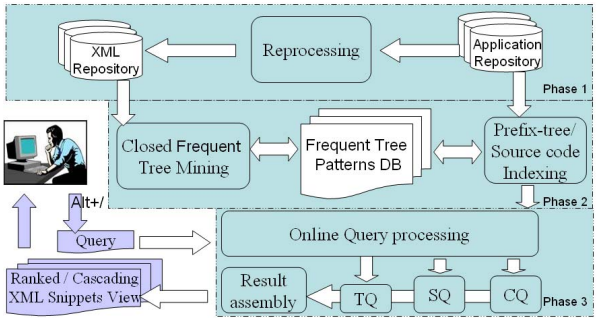

Figure 3. Overview of XMLSnippet

The next section introduces the detailed approach to achieve above objectives.

## III. XMLSNIPPET APPROACH

The architecture of XMLSnippet is shown in Fig. 3. The architecture consists of three major phases: pre-processing, offline mining and online query. In phase 1, application repositories are first preprocessed, and then the result (XML Repository) is fed to the closed frequent tree mining. All the resulting patterns are stored in the frequent tree patterns database. All paths starting from the root of all tree patterns are indexed by a prefix tree (also known as a trie). The ample source code is indexed by an inverted index that is used to speedup *CQ*. The cores of XMLSnippet are closed frequent tree mining and prefix tree & source code indexing. Frequent tree pattern mining distinguishes our tool from other code recommendation tools that utilize frequent item set patterns [5], or sequential patterns [6]. The online query phase is responsible for providing efficient and effective online assistance for programmers.

### A. Preprocessing

In general, it is hard to find meaningful patterns from frameworks serving different proposes. For example, iBATIS and Struts generally will not share a meaningful frequent pattern. Hence, we only mine patterns within applications of the same framework. For this purpose, we first need to classify input XML configuration files into different categories. Many typical features that can be indentified from XML docs in framework programming, such as "SQLMap" in iBATIS, "struct-config" in Struts, are sufficient for us to correctly classify XML documents into corresponding category by framework type. The mining in phase 2 is processed over separate groups of framework XML documents one by one.

### B. Offline processing

#### 1) Closed Frequent Tree Mining

All XML documents are modeled as unordered labeled rooted trees as [10,11] do, where each node in the tree is an element, or attribute. More formally, any attribute of an element will be a child of the element node; a sub-element will be a child of its parent element. As an example, Fig. 4 shows the XML tree of the XML doc in Fig. 1(b). We use a triple $T = (N, E, ft)$ to denote an XML tree extracted from an XML configuration file, where $N$ is the set of nodes

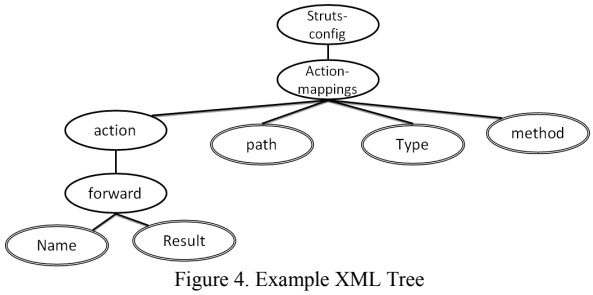

Figure 4. Example XML Tree

(including two types of nodes, one is element node $N_e$, the other is attribute node $N_a$), $E$ denotes the edge between parent and child, and $ft$ indicates the framework types of $T$. Notice that the value of attributes are ignored in the tree structures since they generally vary in different projects.

Let $D$ be a database of XML trees. For a given pattern $t$, we say $t$ occurs in a XML tree $s \in D$ if $t$ is a subtree of $s$. The support of a pattern $t$ in the database $D$ is defined as the number of XML tree $s \in D$ such that $t$ is a subtree of $s$. A pattern $t$ is called frequent if its support is greater than or equal to a minimum support specified by a user. The frequent tree mining problem is to find all frequent subtrees in a given XML tree database. One nice property of frequent subtrees is the apriority property, i.e., if $t$ is frequent, all its subtrees are also frequent. A frequent tree $t$ is closed if none of the proper supertrees of $t$ has the same support that $t$ has.

Given the collections of XML trees, reusable XML snippets for frameworks can be found by frequent tree pattern mining [10,11]. Notice that we only mine closed frequent trees instead of complete frequent trees due to the following reasons. In general, there are fewer closed frequent trees compared to the total number of frequent subtrees [11] and we can recover all frequent subtrees with their supports from the set of closed frequent trees with their supports [11]. Hence, mining closed frequent trees is an effective way to mine informative yet non-redundant patterns from XML configuration repositories.

In general, different frameworks have disparate syntax or semantics, which is one of the challenges in sample code recommendation for framework-based applications. However, by predigesting sample code recommendation problem to closed frequent tree mining, all syntax or semantics of different frameworks are encapsulated into a unified model: tree patterns, because the combination of attribute syntax and the hierarchical relationship of elements are all preserved in tree patterns. Hence, we can deduce our tool based on tree pattern mining is generalized.

Most previous researches on coding assistance reduce the problem to frequent item set mining or frequent sequence mining [5, 6]. Compared to them, frequent closed tree mining preserves the structural information expressed by the XML configuration files. Structural patterns are more informative than those linear patterns such as frequent item set or frequent sequence. Therefore, tree pattern mining provides us opportunities to find more interesting patterns and more hidden knowledge from framework-based code repositories. We leverage the algorithm proposed in [11] to extract the informative tree patterns hidden in XML

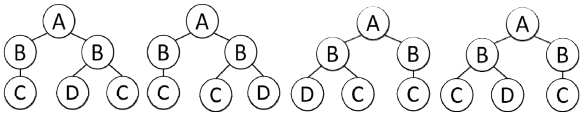

Figure 5. Illustration of DFCF code

configuration files. The mining procedure is shown in Algorithm *CMTreeMiner*, which follows the pattern-growth approach. The algorithm starts from the generation of seed patterns, which are frequent edges. Then, the algorithm grows patterns from these candidate one-edge trees by adding one edge at a time until the pattern is the largest one under a certain frequency.

From a rooted unordered tree we may derive more than one rooted ordered trees, which may cause redundant generation of candidate patterns. To solve this problem, a canonical form DFCF (depth-first canonical from) [11] will be used to represent an unordered labeled tree uniquely. Fig. 5 shows four trees derived from one unordered labeled tree. The DFCF codes of the four trees in the figure (from *a* to *d*) are ABC$$BD$C#, ABC$$BC$D#, ABD$C$$BC#, and ABC$D$$BC#, respectively, where '$' denotes null and '#' denotes the end of encoding. Finally, we select the one with the minimal lexicographical order as the DFCF code of the unordered tree. In Fig. 5, the depth-first traverse code of tree (d) is the DFCF for the unordered tree.

---

*Algorithm 1: CMTreeMiner(D,minsup)*
*Input: the tree database D, the minimum support threshold minsup*
*Output: the set of all closed frequent tree patterns CL*
1. *CL ← null*
2. *C ← frequent 1-trees*
3. *CM-Grow(C,CL,D,minsup); //grows patterns one edge at a time*
4. *return CL*

---

*Subroutine: CM-Grow(C,CL,D,minsup)*
1. *for each t ∈C:*
2. *Generate candidate trees patterns with one more edge, Let E be the set of these patterns*
3. *if E ≠null*
4. *then CM-Grow(E,CL,D,minsup)*
5. *if cannot grow any more*
6. *CL ←CL ∪{t}*
7. *return*

---

In step 2 of the subroutine CM-Grow of Algorithm CMTreeMiner, various heuristic rules that are based on both left and right *blanket* (each of which is a supertree having just one more vertex than the given subtree) and POSET (partially ordered set) structure of all frequent subtrees, will be used to avoid generating redundant candidate frequent subtree patterns in *E*. In the algorithm, the blanket is also used to determine the closeness and maximum of a frequent subtree patterns, as well as to prune branches in the POSET that do not yield closed frequent trees.

The distinctive characteristics of XML trees in framework-based programming allow us to speed up the closed frequent tree pattern mining procedure mentioned above. First, in an XML tree, there might exist more than one leaf such that the paths starting from the root to the leaves are identical to each other if the detailed attribute values are ignored. In these cases, we only keep one such path to reduce the size of each XML tree and thereby improve the efficiency of the mining procedure. Second, we find that most elements have only one or two possible heights in different XML trees. For example, the "action" element in Struts frameworks has height 3 or 4 in almost all XML configurations in the Struts framework. Based on this fact, we first record the possible heights for a certain element, and then use this information as a pruning rule to avoid generating unnecessary patterns. Finally, XMLSnippet only needs those patterns containing the root element. Therefore, all those patterns without the root node will not be generated. All these factors can be efficiently utilized to speed up the pattern mining procedure.

*2) Prefix Tree and Source Code Indexing*

---

*Algorithm 2: ConstructIndex*
*Input: frequent tree pattern set FTP*
*Output: Trie-like path index tree IDXT*
1. *IDXT ← null*
2. *for each frequent tree pattern t in FTP:*
3. *for each root-to-leaf path p ∈T:*
4. *if IDXT has path p:*
5. *add T to the leaf*
6. *else:*
7. *insert path p into IDXT*
8. *initialize path p's leaf with T*

---

*Algorithm 3: Retrieve*
*Input: a prefix path p, index tree IDXT*
*Output: snippet set S //i.e. the set of tree patterns containing p*
1. *if path p is found in IDXT*
2. *S ← null*
3. *for every child node n of the end of p*
4. *S ←S ∪{n}*
5. *sort patterns in S into the descending order of their support*
6. *return S*
7. *else*
8. *print XMLSnippet has no proposal*
9. *return null*

---

Once the frequent tree patterns (*ftp*) are extracted, we need to build an index to accelerate the retrieval of these patterns, ensuring the online performance of coding assistance. Suppose a programmer has finished editing a node of the XML tree (refer to Fig. 4 as an example), we use the location of the current cursor, denoted by *c*, to indicate the node that has just been edited in the following text. We call the root from the root of the XML tree to *c* as the prefix of current cursor. Such a prefix is essentially an element/attribute sequence from the root to the node just before the current cursor. Our tool needs to retrieve tree patterns that contain exactly the same prefix. For this purpose, we can extract all paths starting from the root (i.e. element/attribute sequences) of XML tree patterns in our repository, and then build a trie/prefix tree [18] on all these sequences. We use *ftp*(*p*) to denote the function that gets the set of *ftp* containing element/attribute sequence *p*. For example, suppose the node just before the current cursor is "Action" (as in Fig. 4), then *ftp*(*p*) will help us directly retrieve the tree patterns that contain prefix:/struct-config/action-mappings/action/.

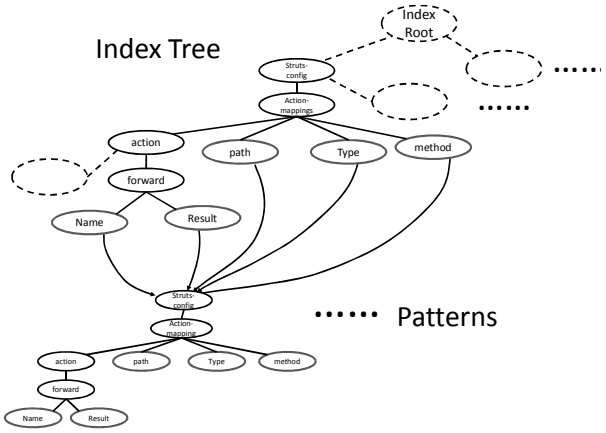


Figure 6. An index structure example

A trie or prefix tree is a common multi-way tree structure for indexing strings over an alphabet. All the descendants of a node in the trie have the common prefix of the string associated with that node, and the root is associated with the empty string. In the context of XMLSnippet application, each node is associated with an element or attribute. All the descendants of a node in our trie index share the common prefix of the element/attribute sequence. Given the trie of all prefixes, each prefix can be retrieved in $O(p)$ time, where $p$ is the length of the prefix (i.e., the number of elements/attributes in the prefix).

The detailed procedure to build a trie to index paths in frequent tree patterns is given in Algorithm *ConstructIndex*. For each root-to-leaf path $p$ in every frequent pattern tree $t$, *ConstructIndex* adds it to the index structure *(IDXT)* if $p$ cannot be found in *IDXT*. Each node of the trie corresponds to a possible prefix of element/attribute. Hence, when building the trie, for each node in the trie, the tree patterns that contain the prefix corresponding to the node are stored. After the *IDXT* has been constructed, we can calculate the support of each prefix in a bottom-up manner, which is used in *SQ* later.As an example, we illustrate the constructed trie in Fig. 6 (upper part of the figure) for the tree patterns shown in Fig. 4. As shown in Fig. 6, each root-to-leaf path in each tree pattern can be mapped to a node in the trie, such that in the trie each path from the root to a node represents a certain root-to-leaf path of tree patterns.

During the course of editing the XML configuration file, the prefix before the cursor will be fed to the retrieval procedure to get all frequent tree patterns containing the prefix from the above constructed index structure *IDXT*. The detailed retrieval algorithm is given in Algorithm *Retrieve*.

After obtaining a frequent tree pattern, we use function *sample*(*ftp*) to get the set of corresponding samples $s$ for frequent tree pattern *ftp*. Since the relationship between instances and patterns has already been built in the offline mining process, we can directly return samples (instance) for a tree pattern. Sample $s$ is a triple of the form $<x,c,d>$ with $s.x$ denoting sample XML snippets, $s.c$ denoting the sample client source code and $s.d$ denoting the size of $s.x$ (*i.e.,* number of XML nodes). In general, for a pattern, there exist multiple samples. We use $d$ to rank these samples. The smaller $d$ is the simpler the sample is and consequently more interesting to programmers. On the other hand, we use Lucene [12] to build the index on all key words for boosting the retrieval of XML snippets containing the key words as well as corresponding client source code such as .jsp, .java, and etc.

As mentioned before, in the process of building XML trees, we do not keep those attribute values as tree nodes. Instead, we regard these attribute values as search key words which will be used in the *CQ* later as Lucene query keywords. Those attribute values are stored in the form of quadruple $<type,elem,attr,v>$, where *type* denotes the XML document type, *elem* and *attr* denote the element and attribute to which this value belongs, and $v$ denotes the attribute value.

### C. Online Query

To assist coding, an online prompt is necessary for real applications. Hence, we need to efficiently retrieve appropriate edit rules from the pattern database without sacrificing the correctness of the retrieved rules. As mentioned above, there are three major classes of atomic queries that need to be carefully implemented to assist coding: *TQ*, *SQ*, and *CQ*. Once the hotkey is triggered, the XML document under editing and the current cursor location are fed into XMLSnippet to retrieve recommendations by three basic queries.

During the initialization of an XML document for configuration, the prompt request is handled as *TQ* to get the most frequent *ftp* including the root node of XML files of a specific type as the basic fragment. To process *TQ*, the tool first retrieves the *ftp* with the highest frequency for each framework type (*ft*) in code repositories, and then assembles them in a cascading interactive list for the programmer to choose from by moving the cursor (illustrated in Fig. 2(a)).

After the initiation of XML, the programmer spends most of his time on filling in the detailed content of the XML configuration to control the logic of the project under

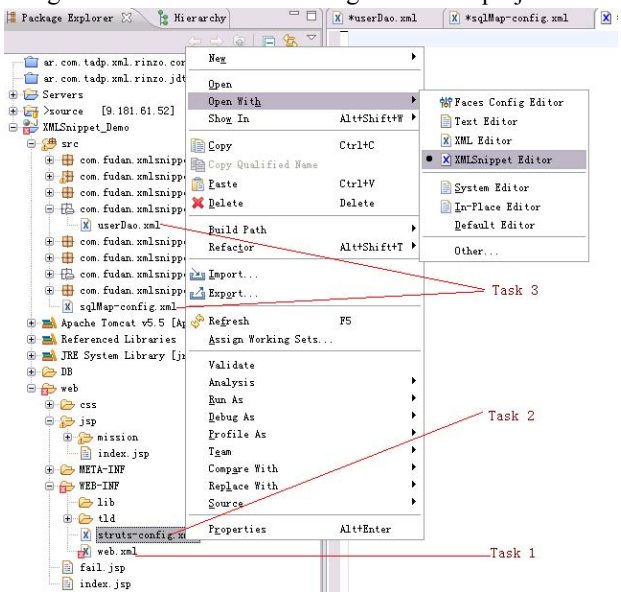


Figure 7. XMLSnippet Tool UI

development. Again, the tool calls *SQ* to generate the structural XML fragments automatically. *SQ* first gets the prefix *p* of the current cursor *c* by function *prefix(c, xml)*. For example, the *prefix* of the current cursor is /struct-config/ in Fig. 2(b). Then, *SQ* generates all the *ftp* containing *p* through function *ftp*(*p*). Let *t* be the set of these *ftp*. Finally, *SQ* sorts *t* by the frequency of each *ftp* into descending order. The *ftp* is then restored to XMLSnippet before its presentation to the programmer. The tool encapsulates the output list as a cascading interactive menu list for the programmer to choose from (see Fig. 2(b)).

Besides *TQ* and *SQ*, the tool also helps the programmer to input the value of specific attributes or elements by *CQ*. *CQ* does not directly provide the recommendation of candidate values. Instead, by *sample(ftp)* function *CQ* provides the sample XML snippets that are similar to the *ftp* under editing, together with corresponding source code that may indicate the value, illustrated in Fig. 2(c). By comparing the highlighted part in both source code and XML fragment sample, the programmer could navigate the project with similar context in non-XML code part of target application.

## IV. EVALUATION

In this section, we conduct a series of experiments to answer the following questions in the evolution:

- Q1: Generality. Could the tool be applied to different frameworks?
- Q2: Effectiveness. Could the tool produce reusable snippets to shorten the programming cycle?
- Q3: Usability. Is the tool useful for programmers with different skill levels?
- Q4: Efficiency. Could the tool mine frequent tree patterns, build index and propose reusable snippets efficiently?

### A. Setup

We delivered XMLSnippet as an Eclipse plug-in that can be invoked from within the Eclipse and evaluate our tool on a semi-manufactured Java web project. The test project is a simple book store management project including the basic CRUD (Create, Retrieval, Update, and Delete) functions. Table 1 shows the basic statistics for the target application. In order to evaluate the benefit of XMLSnippet, except for the XML configuration codes, all other JSP, Java Code, CSS files are correctly provided. Hence, programmers only need to build XML configuration files. As Fig. 7 illustrated, he can click right mouse and select open with "XMLSnippet Editor" for assistance. We designed three tasks for different frameworks involved in the evaluation. These tasks are designed for the same target applications with increasing difficulty. The three tasks are initializing web.xml for Web applications (Task 1), constructing configuration file of Structs for MVC (Task 2), and constructing iBATIS configuration file (Task 3), respectively. The experimental results of each task will be discussed one by one in the following text.

There are three typical types of programmers' skill and background. 1) Without skills of the frameworks. In this case, the programmer tends to search for samples from the Web and learns to use these frameworks from scratch. 2) Without framework-relevant skills, but may have some instructors from professional training. 3) Already experienced for those frameworks.

TABLE I. STATISTICS OF THE TEST APPLICATION

| Target Application | *Java Class* | *JSP* | *CSS* | *XML* |
|---|---|---|---|---|
| #file / #line | 18/582 | 7/420 | 1/84 | 4/153 |

TABLE II. GROUPS OF PROGRAMMERS UNDER EVALUATION.

| Type | *Group ID* | *Experienced programmers* | *Training of related frameworks* | *XMLSnippet as assistant tool* |
|---|---|---|---|---|
| T1 | 000 | No | No | No |
| | 001 | No | No | Yes |
| T2 | 010 | No | Yes | No |
| | 011 | No | Yes | Yes |
| T3 | 100 | Yes | No | No |
| | 101 | Yes | No | Yes |

We invited above three types of programmers to participate in the evaluation for programmers as shown in Table 2. The first bit of group numbered from left to right of the group id indicates whether the programmer is experienced of Web application, Struts, iBATIS. The second bit indicates whether the programmer will get the training on related frameworks used in the target application. The third bit indicates whether the programmer will use the tool for assistance. Thus totally 6 groups are involved.

Notice that, we give all the programmers the training to help them understanding the test semi-manufactured project. In addition we also give those groups using the tool preliminary training about our tool.

Two rounds of evaluations were performed on all the programmers. In the first round, they work under a three-hour time limitation. If they could not complete it, we allow them to continue the tasks without time limit in the second round one month later.

### B. Experiment results

The time spent by each programmer for each task and corresponding achieved functionality (measured by percentage) is shown in Table 3, where Task1 (1) indicates the result for Task 1 in the first round and Task 2 (1+2) indicates the accumulated result for Task 2 in the two rounds. N/A in the table indicates that the data is not available.

#### 1) Task 1: Initialize Web.xml for the target application

The first task is to initialize the web.xml for the target application. The functionality of this task is measured by whether the resulting application could be loaded and launched by the Tomcat server and the default page could be accessed by Internet Explorer. This is the easiest task in our evaluation. All the programmers successfully finished this task except for programmer 1. Compared with group 000, the programmers of group 001 finished Task 1 in 5 and 20 minutes, respectively. In contrast, the programmers in group 010 who accepted the training but accomplished the task without the help of our tool spent more time to accomplish Task 1. Because web.xml has very compact patterns, the programmers can input all the attribute values of session time out, URL access path, etc with only 1 *TQ*, 2 *SQ*s, and

TABLE III. EVALUATION RESULTS. THE RESULT SHOWS THAT FOR ALL THREE DIFFERENT TASKS USING OUR TOOL CAN SIGNIFICANTLY REDUCE PRODUCTION TIME FOR PROGRAMMERS WITH DIFFERENT PROGRAMMING SKILLS

| Time(Minutes) /Percentage(%) | Nonexperienced Programmers | | | | | | | | Experienced Programmers | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *[000]* | | *[001]* | | *[010]* | | *[011]* | | *[100]* | | *[101]* | |
| | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* | *11* | *12* |
| Task 1(1) | 180/100 | 85/100 | 5/100 | 20/100 | 37/100 | 15/100 | 2/100 | 3/100 | 8/100 | 9/100 | 2/100 | 3/100 |
| Task 2(1) | N/A | N/A | 45/40 | 50/60 | 71/90 | 60/80 | 22/90 | 63/100 | 57/100 | 65/100 | 11/100 | 19/100 |
| Task 3(1) | N/A | N/A | 130/0 | 100/0 | 72/0 | 95/50 | 120/80 | 45/100 | 71/100 | 63/100 | 26/100 | 21/100 |
| Task 2(1+2) | N/A | N/A | 68/100 | 65/100 | 101/100 | 72/100 | 32/100 | 45/100 | 71/100 | 63/100 | 26/100 | 21/100 |
| Task 3(1+1) | N/A | N/A | 190/100 | 140/100 | 162/100 | 125/100 | 150/100 | 45/100 | 71/100 | 63/100 | 26/100 | 21/100 |

limited *CQ*. Thus much time to type in the structure elements was saved by using our tool. The group of 011 performed better than the 010 groups as they had accepted the training and did not need to web search for the basic knowledge of web.xml. Even without the help of the tool, the professional programmers in the 100 group finished the job in 8 and 9 minutes, respectively. However, the professional programmers in group 101 used less time than those in group 100 due to the convenience of our tool.

*2) Task 2: Construct the Struts_Config.xml*

In this task, there are about six jumping points for form request for CRUD operation of the application. Both programmer 1 and programmer 2 failed to accomplish this task again. With the help of our tool programmers 3 and 4 spent 45 and 50 minutes on Task 2, respectively. But they only finished 40% and 60% functionality, respectively. In contrast, the programmers in group 010 spent more time to learn Struts from the Web and finished 90% and 80% within 71 and 60 minutes, respectively. Although the tool could save time, it could not replace the self-study for the freshmen of a certain framework. In contrast, the programmers of group 011 achieved the best result among inexperienced programmers, as expected. They spent less time and gained better correctness than those without the help of the tool in group 010. Due to the benefits of our tool, the experienced programmers in group 101 perform best. They accomplished 100% functionality with only 11 and 19 minutes for Task 2, respectively. Note that the time-saving was quite significant since group 101 only consumed about 1/4 time spent by group 100.

We further analyze the statistics in the row Task 2(1+2). Programmer 7 in group 011 performs best, since he accepts framework training and programs with the help of our tool. When we compare group 001 and 010, we find that group 001 used less accumulated time to finish Task 2 although they did worse in the first round. We interviewed programmers 5 and 6 and got an interesting finding: although they did better in the first round, some of the knowledge obtained by the training was lost by the second round one month later. Thus they had to resort to the Web again based on their remaining memorized knowledge of the training in the first round. In contrast, although programmers 3 and 4 expended much time on searching on the Web in the first round, they identified the right tree patterns suggested by *SQ* and the knowledge to use the result provided by *CQ*. Because the knowledge of the tool was retained, the only thing they needed to do was to raise the query again and finished the rest of the XML fragment smoothly.

*3) Task 3: Construct the iBATIS configuration xml files*

The programmers needed to finish the dao.xml and SQLMap-config.xml of the target applications in Task 3. For Task 3, among the inexperienced programmers, only three of them made a certain degree of progress in round 1. However, group 011 did much better than programmers without the help of our tool and programmer 8 even finished all the functionality of Task 3 within the time limit. Programmers 3, 4 and 5 failed because the structural patterns alone are not sufficient to finish Task 3 and SQL programming was necessary to fill the values of SQL element in iBATIS. The failure of programmers 3, 4 and 5 could be attributed to the fact that they are not familiar with SQL statements. Although *CQ* could provide SQL statement samples, SQL recommendation in general is beyond the scope of this paper. On the other hand, the experienced programmers finished all the tasks in time. As in Task 2, group 101 finished the job in much less time than group 100 without the assistance of the tool, because group 100 still needed to spend time to search on the Web to find reusable structural patterns of XML configuration samples to finish the job. It was interesting to find that programmers 3 and 4 used less extra time to finish Task 3 (the extra time can be calculated by Task 3 (1+2)-Task 3(1)) than programmers 5 and 6 even though they performed worse in the first round. That is also because the knowledge of the tool was retained and the only thing they needed to do was to raise the query again and finished the rest of the XML fragment smoothly.

### *C. Performance results*

Public available code from Web or CD attached to books about framework programming construct the application repository. The offline processing statistics are shown in Table 4. For the repository we used in the evaluation, the mining time is 59413 ms, while the indexing time is 29281 ms. Since mining and indexing are executed offline, such performances are acceptable. If we use the method mentioned in [11] directly without our improvement, the mining time is 94164 ms. The difference between the two running times shows the efficiency improvement.

There are in total 137 closed frequent tree patterns produced by our improved mining algorithm. Through manually checking by experienced programmers, we find that all these patterns are novel and meaningful. As comparison, we also give the statistics of the closed frequent tree patterns mined by the original algorithm proposed [11] By manually checking, we found those additional closed patterns are useless mainly because they are not patterns that

TABLE IV. OFFLINE PROCESSING STATISTICS.

| Mining Statistics | *Web.xml* | *Struts* | *iBATIS* | *Spring* | *Hibernate* |
|---|---|---|---|---|---|
| #XML config. File in the repository | 7 | 20 | 110 | 60 | 25 |
| #Closed frequent tree patterns by XMLSnippet | 2 | 12 | 78 | 29 | 16 |
| #Closed frequent tree patterns by [11] | 5 | 27 | 103 | 43 | 36 |
| Closed Tree Pattern mining time cost | | | | 59413 ms | |
| Indexing time | | | | 29281 ms | |

TABLE V. ONLINE QUERY RESPONSE TIME STATISTICS.

| Average Query response time | *Time (ms)* |
|---|---|
| Average TQ response time | 1.0 |
| Average SQ response time | 59.2 |
| Average CQ response time | 338.5 |

start from the root element of a framework or they contain redundant identical internal element nodes that are meaningless in the context of XMLSnippet.

With this tool, the programmers could get code recommendation results by triggering the "Alt+/" hotkey to raise corresponding queries automatically. The average *TQ* response time is only 1 ms. *SQ* needs more time, 59.2 ms, as the tool will list all the XML snippets with the same prefix of the current cursor for the XML docs under editing. *CQ* consumes the most time, that is 338.5 ms. It is obvious that even the worst query time of *CQ* is within the scope of a tolerable and even efficient real time query system for comfortable user experience.

### D. Discussion

#### 1) Threats to validity

Although we tried to find the programmers with skills at the same level and organize them into the same group to evaluate our tool, it is still possible for the existence of evaluation bias caused by the individual difference. That is why the programmers in the same group exhibited different behaviors and different efficiencies to finish the same task. Even so, the difference among groups guided by our tool and the group without our tool is comfortably obvious. Hence, the benefits brought by our tool are still convincing.

#### 2) Lesson and Learning

**Lesson**: (1) From the evaluation, we found that the programmers with ad-hoc training performed better than those only with the tool in the first round. That is to say for some urgent development task, training is necessary. But the persistent and repeatable snippet proposed by our tool is still helpful for the programmers to recall the knowledge learned from training and save time to type these snippets. (2) According to the interview of the programmers, although the tool could propose candidate samples by *CQ*, sometimes it might be challenging for them to refer to the right sample or input the right value since attribute values depend on application context.

**Learning**: Next, we present our leaning from the evaluation results. These learning answers the questions proposed in the beginning of the experimental section.

*a) Answer to Q1*. Our tool promises to be valuable in other frameworks based on XML due to the following reasons. 1) The frameworks tested with promising result are the most representative ones. 2) The underlying mining solution and query solution are general enough, allowing them to be directly applied on other frameworks.

*b) Answer to Q2*. For the programmers with the same programming skill, the tool save them a lot of time to build frame-based applications by applying reusable XML snippets and refer to samples proposed by the tool compared to those without the tool as table 3 shows.

*c) Answer to Q3*. The results show for programmers with different skills, including beginners, the programmers under training, and the experienced programmers, our tool is consistently effective to help build the XML configuration files.

*d) Answer to Q4*. The results from table 4 and table 5 shows the tool is efficient enough to mine application repository, build the index and recommend XML Snippets.

In summary, our tool characterized by generality, effectiveness, usability and efficiency should be a considerable complementary tool for the programmers by self-study/training and also be helpful for those experienced programmers to build framework-based applications relying on XML configuration files.

## V. RELATED WORK

We first survey related works about code reuse. Mandelin et al. developed Prospector [1], which accepts queries in the form of a pair ($T_{in}$, $T_{out}$), where $T_{in}$ and $T_{out}$ are class types, and suggests solutions by traversing all paths among types of API signatures between $T_{in}$ and $T_{out}$. PR-Miner [2] uses frequent item set mining to extract implicit programming rules in large C code bases to detect violations. DynaMine [4] uses association rule mining to extract simple rules from version histories for Java code and detects rule violations. SpotWeb [7] detects framework hotspots and coldspots via mining open source code on the Web. MAPO [8] mines and recommends API usage patterns just of Programming Language (Java) level patterns. CAR-Miner[9] mines exception-handling rules in the form of sequence association rules. All these previous approaches mined simple association rules, which are not sufficient to characterize rules in framework logics in the form of XML. In contrast, our approach can mine more complex rules in the form of frequent tree patterns to handle XML configuration files.

Strathcona [3] compares the context of the code under development with the code samples in the example repositories, and recommends relevant examples. XSnippet [5] suggests relevant code snippets for the object instantiation task at hand. Unlike these works, XMLSnippet recommends the XML fragment, by mining closed frequent through three types of queries, to assist coding in framework-based applications. PARSEWeb [6] interacts with a code

search engine to gather relevant code samples and performs static analysis over the gathered samples to extract required sequences. But the search result obtained by submitting XML element names as the key words to the search engine is always incomplete when building the complex tree patterns for coding assistance in framework-based applications [6].

Eclipse XML Editor [20] recommends XML snippets by directly generating sub elements/attributes of a certain element based on the DTD[19] schema of the XML files. As we have pointed out in previous sections that DTD based recommending returned sub elements/attribute without discrimination and still need resort to expertise to build proper subtrees, which make it not so efficient compared to XMLSnippet. The industry field also contributes tool like the XML Editor in Visual Studio 2010 [18] to generate XML snippets from an XML Schema definition language schema. Compared to it, XMLSnippet is more flexible, efficient and effective. First, when editing an XML file, one has to bind a schema file to the XML file by clicking and selecting schema file in XSD Schema dialog box manually. In XMLSnippet, the schema is determined dynamically according to both the XML file edited so far and user's selection. Secondly, the feature offered by Visual Studio is only available on elements and the current element in editor must be empty with no attributes. As mentioned earlier, XMLSnippet can not only be applied on elements/attributes but also provide suggestions on attribute values. Thirdly, the schema files used in Visual Studio XML editor are generated beforehand in accordance with their applications. On the contrary, XMLSnippet generates tree patterns by data mining technology, which saves much time and effort and guarantees the quality of recommended snippets since these snippets are frequently used in real applications.

## VI. SUMMARY AND FUTURE WORKS

In this paper, we propose a method to recommend reusable XML snippets to help programmers configure XML files. Some key techniques such as close frequent tree mining, prefix tree based indexing, and three types of queries including *TQ*, *SQ*, and *CQ* are proposed. We have implemented the proposed method in a system XMLSnippet. Evaluation results show that our tool can actually propose relevant reusable XML snippets for both inexperienced and experienced programmers, which significantly improves their efficiency in building framework-based applications.

## ACKNOWLEDGMENT

This work was supported by National Natural Science Foundation of China under No.61003001, No.61170006. Specialized Research Fund for the Doctoral Program of Higher Education No. 20100071120032; Key Program of National Natural Science Foundation of China under No. 61033010; National Science and Technology Major Project of the Ministry of Science and Technology of China under grant No.2010ZX01042-003-004.

## REFERENCES

[1] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. Proc. ACM SIGPLAN conference on Programming language design and implementatation(PLDI 05), ACM, Jun. 2005, pp. 48–61, doi: 10.1145/1065010.1065018.

[2] Z.M. Li and Y.Y. Zhou, PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. Proc. the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE 13), ACM, Sept. 2005, pp. 306–315, doi: 10.1145/1081706.1081755.

[3] R. Holmes and G. Murphy. Using structural context to recommend source code examples. Proc. the 27th international conference on Software engineering (ICSE 05), ACM, 2005, pp. 117-125, doi: 10.1145/1062455.1062491.

[4] V. B. Livshits and T. Zimmermann. DynaMine: Finding common error patterns by mining software revision histories. Proc. Proc. the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE 13), ACM, Sept. 2005, pp. 296–305, doi: 10.1145/1095430.1081754.

[5] N. Sahavechaphan and K. Claypool. XSnippet: Mining for sample code. Proc. the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion (OOPSLA 07), ACM, 2007, pp. 413–430,doi: 10.1145/1297846.1297946.

[6] S. Thummalapenta and T. Xie. PARSEWeb: A Programmer Assistant for Reusing Open Source Code on the Web. Proc. the twenty-second IEEE/ACM international conference on Automated software engineering (ASE 07), ACM, 2007, pp. 204–213, doi: 10.1145/1321631.1321663.

[7] S. Thummalapenta and T. Xie. SpotWeb: Detecting framework hotspots and coldspots via mining open source code on the web. Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 08), IEEE Computer Society, Sept. 2007, pp. 327–336, dot: 10.1109/ASE.2008.43.

[8] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and Recommending API Usage Patterns. Proc. the 23rd European Conference on ECOOP 2009(ECOOP 09), Springer-Verlag, 2009, pp. 318–343, doi: 10.1007/978-3-642-03013-0_15.

[9] S. Thummalapenta and T. Xie. Mining Exception-Handling Rules as Sequence Association Rules. Proc. the 31st International Conference on Software Engineering (ICSE 09), IEEE Computer Society, 2009, pp. 496–506, doi: 10.1109/ICSE.2009.5070548.

[10] M.J. Zaki. Efficiently Mining Frequent Trees in a Forest, Proc. the eighth ACM SIGKDD international conference on Knowledge discovery and data mining (KDD 02), ACM, 2002, pp. 1021 – 1035, doi: 10.1145/775047.775058.

[11] Y. Chi, Y. Xia, Y. Yang, and R. R. Muntz. Mining Closed and Maximal Frequent Subtrees from Databases of Labeled Rooted Trees. Knowledge and Data Engineering, IEEE Transactions on , vol.17, Feb. 2005, pp. 190-202, doi: 10.1109/TKDE.2005.30.

[12] http://lucene.apache.org/

[13] Apache Struts project. http://struts.apache.org/.

[14] The iBATIS Date Mapper. http://ibatis.apache.org/.

[15] http://www.jwebhosting.net/jpetstore/.

[16] Prefix Tree. Wikipedia. Last modified 5 Sep 2011. http://en.wikipedia.org/wiki/Trie.

[17] Eclipse. www.eclipse.org/

[18] http://msdn.microsoft.com/en-us/library/ms255822%28v=VS.100%29.aspx

[19] http://www.w3schools.com/dtd/default.asp

[20] http://editorxml.sourceforge.net/