Information and Knowledge Retrieval within Software Projects and their Graphical Representation for Collaborative Programming

Ivan Polášek¹, Ivan Ruttkay-Nedecký², Peter Ruttkay-Nedecký², Tomáš Tóth², Andrej Černík², Peter Dušek²

¹ Faculty of Informatics and Information Technology, Slovak University of Technology in Bratislava, Slovakia, e_mail: polasek@fiit.stuba.sk
² Gratex International, a.s., Bratislava, Slovakia, www.gratex.com, e_emails: peter.ruttkay-nedecky@gratex.com, ivan.ruttkay-nedecky@gratex.com, tomas.toth@gratex.com, andyc@gratex.com, pdusek@gratex.com

Abstract: The paper proposes parsing and visualization of the source code, recognition of the authors and users, structures and types of the source code to enhance collaborative programming in medium and large software projects.

Keywords: collaborative programming, visualization, recognition, authors, users, source code type, code marking

1 Introduction

This paper describes our approach of the concept for collaborative programming. It supports developers especially in medium and large teams, where it is hard to understand the large structures or extensive content and difficult to find code authors and particular people who work at a concrete modules to ask them to help newcomers and other colleagues.

We decided to create intelligent environment, using data mining methods for highlighting interconnection and graphical representation of the information and knowledge to start source code reusing, corrections or implementing new features. Our colleagues from Vision & Graphics Group¹ analysed similar software visualization methods in 3D space [12].

¹ vgg.fiit.stuba.sk
2 Parsing and Visualization

We are using abstract syntax tree (AST) representation for extracting information from source code [1][5][6]. Information extraction is done by recursive scanning of the provided AST. Required properties are read for each node and then the scanning process continues on its children:

\[
\text{ProcessAST(IN: nodes)} \\
\quad \text{for node in nodes} \\
\quad \quad \text{ReadNodeData(node)} \\
\quad \quad \text{ProcessAST(node.Children)}
\]

For visualization we are using our own graph visualization engine developed on top of graphical engine Ogre3D [1] and the Fruchterman-Reingold [3] force-directed layout algorithm (see Figure 1). Key concept of the Fruchterman-Reingold is minimizing the energy of the system by moving the nodes and changing the forces between them.

![Figure 1](image1.png)

Visualization of the source code using Fruchterman-Reingold algorithm

We are also developing our own graph representations, which are more semantically oriented than a general force-directed algorithm. Snapshots of these graphs are provided in following figure.

![Figure 2](image2.png)

Semantically oriented source code visualization
3 Tier Recognition

Purpose of tier recognition is to identify tiers in three-tier based systems. These systems are composed of [7] presentation tier (presents results of computation to system users and collects user input), application processing tier (provides application specific functionality) and data management tier (manages the system databases).

The recognition is performed on multiple levels - classes, namespaces, projects and the others. Each code entity (class, method, field, project, etc.) is described by its child code entities in the sub level (for example a class is described by its methods), but also the code entity itself describes its child code entities (for example methods in a data tier class will most likely belong to data tier).

In our approach, we determine tiers in several ways, which can be combined to achieve more precise results.

3.1 Keywords

In source code it is common that the name of a type (class, interface, structure, enumeration) describes its purpose. For instance “DbCommand” at first glance tells about the type, that it represents some sort of database command encapsulation. In our method, we are using this common practise to identify specific keywords in names of code identifiers with intention to recognise code tier of a given type. For example we may be able to associate the mentioned type “DbCommand” with data tier, because of keyword “Db” in its name.

3.1.1 Identifiers

Our method is using multiple identifier types as a source for search. As can be expected, the primary identifier type is name of the type. Additional identifier types are name of the base type, namespace name and, in a case of integrated development environment that supports grouping of source code into projects, also project name.

3.1.2 Keywords Dictionary

This method requires a set of known keywords and their tier assignments as an input. We are calling this set a keyword dictionary. It is possible for one keyword to be used in multiple tiers. Therefore it is essential to perceive each tier assignment of a keyword as a rate, which defines, how much the keyword is specific for a given tier. Method requires that the sum of tier assignment rates is equal to one for each keyword in the dictionary. For our test, we have used only a small and manually constructed dictionary, but we are planning to use an
automatic keyword extraction to create nontrivial dictionary. Example dictionary is presented in the following table.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Data</th>
<th>App</th>
<th>Presentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>0.90</td>
<td>0.10</td>
<td>0.00</td>
</tr>
<tr>
<td>Db</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Table</td>
<td>0.60</td>
<td>0.00</td>
<td>0.40</td>
</tr>
<tr>
<td>Workflow</td>
<td>0.10</td>
<td>0.80</td>
<td>0.10</td>
</tr>
<tr>
<td>Form</td>
<td>0.00</td>
<td>0.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Control</td>
<td>0.00</td>
<td>0.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

3.1.3 Word Extraction

Word extraction is the central part of the method. Its task is to divide an identifier’s name into separate words, which can be compared with keywords in the dictionary. Extraction result can be seen in the following picture.

![Word extraction](image)

For the purpose of word extraction we have created regular expressions, which define the split points in the identifier’s name.

- **Class name** - `[_<>.,]d+[?<=^[^A-Z]](?=[A-Z])(?=[A-Z][a-z])`
- **Namespace name** - `[_]d+[?<=^[^A-Z]](?=[A-Z])(?=[A-Z][a-z])`
- **Project Name** - `[_]d+[?<=^[^A-Z]](?=[A-Z])(?=[A-Z][a-z])`

3.1.4 Association Rate

After extracting the words from the identifiers, the partial association rate of each identifier type is computed. That means the four association rates given by identified keywords in the type’s name, base type names, type’s namespace and project name. Each partial rate is in range of <0.0, 1.0>. Partial rate is computed using the following pseudocode.

\[
\text{GetTierPartialRate(IN: words, IN: lookupTable, OUT: rate, OUT: baseRate)}
\]

\[\text{baseRate} = 0\]
for word in words :
    if lookupTable.Contains(word) :
        keyword = lookupTable[word]
        baseRate++
        tierRate += keyword.Rate
    if baseRate > 0 :
        rate = tierRate/baseRate
    else :
        rate = 0

As shown in the provided pseudocode, words, that haven’t been successfully matched with any keyword, are not included in the computation. Therefore they are not lowering the resulting rate.

In the next step, these partial rates are merged into final weighted rate. For this purpose each identifier type has been given a weight in range <0.0, 1.0>. It is not required that the sum of these weights is equal to one. Computation of the final weighted rate is presented in the following pseudocode.

GetTierWeightedRate(IN: weights[], IN: unitRates[], IN:baseRates[], OUT: weightedRate)
    weightSum = 0
    for i in [0..3] :
        if baseRates[i] != 0 : weightSum+=weights[i]
    if weightSum == 0 :
        weightedRate = 0
    else:
        for i in [0..3] :
            if baseRates[i] != 0 :
                weightedRate+=unitRates[i]*weights/weightSum

3.1.5 Case Study

In this part we are going to show an application of this method on a very small set of types (Table 2).

<table>
<thead>
<tr>
<th>Type</th>
<th>BaseType</th>
<th>Project</th>
<th>Namespace</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITGetBlobHelper</td>
<td>FTM.DataInterfaces</td>
<td>DataInterfaces.Blob</td>
<td></td>
</tr>
<tr>
<td>Graph3dControl</td>
<td>UserControl</td>
<td>Graph3d.WinForm</td>
<td></td>
</tr>
<tr>
<td>ITWorkflowHelper</td>
<td>FTM.DataInterfaces</td>
<td>DataInterfaces.Workflow</td>
<td></td>
</tr>
<tr>
<td>ITWorkflowEntity</td>
<td>FTM.DataInterfaces</td>
<td>DataInterfaces.Workflow</td>
<td></td>
</tr>
<tr>
<td>WSFrmDataContext</td>
<td>DataContext</td>
<td>FTM.DataClasses</td>
<td></td>
</tr>
<tr>
<td>IDBHistoricalTable</td>
<td>FTM.DataInterfaces</td>
<td>DataInterfaces.HistTable</td>
<td></td>
</tr>
</tbody>
</table>
As a keyword dictionary we are going to use dictionary from Table 1. Following charts display computed partial rates for each keyword type.

Figure 4
Unit rate assignments (1-by name; 2-by base types; 3-by namespace; 4-by project)

These rates are then composed to a final result using following identifier type weights (Table 3).

Table 3
<table>
<thead>
<tr>
<th>Identifier</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>1</td>
</tr>
<tr>
<td>Base Type Name</td>
<td>0.7</td>
</tr>
<tr>
<td>Namespace</td>
<td>0.6</td>
</tr>
<tr>
<td>Project Name</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Figure 5
Weighted assignment rates

Figure 6 shows an application of this method on a real-life project represented as a *Manhattan graph*. Each bar represents a single type. Each subbar represents a
single method, which height is determined by its code line count. Bar darkness represents the rate of a Db tier assignment.

![Figure 6](image)

Code tiers as a Manhattan graph in our visualization environment

### 3.2 Standard Types

Standard types represent generally known types, which are mostly part of programming languages or frameworks. Source code is created using standard types and types that are recursively created using standard types. This method searches used standard types in source codes and determines implemented tiers using knowledge of relations between standard types and tiers.

Figure 7 shows, how tiers are determined from given type declarations. For each type declaration, several steps are performed. First used standard types are extracted and their namespaces are identified. Using a predefined lookup table, tier ratios of the extracted namespaces are determined.

![Figure 7](image)

Determining tiers by examining used standard types.
Error! Not a valid bookmark self-reference. presents a fragment of a lookup table, mapping standard .Net namespaces to ratios for each of the basic three tiers.

If a namespace is not found in the lookup table, its parent namespace is searched and so on up to the root namespace. If even the root namespace is not found, the given namespace is ignored.

In our work the lookup table is constructed manually, but we are planning to use automated crawling techniques and clustering algorithms to automatically construct the lookup table.

Not only namespaces of standard types must populate the lookup table. Standard types themselves can also be present. Furthermore, identified types and their namespaces can also be placed back to the lookup table and extend the knowledge of the process.

Table 4
A fragment of a lookup table, which maps .Net namespaces to tier ratios

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Presentation tier</th>
<th>Application tier</th>
<th>Data tier</th>
</tr>
</thead>
<tbody>
<tr>
<td>System.ComponentModel</td>
<td>0.4</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>System.Data</td>
<td>0.05</td>
<td>0.05</td>
<td>0.9</td>
</tr>
<tr>
<td>System.DirectoryServices</td>
<td>0.15</td>
<td>0.7</td>
<td>0.15</td>
</tr>
<tr>
<td>System.Drawing</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>System.Globalization</td>
<td>0.6</td>
<td>0.1</td>
<td>0.3</td>
</tr>
<tr>
<td>System.IdentityModel</td>
<td>0.2</td>
<td>0.8</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Final ratios for the examined type are calculated from the extracted ratios.

Figure 8
Example - final ratios of examined types
In our approach, we are calculating an arithmetic average of all extracted ratios for each tier separately. Standard namespaces and their types could be weighted.

For each entity, the ratios are calculated from ratios of each its child entities (for example, namespace and its types).

Figure 8 shows an example result of this method. Tiers were determined for a fragment of a system, where mostly presentation tier is present. Empty rows represent types, which were not determined, because they can belong to any tier.

### 3.3 Project Meta-information

Integrated development environments often store meta-information describing source codes in separate data sources. For example Microsoft Visual Studio stores information about source code files in project and solution files.

In our approach, we are examining Microsoft Visual Studio project and solution files and extracting two kinds of information - project types and project output types.

Microsoft Visual Studio project has one or more project types. Each type describes what framework the project uses, whether it is an installer or an extension and so on. Some frameworks support development of specific tiers – for example Windows Presentation Foundation supports development of presentation tier.

Microsoft Visual Studio projects can produce three kinds of outputs. *Windows application* is an executable file with graphical user interface, which in general implements also presentation tier. *Console application* is also an executable file, but without graphical user interface, therefore we consider it as being without presentation tier. *Class library* is a dynamic link library, which can implement all three tiers.

### 4 Source code users

We acquire information about users and their activities on code entities. This information is processed and presented back to users. Our aim is to improve collaboration between users in software development process. In this chapter, two methods are presented.
4.1 Code entities checkout

Usually revision control systems (RCS) allow users to view, which source code files are currently being edited by which users. This helps to avoid conflicts, when two users edit the same file. But in fact, conflicts mostly occur only when two users edit the same portion of the same file, therefore it is not necessary to lock the whole file.

In our approach, we look at source code not only as a set of source code files, but we go deeply into these files to examine their code entities. We are able to determine, which code entities (not just whole files) are currently being edited by which users. This information could then be used to lock concrete code entities rather that whole files.

This method uses the following algorithm to determine, which code entities are currently being edited:

\[
\text{GetChangedCodeEntities}(\text{OUT: changedCodeEntitySet})
\]

\begin{verbatim}
  changedLocalFiles = GetChangedLocalFiles()
  for localFile in changedLocalFiles
    originalFile = DownloadOriginalVersionFromRCS(localFile)
    changedLineIndices = Compare(originalFile, localFile)
    originalFileAst = ExtractAst(originalFile)
    for lineIndex in changedLineIndices
      changedCodeEntity = originalFileAst.GetCodeEntityAt(lineIndex)
      changedCodeEntitySet.Add(changedCodeEntity)
\end{verbatim}

4.2 User activity on code entities

This method monitors activities of users on individual code entities. Our aim is to determine, with which code entities and how users are working and also to measure, how active users on those code entities are.

In our approach, activity is a general term for anything a user performs with a single code entity – editing, pressing the mouse over it, reaching it in source code etc. Each activity has a value in interval \( <0, 1> \), which represents a measurement, how much the user is active on the code entity. \( 0 \) means no activity and \( 1 \) means maximum activity.

A user can perform more activities on a single code entity at the same time. Figure 9 shows, how the final activity of a user on a single code entity is determined. The final activity is composed of all activities the user performs on that code entity. This includes different activities (typing, pressing the mouse etc.) and the same activities performed at different places (typing to single code entity in two different editors).
First final values for all different activities are calculated. In our approach, for each different activity, this is the maximum value from all places, where this activity is performed – for example typing activity in two editors. At this point we have information, what different activities the user performs on the code entity. In addition, we have a measurement, “how much” the user performs those activities.

The final activity for the code entity is calculated from all different activities. In our approach, we sum up all different activities for the code entity. At this point we have the total value, “how much” the user is active on the code entity.

4.2.1 Activity initial value and cooling of activities

Every activity has a predefined maximum and minimum value. When an activity occurs, its value is maximal. Different activities have different maximum and minimum values depending on their relevance – typing activity is more relevant than mouse pressed activity, because it better reflects the user interest on the code entity.

When an activity is not performed for a period of time, its value starts to decrease down to the minimal value. We call this process cooling down of activities and it express the decreasing interest of the user for the code entity. When the activity is performed again, its value is set to maximum.

4.2.2 Case study

Figure 10 shows our implemented prototype, where three activities are monitored for a single user:

- In viewport – the code entity is reached in a code editor (scrolling etc.)
- Mouse down – the user pressed the mouse over the code entity
- Is typing – the user types into the code entity
The figure also shows, how activities are being cooled down in a period of time.

![Figure 10](image)

*Activities for a single user. Activities are being cooled down.*

## 5 Source Code Authors

### 5.1 Author Characteristics

Author of the source code can be everyone, who creatively changed the content of source code file. We can divide the authors to three basic groups:

- real authors of the content, who modify logical nature of source code (adds, modifies or deletes code entities)
- editors, who modify form of source code record, but not its logical meaning (they are refactoring, sorting elements, formatting code, …)
- reviewers, who can comment code or update code due the newer version of used libraries

By other criterion, we can separate authors to first author of source code part and coauthors.

Alternatively we can reduce every developer as a coauthor, because everyone can considerably modify previous version of source code.

Also there are authors, whose source code part has persisted to last or particular version of source code and authors, whose source code part was deleted or considerably modified by the time.

Another criterion determines, how can be author mapped or bind to given source code entity and how this bindings will be represented:

- Authors of structural and syntactical entities of source code – project, package or module, namespace, class, interface, field, property, method, statement
- Authors of the lines of source code

In the first case, authorship may be related to most nested entities of code, then it is implicitly aggregated and presented at all superior elements. In the second case,
authorship must be binded to individual lines of source code. If the fragment of the code is deleted, in the first case, authorship can be moved to superior element. In the second approach, it is not so explicit.

5.2 Presentation of Authorship

Presentation of authorship can be solved by various approaches. In the context of source code versions (changesets), there can be presented:

- Authorship only in particular version: the authors of the last changes of the source code elements.
- Authorship based on life cycle of source code development to particular changeset.
- Authorship based on the whole life cycle of the source code, not only in the presented changeset if it is not the latest one.

Information about authorship should covered: author, changeset, date of change, type of change, which has been done (add, edit, delete).

5.3 Determination of Authorship

Author is defined by changes, which he made in some version and parts of source code. If we need to evaluate authorship in whole history of code entity, we must match code entities between several versions of source code, which is not trivial due to change of identifiers of code entities and changes of entities positions in AST [5]. Identification of source code entities is given by their similarities or matching [1]. This approach (Figure 11) determines authorship of source code entities in object oriented paradigm, where syntax units can be represented as AST.

![Figure 11](image-url)

Phases of authorship determination based on extraction of source code entity changes
5.3.1 Extraction of source code files from SW solution

Source code is recorded and divided in source code files, which is organized to projects or modules that form SW solution. Files can be added, deleted or modified by time. If we want to evaluate authors whose source code changes persisted or vanished by time, we must extract all source code files that ever existed in lifecycle of SW solution. Another approach is to extract only source code files that were part of SW solution in given version. Because extraction of files from RCS can be extensive, modified algorithm to extract files incrementally as new versions arise, will be more appropriate.

5.3.2 Acquirement of source code in all versions

To acquire history of source code means getting history of source code file(s). This operation is supported by revision control systems managing the source code files. Some scenarios we consider as a problem: renaming of the file, moving of the file in solution structure, creation of the new file with the same name in the place of the file previously removed.

5.3.3 Representation of source code versions as AST code entities

Source code in the revision control systems is represented as a set of the files containing lines of the code. The changes between versions are represented as the changes of the lines. But if we observe the authorship, we look at the source code as a set of syntactical units – source code entities in form of AST [5].

We may only consider more significant syntactical units as the classes and methods because entities are extracted independently without relations between them. Every entity must be uniquely identified (by the path in the tree), having the type of the syntax unit and a range in the lines of code in given version of the source code file.

5.3.4 Comparison of source code versions and representation of differences

Comparison of two versions of source code file is done by Diff function based on solving the longest common subsequence problem [8]. Output of Diff function can be represented as a set of code changes, which can be typed as Add, Delete or Modify. Each of the change has the author, range in lines – added in the later version of source code or deleted in the older one or modified in the both versions.
5.3.5 Mapping of code differences to code entities and representation of source code entity changes

In this phase, we create source code entity changes, which are relations between source code elements and source code changes. Code change falls to code entity, if intersection of their ranges is not the empty set. One change can fall to:

- one code entity in new, old or both compared versions of source code
- multiple code entities in one level of syntax tree (for example two functions)
- multiple code entities in multiple levels of syntax tree (method, class, namespace)

One code change can produce many code entity changes, so it is important to create relations (between change, old and new version of the code element) only between the same type syntax unit and on the same level of AST.

5.3.6 Code Entity Changes Extraction Algorithm

Following algorithm is based on previously described phases. Result is set of changes related to the code elements. On this output, we can evaluate authorship metrics.

\[\text{ExtractCodeEntityChanges(IN: solutionPath, OUT: CodeEntityChanges[])}\]

\[\text{for}\]\[\text{filePath in ParseSolution(solutionPath)}\]
\[\text{oldSrcCodeFile = null}\]
\[\text{for}\]\[\text{newSrcCodeFile in GetHistory(filePath)}\]
\[\text{ast = ParseAst(newSourceCodeFile)}\]
\[\text{newCodeEntities = ReduceCodeEntities(ast.Root, empty)}\]
\[\text{if}\]\[\text{oldSourceCodeFile is not null}\]
\[\text{codeChanges[]} = \text{GetCodeChanges (oldSrcCodeFile, newSourceCodeFile)}\]
\[\text{codeEntityChanges += MapCodeEntityChange(oldCodeEntities, newCodeEntities, codeChanges)}\]
\[\text{oldSrcCodeFile = newSrcCodeFile}\]
\[\text{oldCodeEntities = newCodeEntities}\]

5.4 Authorship metrics

In the process of authorship metrics evaluation, we consider the following aspects. First, we calculate authorship for individual types of authors: real authors (as a coauthors), editors, reviewers. Total authorship of author can be evaluated as their weighted sum, where Real author changes will have the heaviest weight and reviewer changes the lightest.

Next authorship is based on the source code change types, where total authorship is weighted by the sum of the particular types: Added (weight 4), Deleted (1) and
Modified which in fact consists from old version deleted (2) and new version added (1). We can polemize which type of the change tends better to the authorship of the code, so weights for each type (in the brackets above) can be the part of knowledge and can be calibrated. Code changes can be measured in lines of code or syntactic code entities, in absolute number or relative index.

Also it is meaningful to consider evaluate metrics in dependence of time in expression of change sets, where the oldest commit is ranked as least important and the latest commit as the most important.

We are using this characteristics for visualization of source code entities authorship in Tent graph. Part of this graph is shown in Figure 12.

![Tent graph](image)

Figure 12
Authorship of source code entities presented in Tent graph

6 Code Marking System

6.1 Marks visualization

The main idea of marking system is tagging useful metadata to selected parts of source codes. Virtual marks, stored in database, are referenced to source code entities (like projects, classes, methods, etc.) When programmer open the source file in development tool, marking plugin will look to database, load contents of virtual marks corresponding to entities found in code, and show the marks as icons aside corresponding code lines. Clicking mark icon, detail of mark will be shown.
The same plugin allows creating new marks: right mouse-click above selected lines (or above method header, class header, project-file name etc.) defines an entity to be marked, and context menu allows creating mark-content. Mark content with timestamp and author’s name will be stored to database, where it is referenced to existing source-code entity.

6.2 What should be marked

In principle, marking system can tag a virtual mark to anything, which identity may be reliably referenced as logical node of virtual syntax tree: Assuming OOP source codes, we can assign mark to any node of hierarchy `solution -> projects -> classes -> methods`, and `line-range` inside class or method. Assuming `web`, we can mark whole `web-site` (referenced by URL), specified `web-page`, graphic `element` on this page, its `sub-elements` etc. Predicting `documentation`, we can mark whole `documentation file`, its `chapters`, `paragraphs` etc. Marked element, of course, must be referenced in database, and marking system must follow specific logical structure of source files to be marked.

6.3 Type of marks

Each mark is stored as database record, so that mark content may be theoretically anything. We decide to classify marks logically by type of content as follows:

- **note** - free-text comment, annotation, remark, recommendation
- **links** - URL or file references to source, know-how, documentation, etc.
- **keywords** - word-list representing category, destination of code, etc.
- **features** - watched technical features, like `%progress`, etc.
- **rating, warnings** - good/bad rating, warning about mistake etc.
- **authors** - list of code-authors, their kind of participation on code
- **history** - history of entity code updates

All types, except the last two items, are user’s marks, which may be created by programmers. Programmer can identify his own class with keywords, assign some feature to method and/or write some recommendation in foreign code.
Metadata like authors and history is read-only tags of each entity. It helps programmers find quickly, who creates some class, who updates it mostly, how methods of some class evolves.

![Figure 14](image)

Creation of the note mark

### 6.4 Principle of Positioning

The key problem of marking source code is dynamic changes of the code, therefore simple referencing mark-positions to line-numbers is unusable. We use referencing in Abstract Syntax Tree remembering identity of entities (like classes or methods) in historical file versions. It means, that the system remembers that method x occupies lines (10-25) in version 1, lines (12-32) in version 2, etc.

When the mark signs whole method x, it will be simple shown aside class-header line in actual version of file.

If the mark signs line-range or single line inside method x, the positioning is more complicated. System remembers file version, in which the mark was created, and line-position inside (in relation to) original entity. Thanks to entity history, we know position of method x in actual file version (e.g., it occupies lines 12-32) and relative position offset (e.g. lines 2-3). Therefore marked lines are 12-13 in new file version. System must verify if lines (12-13) contains equivalent or fairly similar content as corresponding line-range in original version. If it is so, mark will be shown aside line 12 and if not, mark will not be shown.

The mark’s validity may be optionally limited to code version range. For example, link to class documentation has logically unlimited validity, but warning note usually loses sense when the code was corrected or changed.

### 6.5 Utilization of Code Marking Data

The goal of code marking is saving programmer’s time, when they find patterns [4, [9], description, or same features of existing codes. Information accumulated in marking database allows versatile advanced usage:

- **Searching** of project, classes, methods by **keywords, features, authors**, etc.:
  - Find, if exist some method for sending E-mail among team projects
• Search who writes these E-mail routines, which was already tested, etc.
• Find all classes created by some author
• Find all (un-tested | bad-marked | mistaken...) classes in projects
• See Authors participating on some class or project
• See evolution of class through historical versions

• Comparing some project, classes, methods by features
• Classify some project, classes, methods by keywords, features, authors etc.
• Visualization of summaries, like
  • Good-rated / bad-rate rated codes in projects
  • Safe / unsafe, problematic methods in projects
  • Fast / slow, inhibitory methods in projects
  • Just developed / finished codes in projects
  • Documented | undocumented methods in projects

7 Future work

We plan to complete existing environment with the other methods, namely the content recognition using multiagent systems [15,10,13], swarm intelligence [14], neural networks [11] especially Self-Organizing Maps and pattern or antipattern matching.

Presented algorithm for authorship determination does not solve problem with proper identification of code entities through history in scenarios, where identifies or position in AST of code elements has changed. To solve this deficiency, we must focus to the methods for determination of the source code similarities.

Acknowledgment. This contribution is a partial result of the Research & Development Operational Programme for the project Research of methods for acquisition, analysis and personalized conveying of information and knowledge, ITMS 26240220039, co-funded by the ERDF.

References