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

Ivan Polášek[1], Ivan Ruttkay-Nedecký[2], Peter Ruttkay-Nedecký[2], Tomáš Tóth[2], Andrej Černík[2], Peter Dušek[2]

[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-mails: peter.ruttkay-nedecky@gratex.com, ivan.ruttkay-nedecky@gratex.com, tomas.toth@gratex.com, andyc@gratex.com, pdusek@gratex.com

Abstract: This paper proposes information and knowledge mining in the source code of medium and large enterprise projects. Our methods try to recognize structures and types of source code, identify authors and users to enhance collaborative programming, and support knowledge management in software companies. Developers within and outside the teams can receive and utilize visualized information from the code and apply it to their projects. This new level of aggregated 3D visualization improves refactoring, source code reusing, implementing new features and exchanging knowledge.

Keywords: information and knowledge mining; knowledge management; collaborative programming; visualization; recognition; authors; users; source code type; code tagging

1 Introduction

This paper describes our approach to information and knowledge mining, which aims to support collaborative programming and to help software developers in medium and large teams to understand complicated code structures and extensive content as well as to identify source code authors and concrete people working with existing modules. Accordingly, newcomers as well as other colleagues can reference real source code authors and users more efficiently.

There are some works analyzing collaborative source code development and the impact of individual authors' contributions to the selected characteristics of source code [7]. Authors from the data mining domain often utilize code line oriented SW tools that determine only the latest author of every line [8]. There are also approaches to monitor source code users' work [3] [4] including reading, modifying or showing interest in the source code in some other way.

The determination of the source code topics can be used for purposes of identifying the domain expertise of developers [5]. It is also possible to support program comprehension by identifying common topics in source code [6].

For tagging (keywords for features, authors, patterns/anti-patterns, rating, etc.) and comments, we can use two approaches:

- attach tags using comments inserted directly into source code [1],
- separate tags into external DB or sources [2].

We have studied both approaches and now we prefer the second one.

The contribution of our method is to find new level of aggregated visualization for the identified and interconnected information from the source code for collaborative development.

We have decided to create an intelligent environment that takes advantage of data mining methods. It presents information and knowledge in a 3D graphical form and highlights the interconnections between the information. The key goals of this environment are to support the source code reuse, to support refactoring and the implementing of new features. Knowledge exchange and management is also increased through integration with *Gratex Knowledge Office* and source code tagging.

2 Visualization

For knowledge mining purposes, our method extracts *Abstract Syntax Tree* (AST) from the source code [10] [14] [15] and the resulting graph structure is then visualized with acquired information.

We have developed our own graph visualization engine using the graphical engine Ogre3D [11] and the Fruchterman-Reingold [12] force-directed layout algorithm (see Figure 1).

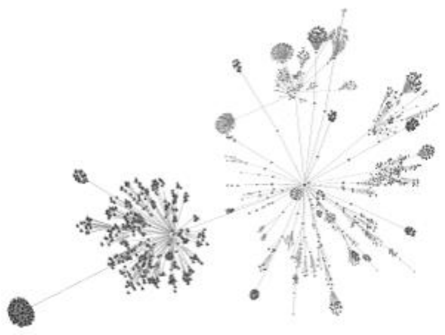


Figure 1
Visualization of the source code using Fruchterman-Reingold algorithm

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 the following Figure 2.

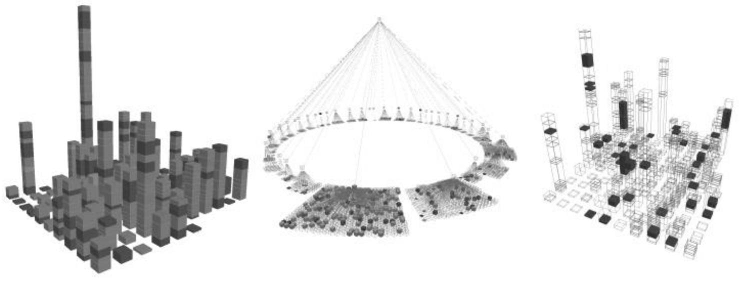


Figure 2
Semantically-oriented source code visualizations

3 Tier Recognition

The purpose of tier recognition is to identify domains in three-tier-based systems. These systems are composed of presentation *tier, application processing tier* and *data management tier* [16]. Each domain is identified by a tier of the associated source code. It is possible to correlate this information through associated entities in AST with other mined information. Therefore, it is possible, for example, to determine the developer's orientation on a given tier. Tier recognition also simplifies navigation in the source code through clustering.

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

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

3.1 Keywords

It is common that the name of the type in source code (class, interface, structure, enumeration) describes its purpose. For instance the name, "DbCommand", from the first glance tells us that the type represents a kind of a database command encapsulation. In our method, we use this common practice to identify specific keywords in names of code identifiers with the intention of recognizing the code tier of a given type.

3.1.1 Identifiers

Our method uses multiple identifier types as the source for searching. As can be expected, the primary identifier is *type name*. Additional identifiers are *base type name*, *namespace* and, in the case of an 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 call this set *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 that defines how much the keyword is specific for a given tier. This method requires that the sum of tier assignment rates is equal to one for each keyword in the dictionary. For our test, we used only a small and manually constructed dictionary, but we are planning to use an automatic keyword extraction to create a nontrivial dictionary. An example of a dictionary is presented in the following table.

Table 1
Example keywords dictionary

Keyword	**Data**	**App**	**Presentation**
Data	0,90	0,10	0,00
Db	1,00	0,00	0,00
Table	0,60	0,00	0,40
Workflow	0,10	0,80	0,10
Control	0,00	0,00	1,00
…	…	…	…

3.1.3 Word Extraction

The task of the word extraction is to divide an identifier's name into separate words that can be compared with keywords in the dictionary. The extraction result can be seen in the following figure.

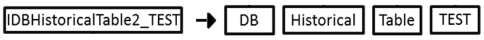


Figure 3
Word extraction

For the purposes of word extraction, we created regular expressions which define the split points in the identifier's name.

- **Class name**- [_<>,]|\d+|(?<=[^A-Z])(?=[A-Z])|(?<=[A-Z])(?=[A-Z][a-z])

- **Interface name**- ^I(?=[A-Z])| [_<>,]|\d+|(?<=[^A-Z])(?=[A-Z])|(?<=[A-Z])(?=[A-Z][a-z])
- **Namespace name**- [_.]|\d+|(?<=[^A-Z])(?=[A-Z])|(?<=[A-Z])(?=[A-Z][a-z])
- **Project Name** - [_.]|\d+|(?<=[^A-Z])(?=[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. Each partial rate is in the range of <0.0, 1.0>. The partial rate is computed using the following pseudocode.

```
GetTierPartialRate(IN: words, IN: lookupTable, OUT: rate, OUT: baseRate)
  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
```

Words that have not been successfully matched with any keyword are not included in the computation. Therefore, they do not lower 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 the range of <0.0, 1.0>. It is not required that the sum of these weights is equal to one. The 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 will demonstrate the application of this method on a very small set of types (Table 2).

Table 2

Example types

Type	**BaseType**	**Project**	**Namespace**
IGetBlobHelper		Frm.DataInterfaces	DataInterfaces.Blob
Graph3dControl	UserControl	Graph3d.WinForm	Graph3d.WinForm
IWorkflowHelper		Frm.DataInterfaces	DataInterfaces.Workflow
IWorkflowEntity		Frm.DataInterfaces	DataInterfaces.Workflow
WSFrmDataContext	DataContext	Frm.DataClasses	DataClasses
IDBHistoricalTable		Frm.DataInterfaces	DataInterfaces.HistTable

We will use the dictionary from Table 1 as the keyword dictionary. The following charts display computed partial rates for each keyword type.

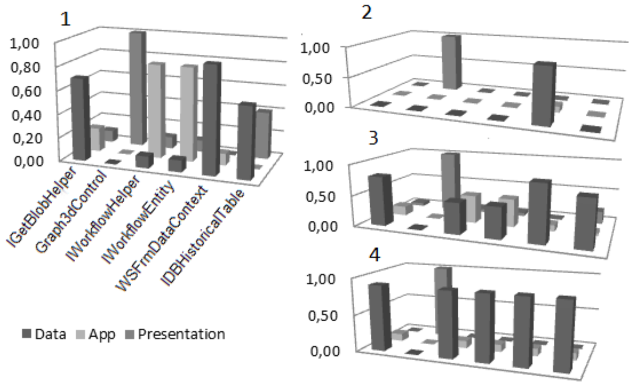


Figure 4

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 the following identifier type weights (Table 3).

Table 3

Example identifier type weights

Identifier	**Weight**
Name	1
Base Type Name	0.7
Namespace	0.6
Project Name	0.6

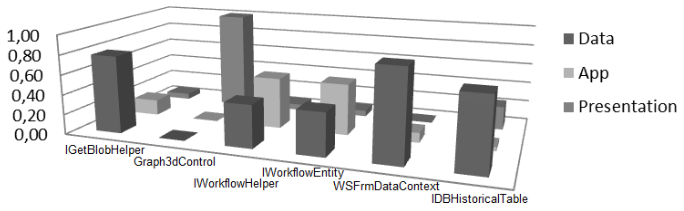


Figure 5

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 sub bar represents a single method, the height of which is determined by its code line count. Bar darkness represents the rate of a DB tier assignment.

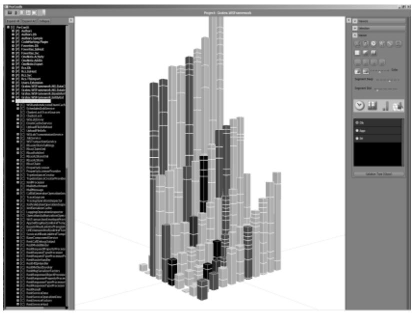


Figure 6

Code tiers as a Manhattan graph in our visualization environment

3.2 Standard Types

Standard types represent generally known types that are usually included in programming languages or frameworks. Source code is created using standard types and types that are recursively created from standard types. This method searches for used standard types in source codes and determines implemented tiers using a knowledge of the relationships 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, the tier ratios of the extracted namespaces are determined.

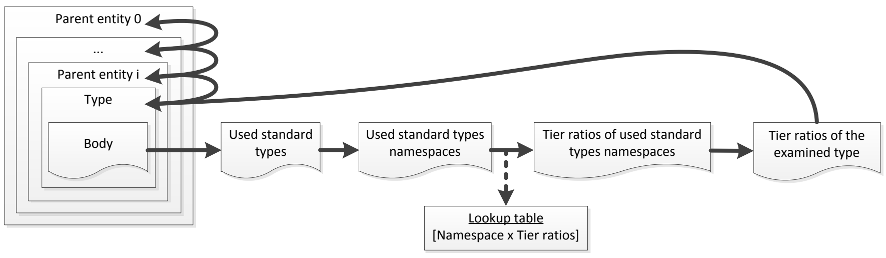

Parent entity 0
...
Parent entity i
Type
Body
Used standard types
Used standard types namespaces
Tier ratios of used standard types namespaces
Tier ratios of the examined type
Lookup table
[Namespace x Tier ratios]

Figure 7

Determining tiers by examining used standard types

Table 4 presents a fragment of a lookup table that maps standard .Net namespaces to ratios for each of the three tiers. The table is constructed manually in our work, but we are planning to use automated crawling techniques and clustering algorithms in the future.

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

Standard types themselves can also be present in the lookup table. Furthermore, identified types and their namespaces can also be placed back to the lookup table and extend the knowledge base of the process.

Table 4

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

Namespace	**Presentation tier**	**Application tier**	**Data tier**
System.ComponentModel	0.4	0.3	0.3
System.Data	0.05	0.05	0.9
System.DirectoryServices	0.15	0.7	0.15
System.Drawing	1.0	0.0	0.0
System.Globalization	0.6	0.1	0.3
System.IdentityModel	0.2	0.8	0.0

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

In our approach we calculate an arithmetic average of all extracted ratios for each tier separately. Standard namespaces and their types could be weighted. Ratios of each entity are calculated from ratios of its child entities (e.g. namespace and its types).

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

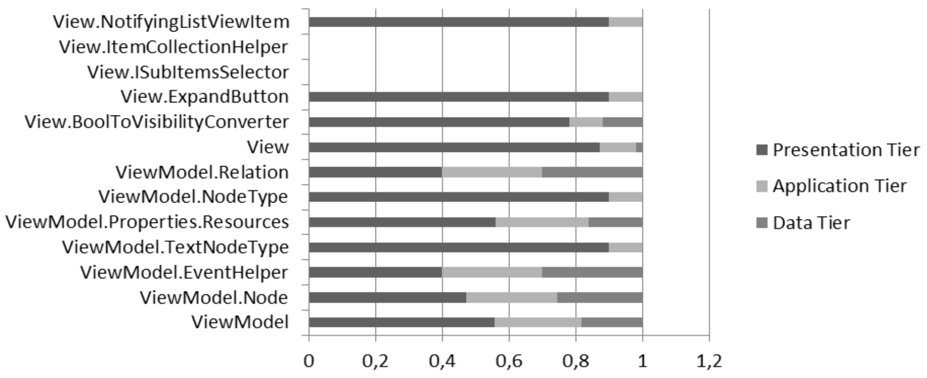


Figure 8

Example - final ratios of examined types

3.3 Project Meta-Information

Integrated development environments often store meta-information describing source codes in separate data sources. For example Microsoft Visual Studio (VS) stores information about source code files in project and solution files. We examine these files and extract two kinds of information - *project types* and *project output types*.

VS 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 the development of specific tiers; for example Windows Presentation Foundation supports the development of a presentation tier.

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

4 Source Code Users

Our method acquires information about users and their activities on code entities. This information is used to model user behavior and represents a source of knowledge for our visualization. Two methods are presented in this chapter.

4.1 Code Entities Checkout

Revision control systems (RCS) usually 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 conflicts mostly occur only when two users edit the same portion of the same file. It is not necessary to lock the whole file.

We look at source code not only as a set of files. We go deeper into these files to examine their code entities. We 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 than whole files.

We use the following algorithm to determine which code entities are currently being edited:

```
GetChangedCodeEntities(OUT:changedCodeEntitySet)
  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)
```

4.2 User Activity on Code Entities

This method monitors the activities of users on individual code entities. Our goal is to determine on which code entities and how users are currently working and also to measure this activity.

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 of how active the user is 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 and the same activities performed in different places (typing into a single code entity in two different editors).

First, final values for all different activities are calculated. For each different activity, this is the maximum value from all places where the activity is performed

– e.g. *typing* in two editors. This represents information: what different activities the user performs on the code entity and a measurement of “how much”.

The final activity for the code entity is calculated from all different activities. 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.

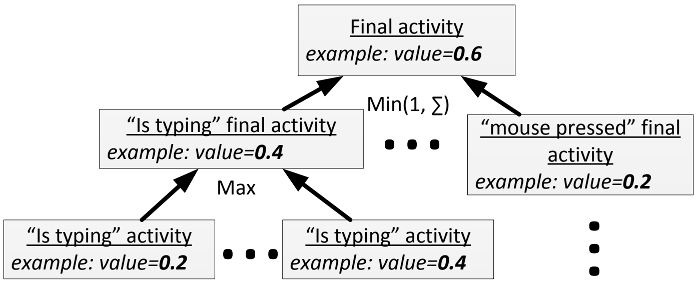


Figure 9
Computing of the final activity of a single user on a single code entity

4.2.1 Activity Initial Value and Cooling of Activities

Every activity has a predefined maximum and minimum value depending on its relevance. When an activity occurs, its value is maximal. When it is not performed for a period of time, its value starts to decrease down to the minimal value. We call this process the *cooling down of activities,* and it expresses the decreasing interest of the user in the code entity.

4.2.2 Case Study

Figure 10 shows a prototype where three activities are monitored for a single user. The cooling down of activities is also shown. These activities are the following:

- 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

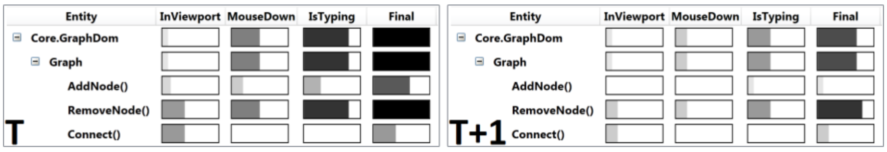


Figure 10
Activities for a single user. Activities are being cooled down.

5 Source Code Authors

To identify the programmer’s skills and quality of his work, we must recognize the authors in the whole source code version history, evaluate their particular contributions and associate them with analyzed code characteristics. We can then identify the responsibilities of each programmer and their impact on team development.

5.1 Author Characteristics

An author of the source code can be anyone who creatively changed the content of the source code file. We can divide the authors into three basic groups:

- real authors of the content, who modify the logical nature of source code (adding, modifying or deleting code entities),
- editors who modify the form of source code record but not its logical meaning (refactoring, sorting elements, formatting code, …),
- reviewers who can comment on the code or an update of the code due to newer version of used libraries.

By a different criterion, we can separate authors to first, being the author of a source code part, and coauthors.

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

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

Another criterion determines how the author can be mapped or bound to a given source code entity and how these 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 lines of a source code.

5.2 Presentation of Authorship

The presentation of authorship can be solved by various approaches. In the context of source code versions (changesets), it can be presented in the following ways:

- Authorship only in a particular version: the authors of the last changes of the source code elements.
- Authorship based on the life cycle of the source code development to a particular changeset.

- Authorship based on the whole life cycle of the source code, not only until the presented changeset, if it is not the latest one.

Information about authorship should cover: author, changeset, date of change, and the type of change that has been done (add, edit, delete).

5.3 Determination of Authorship

The author is defined by changes that the author made in some particular version and in parts of the source code. If we need to evaluate authorship in the whole history of the code entity, we must match code entities between several versions of the source code. This is not trivial due to change of identifiers of code entities and changes of entities positions in AST [14]. The identification of source code entities is given by their similarities or matching [10]. This approach determines the authorship of the source code entities in object oriented paradigm, where syntax units can be represented as AST. It is based on the extraction of source code entity changes and can be divided into several phases (Figure 11):

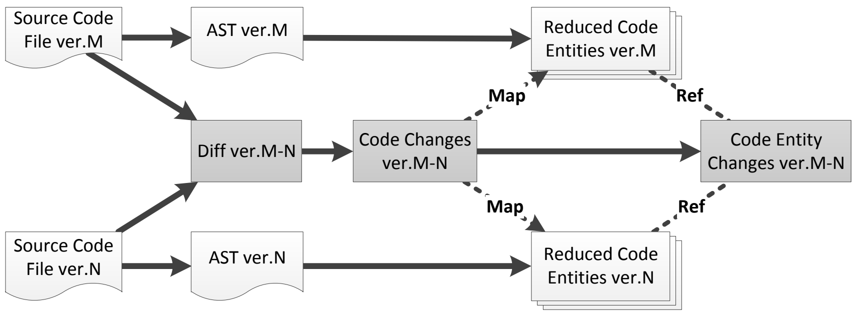


Figure 11
Phases of authorship determination based on extraction of source code entity changes

1) Extraction of source code files from a software solution. Files can be added into, moved within or deleted from the software solution. It is important to identify all source code files contained in the solution throughout its history.

2) Extraction of source code from all files valid for given versions. This is done repetitively for each two adjacent versions and supported by revision control systems. Some scenarios we consider as a problem are as follows: renaming the file, moving the file in a solution structure, creating a new file with the same name instead of the file previously removed.

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

4) Representation of each two adjacent versions as ASTs. Transformation to AST representation is restricted to level of meaningful syntactical units (classes, functions, properties). Lower level content is represented as lines.

5) Comparison of selected source code versions using Diff function based on solving the longest common subsequence problem [17]. The output of Diff function can be represented as a set of code changes, which can be of type Add, Delete or Modify. Each change holds information about the author and the range of affected lines: add in a newer version of source code, delete in older and modify in both versions.

6) 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 the intersection of their ranges is not an empty set. One change can fall to:

   a) one code entity in new, old or both compared versions of source code

   b) to multiple code entities in one level of AST (for example two functions)

   c) to multiple code entities in multiple levels of AST (method, class, namespace)

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

The following algorithm is based on previously described phases. The result is a set of changes related to the code elements. Based on this output, we can evaluate the authorship metrics.

```
ExtractCodeEntityChanges(IN: solutionPath, OUT: CodeEntityChanges[])
for filePath in ParseSolution(solutionPath)
  oldSrcCodeFile = null
  for newSrcCodeFile in GetHistory(filePath)
    ast = ParseAst(newSourceCodeFile)
    newCodeEntities = ReduceCodeEntities(ast.Root, empty)
    if oldSourceCodeFile is not null
      codeChanges[] = GetCodeChanges (oldSrcCodeFile, newSourceCodeFile)
      codeEntityChanges+=MapCodeEntityChange(oldCodeEntities,
newCodeEntities, codeChanges)
    oldSrcCodeFile = newSrcCodeFile
    oldCodeEntities = newCodeEntities
```

5.4 Authorship Metrics

We consider the following aspects in the process of authorship metrics evaluation. First, we calculate authorship for individual types of authors: real authors (as well as coauthors), editors, reviewers. The total authorship of an 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 of an *old version* deleted (2) and a *new version* added (1). We can polemize which type of change tends better to the authorship of the code, and so weights for each type (in the brackets above) can be a part of the knowledge and can be calibrated. Code changes can be measured in lines of code or syntactic code entities, in absolute numbers or relative index.

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

We use these characteristics for visualization of source code entities authorship in *Tent graph*. Part of this graph is shown in Figure 12. Our colleagues from Vision & Graphics Group[1] analysed similar software visualization methods in 3D space [21] [25] without these additional features (authors and code users).

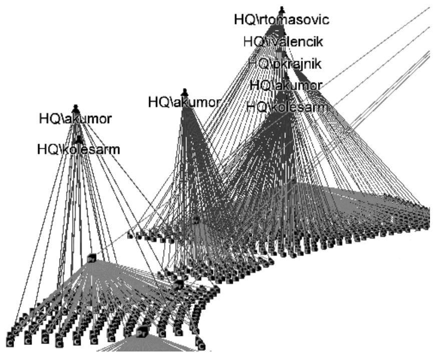


Figure 12
Authorship of source code entities presented in *Tent graph*

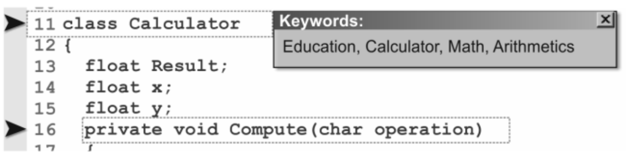
1 vgg.fiit.stuba.sk

6 Information Tags for Knowledge Exchange

Information tags represent a way of binding knowledge with the particular source code. Tags can be created by automated machines or by users. We visualize both types of tags in an integrated development environment as well as in our own graph visualization engine. In this way the knowledge obtained through tags can be presented to programmers, managers and other members of the development team.

6.1 Tags Visualization

Tags are stored in a database and are referenced to source code entities. When a user opens a source code file in a development tool, the corresponding tags are loaded and shown as icons next to corresponding lines. Detailed information of each tag can be shown.



```
class Calculator
{
  float Result;
  float x;
  float y;
  private void Compute(char operation)
```

Keywords:
Education, Calculator, Math, Arithmetics

Figure 13
Displaying tags in a source code editor

The same plugin allows for creating new tags: a right mouse-click above selected lines (or above method header, class header, project file name etc.) defines an entity to be tagged, and context menu allows for creating its content. Tag content with timestamp and author's name will be stored in a database, where it is referenced to an existing source code entity.

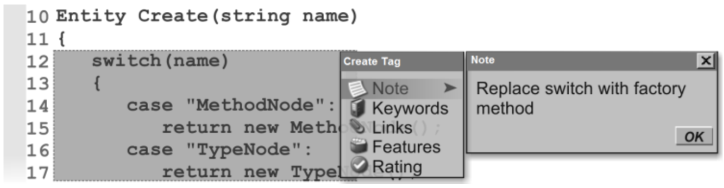
6.2 Type of Tags

We defined the following types of tags:

- *note* – free text comment, annotation, remark, recommendation
- *links* - URL or file references to source, know-how, documentation, etc.
- *keywords* **-** categories, destination of code, etc.
- *features* - technical features like percentage of progress, etc.
- *rating, warnings* – good or 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 for the last two items, are user's marks, which may be created by programmers. A 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 are read-only tags of each entity. It helps programmers to find who creates each class, who updates it mostly, or how the methods of some classes evolve.



```
Entity Create(string name)
{
   switch(name)
   {
      case "MethodNode":
         return new Meth
      case "TypeNode":
         return new Type
```

Replace switch with factory method

Figure 14
Creation of the note mark

6.3 Principle of Positioning

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

When a tag is attached to whole method x, it will be shown next to the class header line in the actual version of the file.

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

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

6.4 Utilization of Code Tagging Data

The goal of code tagging is to save the programmers' time when they find patterns [13] [18], descriptions, or the same features of existing codes. Information accumulated in the marking databases allows versatile advanced usage. For example:

- *Searching* of projects, classes and methods *by keywords, features*, etc.:
  - Find existing methods for sending E-mail in existing source codes
  - Find all classes in projects with low rating
  - See the evolution of the class through historical versions
- *Comparing* projects, classes, methods *by features*
- *Classifying* projects, classes, methods *by keywords, feature,* etc.
- Visualization of *summaries*, like
  - *Good-rated / bad-rate* rated codes in projects
  - *Safe / unsafe or problematic* methods in projects
  - *Fast / slow* methods in projects
  - *Just developed / finished* codes in projects
  - *Documented / undocumented* methods in projects

Future Work

We plan to complete the existing environment with other methods, namely content recognition using multiagent systems [24] [19] [22], swarm intelligence [23], neural networks [20] especially Self-Organizing Maps, and pattern or anti-pattern matching.

In the tier recognition method we are currently using a simple direct method for word extraction from identifiers, but we are planning to apply also other and more sophisticated methods [9].

The presented algorithm for authorship determination does not solve the problem with proper identification of code entities through history in scenarios, where identity or position in AST of code elements has changed. To solve this deficiency, we must focus on methods for the determination of 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

[1] Storey M., Li-Te Ch., Bull I., Rigby P.: Shared Waypoints and Social Tagging to Support Collaboration in Software Development, Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work (CSCW '06) ACM, New York, NY, USA, 195-198, 2006

[2] Guzzi A., Hattori L., Lanza M., Pinzger M.; van Deursen A.: Collective Code Bookmarks for Program Comprehension, Program Comprehension (ICPC) 2011 IEEE 19th International Conference on, Vol., No., pp. 101-110, 22-24 June 2011

[3] Peacock A., Xian Ke, Wilkerson M: Typing Patterns: a Key to User Identification, Security & Privacy, IEEE, Vol. 2, No. 5, pp. 40-47, Sept.-Oct. 2004

[4] Kot, B., Wuensche, B., Grundy, J., Hosking, J.: Information Visualisation Utilising 3D Computer Game Engines Case Study: a Source Code Comprehension Tool, In Proceedings of the 6th ACM SIGCHI New Zealand Chapter's International Conference on Computer-Human Interaction: Making CHI Natural (CHINZ '05) ACM, New York, NY, USA, 2005

[5] Sindhgatta, R.: Identifying Domain Expertise of Developers from Source Code, In Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '08) ACM, New York, NY, USA, 2008

[6] Kuhn, A., Ducasse, S., Gírba, T.: Semantic Clustering: Identifying Topics in Source Code, Information and Software Technology, Volume 49, Issue 3, March 2007

[7] Rahman, F., Devanbu, P.: Ownership, Experience and Defects: a Fine-grained Study of Authorship, In Proceedings of the 33rd International Conference on Software Engineering, ACM, 2011, pp. 491-500

[8] Bird C., Rigby C. P., Barr T. E., Hamilton J. D., German M. D., Devanbu , P.: The Promises and Perils of Mining Git, In Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories (MSR '09) IEEE Computer Society, 2009, pp. 1-10

[9] Enslen, E., Hill, E., Pollock, L., Vijay-Shanker, K.: Mining Source Code to Automatically Split Identifiers for Software Analysis, Mining Software Repositories, 2009, MSR '09, 6th IEEE International Working Conference, pp. 71-80, May 2009

[10] Fluri, B., Wursch, M. Pinzger, M., Gall H.: Change Distilling: Tree Differencing for Fine-grained Source Code Change Extraction, *IEEE Transactions on Software Engineering*, Vol. 33, No. 11 (2007) 725-743

[11] Object-Oriented Graphics Rendering Engine OGRE, www.ogre3d.org (Accessed on 19 August 2011)

[12] Fruchterman, T. M. J., & Reingold, E. M.: Graph Drawing by Force-Directed Placement. Software: Practice and Experience, 21(11) 1991

[13] Beck K.: Implementation Patterns, Addison-Wesley, 2007

[14] Neamtiu I., Foster J., Hicks M.: Understanding Source Code Evolution Using Abstract Syntax Tree Matching, MSR '05 Proceedings of the 2005 International Workshop on Mining Software Repositories, NYC (2005) ACM SIGSOFT Software Engineering Notes, Volume 30, Issue 4, July 2005, NYC

[15] Maletic, J. I., Collard, M. L., Marcus, A.: Source Code Files as Structured Documents, In: 10th IEEE International Workshop on Program Comprehension, Paris (2002)

[16] Sommerville, I: Software Engineering, 9th edition. Addison Wesley, 2011

[17] Eppstein, D.: Longest Common Subsequences. ICS 161: Design and Analysis of Algorithms Lecture notes for February 29, 1996, http://www.ics.uci.edu/~eppstein/161/960229.html (Accessed on 9 March 2012)

[18] Gamma E. et al.: Design Patterns: Elements of Reusable Object-oriented Software, Addison-Wesley, 1995

[19] Tomášek M.: Language for a Distributed System of Mobile Agents, Acta Polytechnica Hungarica, Vol. 8, No. 2, 2011

[20] Lovassy R., Kóczy L. T., Gál L.: Function Approximation Performance of FuzzyNeural Networks, Acta Polytechnica Hungarica, Vol. 6, No. 5, 2009

[21] Šperka M., Kapec P.: Interactive Visualization of Abstract Data, Science & Military, 1/2010, pp. 89-90

[22] Cerná A., Cerný J.: Position Identification of Moving Agents in Networks, Acta Polytechnica Hungarica, Vol. 7, No. 2, 2010

[23] Yilmaz A. E.: Swarm Behavior of the Electromagnetics Community as regards Using Swarm Intelligence in teir Research Studies, Acta Polytechnica Hungarica, Vol. 7, No. 2, 2010

[24] Kelemen J.: Agents from Functional-Computational Perspective, Acta Polytechnica Hungarica, Vol. 3, No. 4, 2006

[25] Kapec P.: Knowledge-based Software Representation, Querying and Visualization, Information Sciences and Technologies, ACM Slovakia, Vol. 3, No. 2, 2011