Acquisition and conveying of information and knowledge in the project of Gratex International and FIIT STU

IVAN POLÁŠEK
FIIT STU, Bratislava, Slovakia

IVAN RUTTKAY-NEDECKÝ
Gratex International, a.s., Bratislava, Slovakia

PETER RUTTKAY-NEDECKÝ
Gratex International, a.s., Bratislava, Slovakia

TOMÁŠ TÓTH
Gratex International, a.s., Bratislava, Slovakia

ANDREJ ČERNÍK
Gratex International, a.s., Bratislava, Slovakia

Abstract. Paper describes first goal of our research and created prototypes: support of collaborative working to increase the productivity of developers. We start with the vision of communications through the multidimensional graph visualizing tasks, projects and their open files or source code elements, their type and contents with patterns and antipatterns, people around working on these files. Everything prepared for our coders we want to reuse also for management, controllers, analysts and testers.

1 Introduction

The main goals of our new research project are: methods for acquisition of information and knowledge (IaK); the model of the user for personalized conveying of IaK; refactoring of the source code and collaborative development; Predictive Project Analysis and integration of the systems. We start with IaK acquisition from source code and collaborative programming.

We want to support developers in software collaborative process to enrich and share their knowledge. Young newcomers need help from authors of the methods and projects to discuss with them the specific features and attributes, if they have to reuse or update concrete part. Also their older colleagues want to know, which files are open or incomplete, who is working on them without unnecessarily disturbing the project leaders or the project members.

Our proposed methods embedded in multidimensional graph try to answer some nontrivial questions in the whole context:

- Which files are complete and incomplete in the project?
- Who is working, on which files (from the point of view of their colleagues and management of the project)?
- Who is the author of the file, if we need to understand it, reuse or change the elements?
- Where is the correct source code (to read and use it as a good sample) and what is incorrect (to avoid it or fix it) according to internal knowledge of colleagues (using marks and rating)?

- What is good or wrong (recognized *patterns* or *anti-patterns*) according to international scientific and professional community?
- To see the set of *Tasks*, *Change requests* and *Bugs* over the source code.
- To detect the type of code (e.g. Presentation/GUI, Business and DB layer),
- To understand contents of the code.
- To see the links to external knowledge sources (professional portals and web pages of vendors, internal and external analytical and technical documents),
- Visualization and animation of the development process for developers and management of the project o see the lacks in their work.

2 Parsing and Visualization of the Source Code for Research and Mining

2.1 Structure of the Source Code

Source code contains various kinds of knowledge scattered in miscellaneous sources. There is always an identifiable structure, which helps us to find, understand and reuse the knowledge we already have.

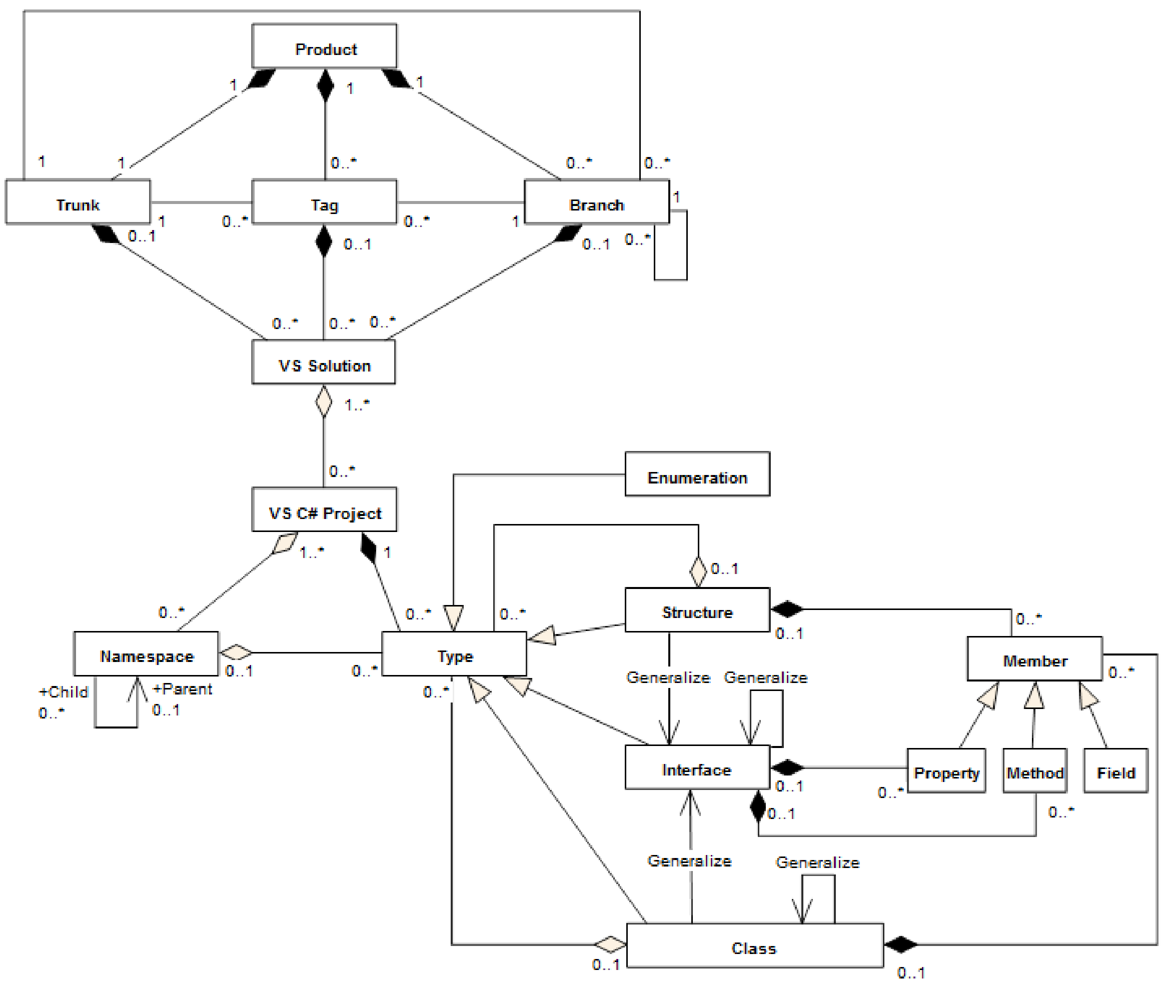


Figure 1 Model of the source code for parsing and knowledge mining

In our approach we focus on case study with C# programming language sources stored as MS Visual Studio projects and solutions on a source control server with a fairly common structure. In Figure 1 we present the structure mostly applicable to any object oriented language and any development environment.

Every source code refers to a product, which is divided into three main parts: *trunk* stores the main development *branch*. Another parallel *branch* can be created copying the *trunk*. Both the trunk and the branches are mostly active. Their source code is changed over time. On the other hand, one *tag* represents a stable version of either a single *branch* or the *trunk*.

In the case of MS Visual Studio, C# source codes aggregate into Visual Studio *projects*, which further aggregate into Visual Studio *solutions*. One project can belong to more than one solution. We abstract away from individual source code files and folder structure and focus rather on relations in the source code itself.

In C# and other languages, *namespaces* are used to be hierarchically structured. One namespace has often more child namespaces and one parent namespace. *Types* can be declared inside or outside a single namespace. In our approach, we consider four main types. The most important types are *class* and *interface*, as they represent the base building blocks of all object oriented programs. Through their structure, behavior and data, they contain the main portion of knowledge of a product. Just like namespaces, classes can be also hierarchically structured. Multiple types can be nested in a single class. Another important kind of hierarchy is generalization or derivation which helps to understand the knowledge from the view of concretization/specialization from general objects to closer aspect.

2.2 Parsing and Visualization

We use AST (Abstract Syntax Tree) technology for extracting information from source code [10,11,1]. For this purpose we use an open source library NRefactory [3] with integrated development environment SharpDevelop [4]. The extracted ASTs are processed then.

The *ProcessAST* is a recursive procedure which processes the specified nodes from the AST.

```
ProcessAST(IN:nodes)
   For each node in nodes
      If node is namespace
         ProcessNamespace(node)
         ProcessAST(node.ChildNodes)
      Else if node is class or interface or structure
         ProcessType(node)
         ProcessAST(node.ChildNodes)
             //children: methods, fields & properties of the type
      Else if node is enumeration or method or field or property
             ReadNode(node)
```

The *ReadNode* procedure represents a stage in the extraction process, when the specified part of the information is considered to be extracted. Attributes of the node are simply read and stored. The information of the node is connected to the information of its parent node.

The *ProcessNamespace* procedure extracts the knowledge from a namespace. A namespace can contain one or more classes, interfaces or other namespaces. In C# a tree hierarchy of namespaces can be created in these two ways:

- A child namespace is declared inside the scope of a parent namespace.
- Each namespace is identified by its full name, containing names of all its parent namespaces as well as the name of the namespace itself. The names are separated by dots. In this case, the name itself defines the hierarchy. For example a namespace declared with name *SuperParent.Parent.Child* has name *Child* and parent *Parent*. The parent namespace of *Parent* is *SuperParent*. A namespace declaration using dots in the name automatically declares all parent namespaces listed in the name. In

the example above with namespace *Child*, *Parent* and *SuperParent* namespaces are declared as well.

The *ProcessType* procedure extracts the knowledge from a class, an interface or a structure. In C# these types can be declared inside or outside the scope of a namespace and in the scope of other class or structure. When a type is declared inside the scope of other type, it is called nested. This way a tree hierarchy of types can be achieved.

Each class, interface and structure is identified by its full name. The full name is composed of the name of the owning project, the full name of the owning namespace, names of all owning types and the name of the type itself.

In C# a single class, interface or structure can be declared more than once at more places. Such declarations are called *partial declarations*. Two parts of the same type can be stored in one or more files, yet they must still exist in the same project, the same namespace and the same owning type – they have the same full name. We merge partial declarations of one type by merging its methods, fields and properties into one type information. When a part of a single type is processed, its methods, fields and properties are added to the information of the type as a whole rather than to the information of just one part.

For visualization we use graphical engine Ogre3D [2] and the Fruchterman-Reingold algorithm [5] as a force-directed layout method, minimizing the energy of the system by moving the nodes and changing the forces between them.

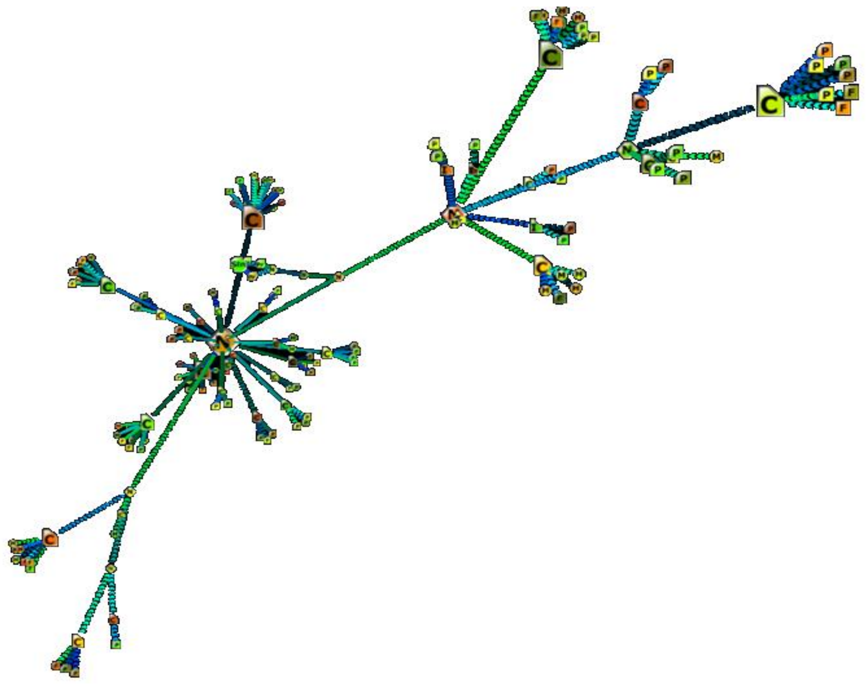


Figure 2 Visualization of the source code

3 Content and Code Type Recognition

Detection of the source code content is surprisingly easier than author recognition, but with the same ambiguous results, if we have no relevant data. We will derive content

- From the names of the project, classes, methods, attributes, fields, properties
- From the changeset attributes
- From the links, comments and code marks with keywords
- From the association (specialization, aggregation, association)

Purpose of the code type recognition is to identify the tier of code in systems based on the 3-tier architecture [8] divided systems into *Presentation layer* (presents results of a computation and data to the system users and collects user input), *Application processing layer* (provides application specific functionality) and *Data management layer* (manages the system databases).

Code type recognition can be performed on a multiple levels of code, such as classes or namespaces, or project. For instance, if we know that a project implements presentation layer, we may also assign all classes and namespaces that it contains to the same layer.

Next possible approaches check

- Project templates, used to specify standard components for a newly created project. Templates delivered with MS Visual Studio are usable for instance to create web projects or GUI (Graphical User Interface) projects like WPF (Windows Presentation Foundation) or Windows Forms. To gather template from an existing project file, it's necessary only to read the content of an XML element under "/Project/PropertyGroup/ProjectTypeGuids/text()" XPath and thus the recognition process is very simple. Each project has at least one template, but it may also contain multiple templates. For instance C# WCF Service Library project has two project type GUIDs (Globally unique identifier). One describes the project as a WCF project and other as a Windows C# project**Error! Reference source not found.**. Recognition by project template classifies a whole group of source code solely by common characteristic given by the project. Therefore it should be used only in case when a project doesn't implement multiple code tiers. Another problem is that the some of the project templates found in MS Visual Studio are not necessarily specialized for a one code tier. Common example is Class Library template, which is defined only as a template for reusable classes and components.
- Common libraries - Microsoft .NET Framework consist of multiple standard libraries (assemblies) and some of them are usually specific for a single code tier. For instance System.Data.dll is usually used for accessing databases so it may be advisable to declare code that is using this library as a data management layer. We may therefore expect that by capturing usage of these common libraries it may be possible to estimate the tier of code in general. C# projects contains list of referenced libraries which is possible to gather from a project file by reading values of XML attributes with "/Project/ItemGroup/Reference/@Include" XPath. This list is not sufficient just by itself, because it is not necessary to really use all of the referenced libraries, but it may provide subset for more accurate analysis process. To gather more specific results it will be necessary to scan code for used classes and to assign them with the correct libraries. Draft of this process is presented in pseudo code below, where suggested code tiers are assigned to classes. Rate of the tier assignment is computed for each class in the project and it is possible to have classes with multiple non-zero tier rates. Classes and interfaces are in **Error! Reference source not found.** uniformly called types.

```
for lib in project.GetReferencedLibraries()
    for type in lib.GetTypes():
        typeLibMap[type] = lib
for class in project.GetClasses(project):
    for baseType in class.GetBaseTypes():
        baseTypeTier = GetCommonTypeTier(baseType)
        if baseTypeTier <> None:
            classTier[class][baseTypeTier] += 100
    for method in class.GetMethods():
        for usedType in method.UsedTypes():
            usedTypeTier=GetCommonTypeTier(usedType)
            if usedTypeTier <> None:
                classTier[class][usedTypeTier] += 5
```

```
def GetCommonTypeTier(type)
    if type in typeLibMap
        typeLib = typeLibMap[type]
        if typeLib in libTierMap
            return libTierMap[typeLib]
    return None
```

- Class name or namespace – Name of the class should always reflect its designated purpose. In this case it may be possible to search for common words in class names like “Data” or “Web” in order to recognize tiers of classes. Besides the class names it is also possible to scan names of the classes’ namespaces or even projects
- Author – Authorship of the code may provide another technique for code type recognition. In case when the author is strictly specialized to a single code tier, it should be possible to successfully assign code to author’s tier. It is necessary to take into a consideration, that the one code may have multiple authors and therefore it is advisable to measure contributions of all authors on the finale result.
- Explicit naming rules – This approach is similar to approach based on class names but it works only for strictly defined naming conventions specified by some guidelines. We are expecting that this technique may provide accurate results, but it will be possible to use it only on a small number of cases.

We may assume that none of the described techniques alone will provide sufficient results in real life systems and it will be necessary to combine their assessments to compute final results. Success rate of each technique must be measured to generate their initial weights in the result computation and these weights should be gradually adjusted during real life usage by rating or correcting of result assessments by human users.

4 Source code users

4.1 Authors

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

- real authors of the content (they create, add, change or delete important parts of code and change logical aspects)
- editors, modifiers (they refactor the code and they understand it)
- commentators (they are making comments, sorting elements, formatting code, …)
- bunglers

By another criterion, authors can be distinguished to:

- Authors of files and lines of source code
- Authors of elements of source code – project, package or module, namespace, class, interface, field, property, method, statement

This criterion determines, how can be author mapped or bind to given source code fragment and how this bindings will be represented. In the first case, authorship may be bind to most nested element of code, then its implicitly aggregated and presented at all superior elements. In the second case, authorship must be bind to individual lines of source code. Authorship of file may be apprehended by two approaches. Either author is person, who created file and coauthors are modifiers of file attributes, or coauthors are authors of source code fragments, thus content. If the fragment of code is deleted, in the first case, authorship can be moved to superior element. In the second approach, it is not so explicit.

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, therefore authors of last changes of source code elements.
- Authorship based on life cycle of source code development to particular changeset.
- Authorship based on the whole life cycle of source code. It means that if presented changeset is not the latest, there will be presented authors or future (ad efectum of presented changeset) changes too.

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

4.2 Comparisons of code lines

In this approach, source code authors detection is based on the author determination at the level of source code files and lines.

As the basis of this approach can be used function Annotate available in MS Visual Studio 2010 as part of Team Foundation Server 2010, which enables presentation of information about changes in given file of particular changeset. Inter alia, it shows attribute owner, which is author of last change at the level of source code line. Annotate has been created as extension to VS 2005, where this function got and compared all versions of particular file and the output was for every line labeled by attributes changeset, date and user of last change. Deleted lines didn't appear.

Principle of this approach is that in first version, author of all lines is one person. In the following versions, some code lines are added, deleted or edited. If they are added, new author with other attributes is recorded. If code line is deleted, line information is ignored.

But we cannot consider author of the source code author only the person making last change, but all persons, who participate in the code evolution process involving extending, modifying and reduction of code too. Authors of code can be considered persons, whose source code fragments have been modified so far, that nothing of it left. But these fragments and know how could have been the basis of further changes, so original authors should be involved. Another case is the code fragment replaced with the other due to faultiness, depreciation of use of new or another code library for implementation. Determine the cause of code fragment replacement can be very complicated, if not impossible, but using of suitable mark can solve this problem.

If we will build up on the basis of function Annotate and we want to analyze authors of source code fragments, we must consider authors involved in the whole code evolution, not only the authors of the last changes of lines. There must be stored information about deleted lines and changed lines too and distinguish between change of line and delete/add of line.

This principle of source code authors detection is especially suitable, if we want to find out authorship on the level of method bodies. With the use of information retrieved, we can then map or bind authorship to superior elements.

4.3 Comparison using Abstract Syntax Tree

Another approach in detection of authorship and other information about source code fragments is based on principle of processing source code represented in hierarchical abstract syntax tree (AST). This approach is more suitable, because AST reflects individual source code elements (class, method, property, etc.), so it enables to manipulate with them directly and move from superior to nested elements, from the root to leaf nodes, practically from project node to elements like method, field, property. Versions are compared using differences in trees. If they are different, comparison proceeds to nested elements and new iteration in compartment begins. Comparison will proceed to the list elements. In individual elements we can analyze differences.

4.4 Comparison techniques

Comparison of particular (two) source code versions can be done by two approaches:

- Getting and making differences. This approach supposes full get of all source code versions and continual comparison of two version, from the oldest to most recent. For the severe processing in the context of time complexity, this processing is suitable for full source code analysis, which means, that source code wasn't previously analyzed and meta model with author information should be created.
- Pending changes during check-in. Second approach supposes that source code was previously analyzed and meta model was created. There is always compared currently checked-in version to latest version, to which valid meta model exists. There can be optionally compared pending changes to latest valid version. Opposite to previous approach, this approach of analysis is less time complex, its constant versus linear.

5 Code Marking for Collaborative Programming

Code marking can help us to clarify the code with keywords (finished, completed, open, obsolete, uncompleted, example, unusual, probe, stable, final, test, original, reused), implementation [9] and design [7] patterns (Observer, Proxy, Mediator, Visitor, etc.), antipatterns [6] (Duplicated Code, Long Method, Divergent Change, etc.), rating (slow, fast, good, bad), etc.

Source code marking is our experimental method, how to enrich the communication and project with keywords, features, remarks, links (to the knowledge sources) and rating.

Asset of this approach is that it can be used in searching content, features (), good material, implementation and design patterns, another similar mistakes for correction,

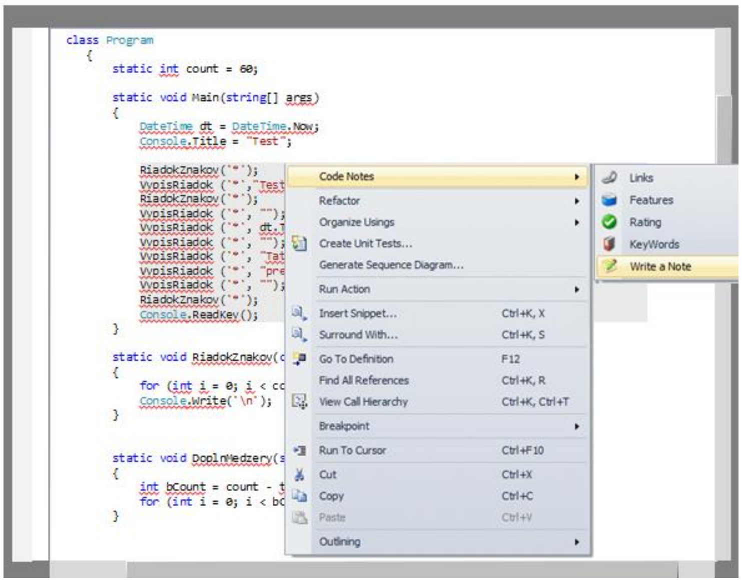
We can monitor

- who generates most of good or bad rating marks
- which programmers are usually authors of „antipatterns“
- which programmers writes unusual (non-standard , strange) codes
- occurrence of problematic codes related to projects (in which project(s) occurred)
- show relations with different projects

Common shared environment with code-marking brings two benefits:

- spontaneous *information (discussion) forum* about entities of developed software,
- *knowledge base* for analysis.

The advantage is in added information, referring directly to entity (project, class, method) and visible during work with this entity in developing environment.



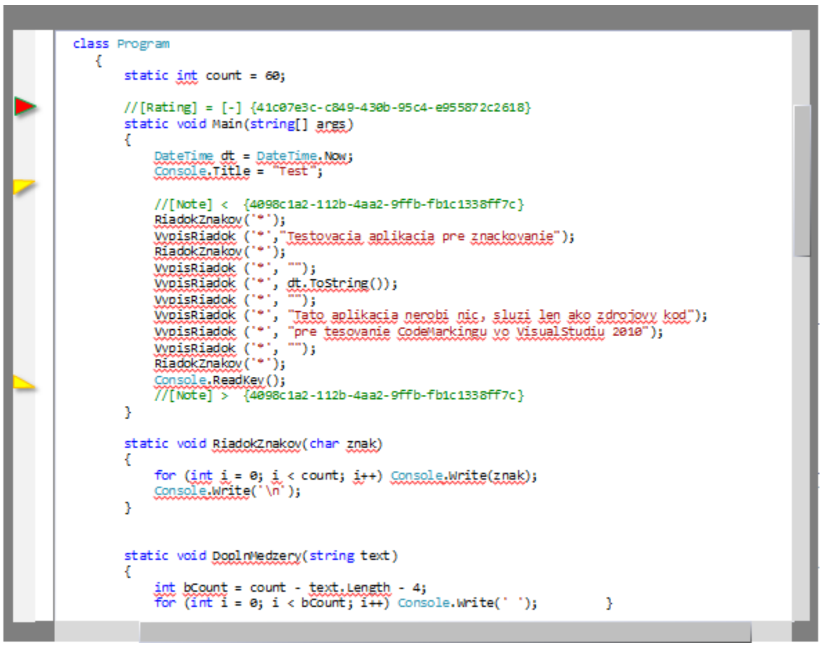
```
class Program
{
    static int count = 60;

    static void Main(string[] args)
    {
        DateTime dt = DateTime.Now;
        Console.Title = "Test";

        RiadokZnakov('*');
        VypisRiadok ('*',"Test
        RiadokZnakov('*');
        VypisRiadok ('*', "");
        VypisRiadok ('*', dt.T
        VypisRiadok ('*', "");
        VypisRiadok ('*', "Tat
        VypisRiadok ('*', "pre
        VypisRiadok ('*', "");
        RiadokZnakov('*');
        Console.ReadKey();
    }

    static void RiadokZnakov(c
    {
        for (int i = 0; i < co
        Console.Write('\n');
    }


    static void DoplnMedzery(s
    {
        int bCount = count - t
        for (int i = 0; i < bC
    }
```

Figure 3 Selecting the block and adding the code mark (*Note*)

We analyze four approaches of the code marking:

- inserting identifiers to the source code, written as a comment acceptable in actual programming language, using generated *GUIDs* (Globally Unique Identifiers), for example: `//{3a768eea-cbda-4926-a82d-831cb89092aa}`
- referring code path and position, e.g. saving a path (namespace/ project/class/method) with line number, hash of Abstract Syntax Tree Pattern (ASTP), etc.
- saving in auxiliary file,
- saving identifier and whole remarks in the source code as a comment:

`//{3a768eea-cbda-4926-a82d-831cb89092aa} open, Proxy, accounting`



```
class Program
{
    static int count = 60;

    //[Rating] = [-] {41c07e3c-c849-430b-95c4-e955872c2618}
    static void Main(string[] args)
    {
        DateTime dt = DateTime.Now;
        Console.Title = "Test";

        //[Note] <  {4098c1a2-112b-4aa2-9ffb-fb1c1338ff7c}
        RiadokZnakov('*');
        VypisRiadok ('*',"Testovacia aplikacia pre znackovanie");
        RiadokZnakov('*');
        VypisRiadok ('*', "");
        VypisRiadok ('*', dt.ToString());
        VypisRiadok ('*', "");
        VypisRiadok ('*', "Tato aplikacia nerobi nic, sluzi len ako zdrojovy kod");
        VypisRiadok ('*', "pre tesovanie CodeMarkingu vo VisualStudiu 2010");
        VypisRiadok ('*', "");
        RiadokZnakov('*');
        Console.ReadKey();
        //[Note] >  {4098c1a2-112b-4aa2-9ffb-fb1c1338ff7c}
    }

    static void RiadokZnakov(char znak)
    {
        for (int i = 0; i < count; i++) Console.Write(znak);
        Console.Write('\n');
    }


    static void DoplnMedzery(string text)
    {
        int bCount = count - text.Length - 4;
        for (int i = 0; i < bCount; i++) Console.Write(' ');        }
```

Figure 4 One single (*Rating*) and one pair mark (*Note*)

Figures 3 and 4 show description of

- *single marks* (e.g. *rating*) used for marking whole *Project*, *Class*, *Methods*, *line*, etc.
- *pair marks* (two marks with the same identifier) used for marking first and last line of *text block* (e.g. *note*).

Because of simple identification of the pair and single marks, we use one-character prefix '=' for single mark, and '<' & '>' prefix for the opening and closing marks.

6 Knowledge Sources for the Collaborative Development

If we want to present heterogeneous information sources in common manner, it is necessary to create complex system for optimal data acquisition from these sources, tracking of changes, synchronization of data, processing and presentation of acquired data.

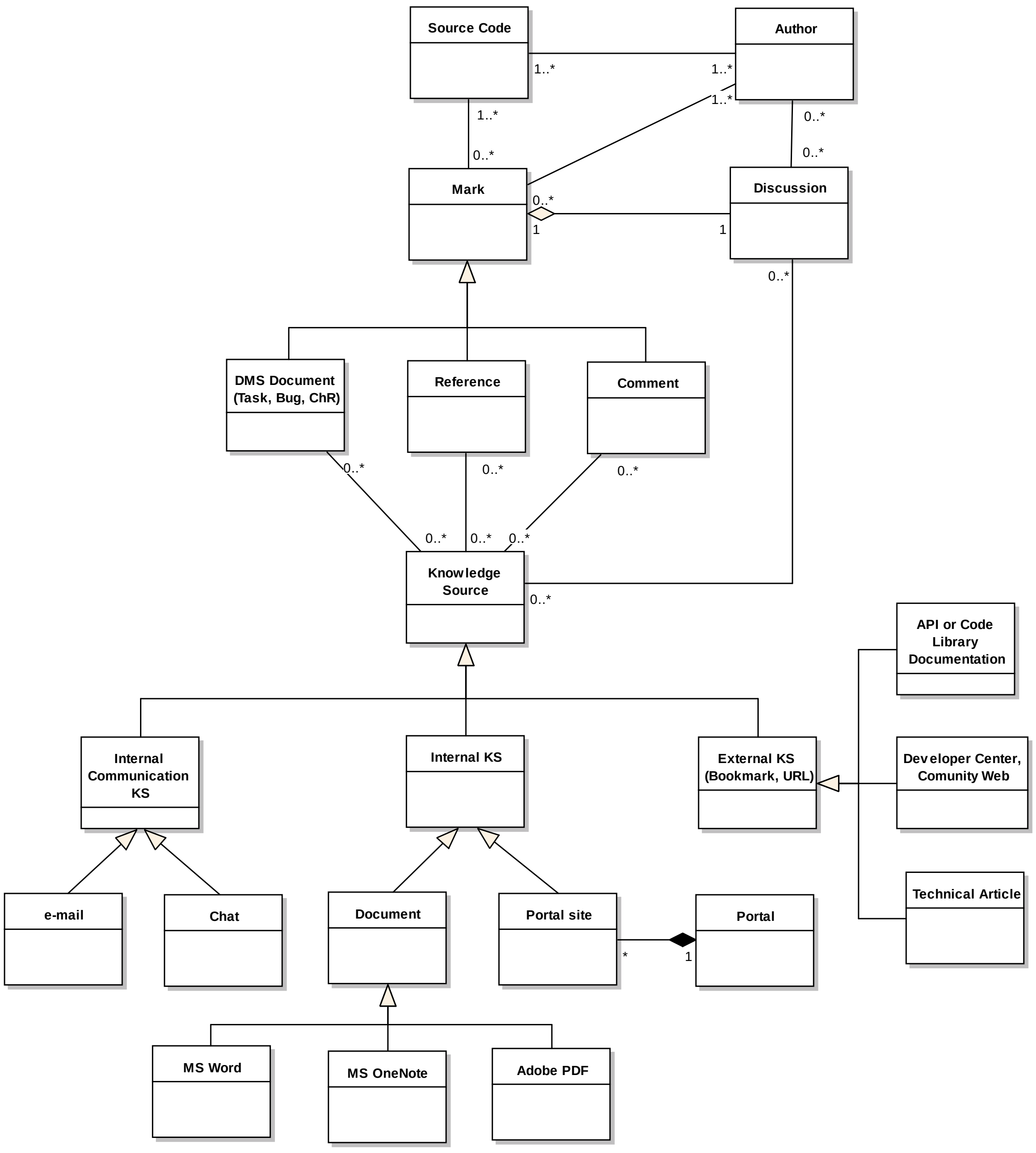


Figure 5 Model of the knowledge sources over the source code

Automatic creation can be done in the case of source code marking with files from Document Management System (DMS). Marks are created within the check-in operations of TFS, when the source code changes are bind to correspondent document from DMS – implementation task (TASK), product bug (BUG), product change request (CHR), etc. Relation on knowledge source can be created by various manners:

- In the comment of the check-in operation, URL to knowledge source is stated (e.g. Web URL, document or portal page), which can be transferred to mark itself.
- In some attribute of the DMS document itself (e.g. comment), URL to knowledge source is stated.
- If the knowledge source is reach text document in some format (.pdf, .doc, OneNote, etc.), this can be attachment of DMS document, relation to it presented in graph is created through the binding to DMS document.

Creation of relationships by user is supposed to associations created through the manipulation with marks. Relation arises on the basis of URL to knowledge source as follows:

- As part of content in DMS Document Mark
- As part of content of discussion for random mark
- Directly by creating of Reference Mark
- As part of content of Comment Mark

7 Conclusions and future work

This paper describes our preliminary study, plans and visions in this part of research project and our first prototypes for parsing, visualization, code marking and user modeling. In the next steps we will enrich research of user model, knowledge mining, visualization of sources, knowledge and people in 3D graph using our application of 3D cave.

Acknowledgment. This work was partially supported by the Scientific Grant Agency of Slovak Republic, grant No. VG1/0508/09. This contribution is also 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.

Literature

1. 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
2. Object-Oriented Graphics Rendering Engine OGRE www.ogre3d.org (Accessed on 19 August 2011)
3. Grunwald D.: NRefactory. SharpDevelop http://wiki.sharpdevelop.net/NRefactory.ashx (Accessed on 19 Agust 2011)
4. The Open Source Development Environment for .NET. IC#Code. http://sharpdevelop.net/opensource/sd (Accessed on 19 August 2011)
5. Fruchterman, T. M. J., & Reingold, E. M.: Graph Drawing by Force-Directed Placement. Software: Practice and Experience, 21(11), 1991

6. Fowler M. et al: Refactoring: Improving the Design of Existing Code, Addison-Wesley, 2000
7. Gamma E. et al.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995
8. Buschmann F. et al.: Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing, Wiley, 2007
9. Beck K.: Implementation Patterns, Addison-Wesley, 2007
10. 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
11. Maletic, J.I., Collard, M.L., Marcus, A.: Source code files as structured documents, In: 10th IEEE International Workshop on Program Comprehension, Paris (2002)

Contact data:

Ivan Polášek, Ing., PhD.,
FIIT STU, Ilkovičova 3, 842 16 Bratislava 4, Slovakia
polasek@fiit.stuba.sk

IVAN RUTTKAY-NEDECKÝ, ING.
Gratex International, a.s., Galvaniho ul. 17/C, Bratislava 821 04, Slovakia
ivan.ruttkay-nedecky@gratex.com

PETER RUTTKAY-NEDECKÝ, ING.
Gratex International, a.s., Galvaniho ul. 17/C, Bratislava 821 04, Slovakia
peter.ruttkay-nedecky@gratex.com

TOMÁŠ TÓTH, ING.
Gratex International, a.s., Galvaniho ul. 17/C, Bratislava 821 04, Slovakia
tomas.toth@gratex.com

ANDREJ ČERNÍK, ING.
Gratex International, a.s., Galvaniho ul. 17/C, Bratislava 821 04, Slovakia
andyc@gratex.com