ISSN: 2229-6956 (ONLINE)
DOI: 10.21917/ijsc.2021.0334

CODE PRESENCE USING CODE SENSE

M.A. Krishna Priya[1] and Justus Selwyn[2]

[1]*Department of Computer Science, Bharathiar University, India*
[2]*School of Computing Science and Engineering, Vellore Institute of Technology, Chennai, India*

Abstract

Programmers are tightly occupied with the development workflow. Conscious contemplation of the right code solution and proactive presence of the solution lists is presented in this paper. Refining and sensing the optimum solution lists and the proactive presence of code solutions is incorporated in this code sense algorithm. This architecture allows developers to generate and integrate best code solutions directly in to solution bases. Code Sense Interface can support collaboratively with various platforms which can be plugged in with any development interface. Using Code sense, proactive presence of code is implemented as an assistant named Code Proactive Assistant. Code suggestions for a particular requirement scenario are considered as the primary goal. In addition to showing the feasibility of this approach, it provides further evidence in support of the claim that integrating specialized code sense interfaces directly into the editor is valuable to professional developers.

Keywords:

Code Sense, Code Proactive Assistant, Collaborative Learning

1. INTRODUCTION

Nowadays, software developers tend to use code support feature which is found in modern source code editors. These editors provide hand in hand support for software development with its floating menu which contains related data such as methods, types etc. By selecting the relevant data from this menu, developers can avoid typographical and syntax errors and also unnecessary navigation between the programming interface and the search engine can be avoided. Developers can also explore unfamiliar development interfaces without the psychlogical threats which incurs for any human while facing a brand new environment. With this code sense feature developers need not face the mental overhead that is associated with switching between different applications or search engines for the appropriate or right documentation and suggestions.

Code support menus are previously been suggested in the literature. These have focused on leveraging additional sources of information, such as databases of usage history [1] [2], inheritance information [2], API specific information [2] [3], partial abbreviations [4], examples extracted from code repositories [5] [6] and crowd sourced information [7] [8], to increase the relevance and sophistication of the featured menu items.

In this system, the code sense feature primarily supports the editor with code solutions for the developer s keyword. This Knowledge management strategy determines the solution code sets that are available in the solution base; so far library providers do not specify contextually relevant code sets for any required logic. In this paper, a technique is proposed called code sense that consciously finds mapped code solutions for the context using the required scenario which is sensed with the user s keyword, and also supplies suggestions proactively. This assistant can be integrated as a plugin with developer's editor directly. When developers are familiar using this feature coding task becomes significantly simpler. Any developer would always prefer to use tools without leaving their editing environment.

In this paper, recommendations and supports by the code sense is discussed in the context of code construction. When the developer invokes the code sense by typing the keyword at the indicated cursor position which is denoted with a symbol (?), the editor looks for a knowledge map solution code set associated with the keyword being entered, for example if a developer looks for sort code, then an associated code solutions are listed in descending order according to usage count, once the developer finds the appropriate solution for a particular context, that code set can be added in to the standard code editor by clicking on it.

The developer can interact with such recommended solution code sets execute and check immediately by providing parameters and other information related to the requirement, then post feedback about the code set, these comments can be made according to the behavior of the code being constructed. Plug in proactively picks the code solutions for the required keyword, with a click or by the press of enter key the user can select the required code solution which is inserted in to the editor. In accordance with best practices, the following questions to be addressed before designing and implementing our proactive code sense system:

1. What are the specific requirement scenarios for the development of proactive code sense system in a professional setting?
2. What capabilities the proactive code sense systems provide to identify the required scenario with a given keyword?
3. What usability for solution code sets provided in user interface designs?

A study with 45 professional developers (Section 2) helped to find answers for the above questions. Their responses, informal interviews revealed a number of small unnoticed requirements for user interfaces as well as the underlying proactive code sense architecture (Section 3). Participants also suggested a number of scenarios, demonstrating the applicability of this technique. These are organized into several broad categories (Section 4). Next, we describe Code Sense archetype that implements the proactive code sense architecture for the Java programming language (Section 5), allowing Java library developers to associate customized solution code sets with their own classes. Various design choices are described that satisfy the requirements discovered in the preliminary investigations and briefly examine necessary trade-offs.

Finally, related works are discussed with the implemented code sense system that assists developers as they type code keywords (Section 6). The study provides specific evidence in support of the broader claim 8that highly-specialized tools that are integrated directly with the editing environment are particularly useful. Fundamentally proactive code sense system is useful because it makes discovering and developing simpler.

2. STUDY

The concept of proactive code sense shows clear concrete system with to solution code sets, and also posses' requirement scenarios to justify this need by the conduction of study with professional software developers.

2.1 PARTICIPANTS

Some participants for this survey are professional developers and also computer science graduate students. In both cases the survey stated that development made easy can help out naive developers with some familiarity with any object-oriented programming language like Java, C# or Visual Basic and an integrated development environment like Eclipse or Visual Studio. Participants took the survey, and they were offered with the new experience. Out of the 45 people who took the survey, 23 participants completed it. The responses from completed surveys were examined.

2.2 FAMILIARITY WITH TECHNOLOGY

The participants were asked about their level of familiarity with several programming languages, on a five-point Likert scale. 63% of the participants indicated that they were an expert in at least one language, and an additional 37% were "very familiar" with at least one language. On average, participants rated themselves as very familiar with Java, C, C++ and JavaScript, familiar with C#, Python and PHP and somewhat familiar with Visual Basic and Perl. It is been asked the participants to select which integrated development environments (IDEs) and code editors that they were familiar with. The Eclipse IDE was familiar to 82% of participants. This was followed by Visual Studio at 51%, Netbeans at 77%. Participants could also enter "other" choices and a number of editors and IDEs were entered, including BlueJ, Notepad++.

2.3 MOCK SOLUTIONS

Participants were presented with a series of mock solution codesets for a Sort class. Participants got to see the mock screenshots that demonstrates how a developer uses the cursor that provokes the solution popups, and mock shows how the codeset would be inserted once a selection had been made. For the Sort class, the majority of participants indicated that they would look in the code sense popups (56%) or documentation (20%) for a predefined code corresponding to sort. Another 24% indicated that they would either do on their own or use an external search engine to determine the corresponding code to the sort. Finally, after showing the series of mock screenshots, participants were asked to rate how useful the integrated system would be to them if they need to work for the corresponding requirement. The responses to this question according to individual it varied, 13% found using it every time, 27 % used most of the time, 32% utilized some of the time, 11% used rarely, 17% were proficient enough they did not use at all. Half of the participants indicate that they would use code sense at least some of the time. In addition with rating, remarks from participants were also collected. These responses were helpful while developing the design criteria.

3. DESIGN IDEAS

The above study and suggestions from developers helped out in designing the system. In the section headings below, the number of survey responses, summed over the mock screens, that contained the solution lists are listed in parenthesis. These criteria were useful in designing Code Sense (Section 6) and it is noted that this criteria may also be relevant to editors.

3.1 RECOMMENDATION RUBRICS

Many developers noted that in the prototyping phase of a project, if recommendations also expressed in along with the solution code sets, it would be more helpful to pick a solution and try it with inserting parameters so that it could be tested. The multiple resulting solution code list often confuses several participants, and they suggested for a note that the information such as Best used, Better used, Never used should appear in solution lists.

3.2 TESTING CAPABILITY

The code set shown to the participants allowed users to immediately test a solution against provided parameters. These test values and the matched results were inserted as comments below for the picked source code. A number of participants requested to generate unit tests, to support the generation of unit tests, the proactive code sense architecture need to support code testing locations which can be separately used for testing.

3.3 HISTORY SUPPORT

Several participants asked for the ability to revoke option for previously selected code sets. In order to support this feature, the architecture must provide the extension with enough information to reconstruct. Users might need to modify the selected code and have these modifications reflected in the new solution code set upon solution knowledge base.

3.4 PORTABILITY

The mock screens were showed to the users based on the Java and the Eclipse IDE. As we showed, a number of participants preferred other languages or editors. Many of these participants made comments asking that the features of IDE and programming language independent. Indeed, the solutions could be used with only slight modifications in a variety of programming languages, given suitable architectural support for porting between editing environments.

4. REQUIREMENT SPECIFICATION

At the end of the study, the participants were asked to suggest other code sets that benefits to support the claim of the proactive code sense that characterize the specific scenarios. A total of 119 participants made one or more suggestions, which is classified into following categories.

4.1 POPUP WINDOWS

These popular suggestions are given as popup windows, influenced perhaps by the solution code sets. While other participants were focused on user interface elements, such as buttons and layouts, A few also suggested editing popups for manipulating solution code sets to enhance according to recent scenarios with intuitive manner.

4.2 SIMPLIFIED SYNTAX

In some cases, simple syntax recommendations were noted and that syntax provided was felt more desirable by users. To implement, the popups can also be labeled with two rubrics, one is practice or usage rubric such as best used, better used and never used, another rubric is for complexity such as simple, medium and complex, which helps the user to avoid popups that contains complex solution sets

A related category of suggestions consisted of solution sets that offered a more natural interface containing code in different languages with the additional parameter mentioning the language or the domain to identify the required syntax that is domain-specific syntax for a particular expression or statements in the codeset.

4.3 DOCUMENTATION

Some relevant examples integrated with demos were suggested by some participants along with the editor, so that these solutions with the detail documentation can help the users and it also provides instinct to choose the right solution.

4.4 INTERFACE

According to the well-known Curry-Howard isomorphism between programming languages and formal logics, proof terms correspond to expressions and propositions correspond to types [9]. Proactive code sense works directly with the required scenarios to help developers construct code sets. If it is integrated to a language environment with an interoperable connection for code construction, then it would be useful for building code interactive assistant interface.

5. RELATED WORK

In addition to the code sense work discussed above, some other research areas related are listed here.

5.1 LANGUAGES

Graphical user interface entity that generates text representations can be considered as an approach that follows interaction techniques from visual and conventional programming languages.This approach addresses some of the usability challenges previously associated with visual languages [16].

5.2 LIBRARIES

Libraries behave as a support element for development activity. Active libraries [16] are libraries that contain methods or procedures that is invoked at either compile-time or design-time.

5.3 PRODUCTIVITY

Clean code is less likely to contain errors and is easier to test, understand, and modify-all factors that contribute to fewer bugs and greater reliability. The Green Hills Compilers [17] enable enforcement of clean coding conventions, which is a collection of Compiler warnings and errors that enforces a stricter coding standard than regular C and C++. The Builder automatically analyzes the dependencies of a project and compiles and links as many files in parallel as possible, taking full advantage of modern multicore systems. The Builder and other sophisticated features significantly reduce the overhead of project development. A seamlessly integrated project manager, editor, flash programmer, instruction set simulator, and more enable you to jumpstart development and work more efficiently. Barista [18] and the RBA editor merge concepts from both text-based and structured within a relatively conventional layout. Barista provides the opportunity for rich type-specific interfaces. The RBA editor focuses on code readability rather than new modes of interaction. Both tools use a custom domain-specific language.

5.4 DEVELOPMENT ENVIRONMENT

Some more environments are specifically designed for certain types. CodeRush [19] and Resharper [20] have dialogs that allow developers to launch a color picker directly from the code editor. IntelliJ IDEA has an inline regular expression palette, driven by its Intentions system, as well [21]. However, these specific features are hard-coded user-defined types cannot provide similar functionality. Recent versions of Visual Studio support user-defined palettes associated with specific fields, rather than classes, of user interface widgets. These are shown only in the property pane when using the graphical window layout editor.

6. SYSTEM IMPLEMENTATION

The survey, helped to build this code sense system Code Proactive Assistant. It is been decided to build the system for NetBeans Java IDE because this is widely-used by the participants in the survey. The sections below describes this novel design decisions made it possible and the way it satisfies design criteria from Section 3 and enabled several use cases described in Section IV. The end result is a simple system that allows an API’s developers, as well as external developers, to build self-sustainable projects that can be associated with both built in solutions and also with user solutions. Code PA can discover and provokes solutions and displays them through the standard popup windows.

6.1 BASEWORK

This work is based on KM Trajectory Service Frame work [6] [10] which is framed to reduce the dependency on human resources in software development organizations. This Code Extractor (CE) [7] [11] works as a background process. This tool monitors the project repository and its files, which are saved recently. Code Extractor listens and extracts the code on day to day basis. Programmer’s activity is continuously monitored by CE. Code is extracted from the programmer’s application only after he/she saves it. Files are tracked, code sets from the software application’s work area is extracted. This process silently extracts source code from the work area of an application domain and stores the source code sets in code log. Code log consists of multiple code segments. Solution knowledge base is synthesized respectively [9] [12]. Trace, extract code fragment and store it, identify code scenario of the code fragment and assign code name and type, Manipulate code fragment as code pattern and Store code pattern. SKB algorithm automatically parts and extracts code fragment from code set, starts by collecting various files from selected directories, Program files are segregated through extension filter which filters extensions like c and cpp and extra. Next, it scans lines of code and extracts code fragment from each file. Keywords and comments from the extracted code fragments are used for contemplation to identify code Scenario. Solution Knowledge base consists of code segments which are extracted from code logs according to comment lines or syntax lines (symbols). The purpose of the code segment is known only when the developer documents the purpose either as comments or as a documentation note. Code retrieval from knowledge base is based on key word/ text search that acts as input parameter for querying in the knowledge base. User specifies the requirement in simple sentence; our model discovers and elicits the appropriate code from the knowledge base by mining through the keywords. This work features a model that analyzes source code and uses data from the Solution Knowledge Database (SKB) to create feature sets for recognizing a set of code patterns. Specifically, source code is represented using an Abstract Syntax Tree [13], which provides the ability to extract statements. Knowledge Map algorithm processes thousands of code fragments, discover and supply the required ones This model proposes the extraction of code sets from project repositories to present a set of solutions to the user for each requirement.

6.2 CODE PROACTIVE ASSISTANT

Conventional databases are not designed for retrieval and identification of patterns especially triggered when user types keywords or using user's keywords as factors. The approach is been developed to identify and predict appropriate code pattern with logical traits. Usually different clustering classification methods and tools are used for prediction. Wide range of keyword factors and their numerous combinations in normal conditions generate customized responses. This suggest that the identification and characterization of keywords and their code patterns shows an expression profile of similar code patterns would increase the understanding of it and provide suggestions in software manipulation to improve software growth in many fold. Advancements have revolutionized the identification of required pattern from similar kind of code patterns makes it possible to chart out required individual patterns and also to identify, compare and contrast the requirement specific code pattern of one condition.

Code Sense Algorithm

Input: repository dataset

Output: appropriate solutions for a requirement keyword

```
CodeSense (in Key, out Value)
Begin
For each row in knowledgebase do
  datalines = select records() //read data from knowledge base
    For each datalines do
        function map(each record):
        key = get codename(ticket)
        value = extract code(record)
/extract using select query where codename=ticket
// read attributes from record
        append solutionlist(key, value, usagecount)
        show(solutionlist)
    End
        /* each reduce identifies, detects and shows set of code
        patterns for a particular requirement scenario identifying a
        certain code*/
        //Sort solutionlist with usagecount in descending order
        solution list (in values, out values )
        For each solution from reduce(solution list) do
            suggestionlist.value=sortdesc(solutionlist.usagecount)
            //supply suggestionlists proactively as popups
            display suggestionlist.value
        End
    End
```

Computational methods are used to mine the patterns related to requirement scenario. Code fragments are stored in abstract syntax tree [13] format. Through user's keyword the stored structure is analyzed and matched set of code fragments are detected, discover and elicit the optimum one through the usage count value with the combination of particular keyword. The drafted blueprint is used to create a suggestion system to provide proper assistance to software developers. Classifiers with metric value will be used by the system for the code knowledge evaluation with machine learning community such as optimum rank, usage count either accept or reject. This construct infers solution using interactive learning based on the user's requirement statement or code name.

MapReduce programming model is known for processing large data sets [14]. MapReduce is a framework used for writing applications to process huge amounts of data in a reliable manner. MapReduce is a processing technique and a program model for distributed computing. The MapReduce algorithm contains two important tasks, namely Map and Reduce. Map takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs). Secondly, reduce task, which takes the output from a map as an input and combines those data tuples into a smaller set of tuples. As the sequence of the name MapReduce implies, the reduce task is always performed after the map job. Code Sense is the extension of the Knowledge map algorithm through which the populated code solutions in the solution knowledge base is learned and reflected according to the requirement scenario, next every outcome of the different input scenarios are observed with the optimum outcomes using the usage count values, finally the algorithm senses the right and appropriate code solution for the particular requirement.

- Populate solutionlist through Knowledge map. Use code keyword with knowledge map to discover requirement scenario from the available code names through map reduce.
- Sense right solutions through observing the usage count of individual solutions.

- Supply the appropriate suggestions.

6.3 GRADLE WITH GROOVY

Gradle is a new and revolutionary build tool, based on the Groovy programming language. A Gradle plugin is bulit, which can be used across many different projects. Gradle is used to implement the plugin which can be reused with the build logic, and share it. It can be built in any language can be compiled as JVM byte code. In this code sense, Java is used as the implementation language for standalone plugin project and Groovy in the build script plugin. Source plugin can be put in several places. The source for the plugin can be included directly in the build script. This has the benefit that the plugin is automatically compiled and included in the class path of the build script without having to do anything. However, the plugin is not visible outside the build script, and so it cannot be reused outside the build script it is defined in. the source for the plugin in the directory is rootProjectDir/buildSrc/src/main/groovy. Gradle will take care of compiling and testing the plugin and making it available on the classpath of the build script. The plugin is visible to every build script used by the build. However, it is not visible outside the build, and so cannot reuse the plugin outside the build it is defined in. create a separate project for the code interactive/proactive assistant (Code PA) plugin. This project produces and publishes a JAR which can then be used in multiple builds. Generally, this JAR might bundle related task classes into a single library. A class is written that implements the Plugin interface to create a gradle plugin and the plugin is applied to a project, Gradle creates an instance of the plugin class and calls the instance using Plugin.apply() method. The project object is passed as a parameter, the plugin configures the project

A new instance of a plugin is created for each project it is applied to and also the Plugin class is a generic type receiving the Project type as a type parameter. A plugin can instead receive a parameter of type Settings, in which case the plugin can be applied in a settings script, or a parameter of type Gradle, in which case the plugin can be applied in an initialization script.

Plugins offer configuration options using extension objects for build scripts and for other plugins to customize its work. The Gradle Project has an associated ExtensionContainer object that contains all the settings and properties for the plugins that have been applied to the project. Configuration for the code PA plugin is provided by adding an extension object which is an object with Java Bean properties that represent the configuration to this container. Plugin Extension is an object with a property called message. The extension object is added to the project with the name Sense. This object then becomes available as a project property with the same name as the extension object. Often, several related properties specified on a single plugin. Gradle adds a configuration block for each extension object, settings can be grouped together.

6.4 CODE PROACTIVE ASSISTANT

Code proactive assistant is an injected code within which the application renders to communicate with the native platform on which it runs. This provides access to device and platform functionality. All the main code sense features are implemented as a plugin; Plugins comprise a single JavaScript interface along with corresponding native code libraries for each supported platform. In essence this hides the various native code implementations behind a common JavaScript interface. This project is simply a Java project that produces a JAR containing the plugin classes. The easiest and the recommended way to package and publish a plugin is to use the Java Gradle Plugin. This plugin will automatically apply the Java Plugin, add the gradleApi() dependency to the api configuration, generate the required plugin descriptors in the resulting JAR file and configure the Plugin Marker Artifact used when publishing.

6.4.1 Code PA - Creating Plug-In:

```
plugins { id ‘java-gradle-plugin’}
gradlePlugin {plugins {simplePlugin {
 id = ‘org.sense.pa’
implementationClass = ‘org.gradle.SensePlugin’
 } } }
```

Plugin ids are fully qualified in a manner similar to Java packages (i.e. a reverse domain name). This helps to avoid collisions and provides a way to group plugins with similar ownership. Plugin id should be a combination of components that reflect namespace and the name of the plugin it provides. For a Github account named “sense” and plugin named “pa”, a suitable plugin id is com.github.sense.pa. Although there are conventional similarities between plugin ids and package names, package names are generally more detailed than is necessary for a plugin id. For instance, it might seem reasonable to add “gradle” as a component of plugin id, but since plugin ids are only used for Gradle plugins. Generally, a namespace that identifies ownership and a name are all that are needed for a good plugin id.

6.4.2 A Build for Code PA:

Code Sense Pseudo Code

```
// Import packages
import com.eviware.soapui.support.XmlHolder
def groovyUtils = new
com.eviware.soapui.support.GroovyUtils(context)
def projectpath = new
com.eviware.soapui.support.GroovyUtils(context).projectPath
import groovy.sql.Sql
import java.sql.Driver
Class SensePlugin
{
  //code to connect to solution DB
  def SQL = Sql.newInstance(‘jdbc:sqlserver:”solution env”,
  ‘codesense’,
  ‘codesense’,’com.microsoft.sqlserver.jdbc.SQLServerDriver’)
  Database
  // SQL query
  def sqlStr = “Select codekey, codesolution from
  SCHEMA.solutiondb where codekey = ‘12345’”
  // defining a map to store the possible solutions from the DB
  def possSolutions =  [:]
  int SolutionID = 1
  // Executing a SQL query
  sql.query (sqlStr) { row ->
  while (row.next ()) {
  def solution = row.getString (“solutiondb”)
    SolutionID = SolutionID+1
    possSolutions.put (SolutionID, solution)
  }
  def sortedsolutions = possSolutions.sort {it.key}
  log.info ”The possible solutions are “ +sortedsolutions
}
```

6.4.3 Publishing Plug-In:

To publish Code PA plugin internally for use within the organization, publish it like any other code artifact. The plugin in a build script is configured in the repository in pluginManagement {} block of the project’s settings file. The plugin has been published to a local repository:

```
plugins { id ‘org.sense.pa’ }
```

In addition to plugins written as standalone projects, Gradle also allows you to provide build logic written in Groovy as precompiled script plugins. These are written as *.gradle files in src/main/groovy directory Precompiled script plugins are compiled into class files and packaged into a jar. For all intents and purposes, they are binary plugins and can be applied by plugin ID, tested and published as binary plugins. To apply a precompiled script plugin, It is required to know its ID which is derived from the plugin script’s filename src/main/groovy/sense.java-library- convention.gradle Plugin ID sense.java-library-convention. To implement and use a precompiled script plugin in a buildSrc project, create a buildSrc/build.gradle file that applies the groovy-gradle-plugin plugin, then create a new java-library-convention.gradle file in the buildSrc/src/main/ groovy directory and set its contents buildSrc/src/main/groovy/java-library-convention.gradle

```
plugins {
    id ‘java-library’
    id ‘checkstyle’
}
```

This script plugin simply applies the Java Library and Checkstyle Plugins and configures them. This will actually apply the plugins to the main project. Finally, apply the script plugin to the root project. Apply the precompiled script plugin to the main project plugins {id ‘java-library-convention’}. In order to apply an external plugin in a precompiled script plugin, it has to be added to the plugin project’s implementation classpath in the plugin’s build file properties filename matches the plugin id and is placed in the resources folder, and that the implementation-class property identifies the Plugin implementation class. And because an extension object is simply a regular object, using an extension object extends the Gradle groovy script to add a project property and groovy block which can be nested inside the plugin block by adding properties and methods to the extension object. Mapping extension properties to task properties. The imperative logic is hidden in the plugin implementation.

```
buildSrc/build.gradle.
plugins { id ‘groovy-gradle-plugin’ }
repositories { jcenter() }
dependencies{implementation ‘com.bmuschko:gradle-docker-plugin:6.4.0’}
implementation-class=org.gradle.SensePlugin
class SensePluginExtension
  {
  String message
  String codename
  }
class SensePlugin implements Plugin<Project>
  {
  void apply(Project project)
  {
  def extension = project.extensions.create(‘codename’, SensePluginExtension)
  project.task(‘sensing’)
  {
  doLast
  {println”${extension.message}for ${extension.codename}”
  } } } }
apply plugin: SensingPlugin
// Configure the extension using a groovy block
sensing {
 message = ‘solution’
 codename = ‘codekey’
}
```

Capturing user input from the build script through an extension and mapping it to input/output properties of a custom task is a useful pattern. The build script author interacts only with the domain specific language defined by the extension

7. CONCLUSION

Embedding specialized tools into a developer’s environment is ultimately helpful; the code sense concept is a general archetype which supports proactively that can be easily followed in any level of organization. The usability is identified with a number of use case scenarios Design constraints as well as the archetype of the tool is recognized through a survey from developers. With these findings, code proactive assistant is built that suggests solution decisions.

7.1 FUTURE WORK

Code sense systems can be enhanced in various perceptual angles which will considerably ease the software process for sure. In future, it is possible to explore an extensible machine driven code generation with intuition that instincts to identify a particular solution according to the need to eliminate the difficulties for the right decisions to follow the solutions. Even though software developer is well versed in particular technology, the developers have to update themselves frequently to accommodate their skill with the new technologies, in order to address this problem an unified code representation, unified code patterns can be used in this supply system as a further work.

REFERENCES

[1] R. Robbes and M. Lanza, "How Program History Can Improve Code Completion", *Proceedings of 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 317-326, 2008.

[2] D. Hou and D. Pletcher, "An Evaluation of the Strategies of Sorting, Filtering, and Grouping API Methods for Code Completion", *Proceedings of IEEE International Conference on Software Maintenance*, pp. 233-242, 2011.

[3] H.M. Lee, M. Antkiewicz and K. Czarnecki, "Towards a Generic Infrastructure for Framework-Specific Integrated Development Environment Extensions", *Proceedings of International Workshop on Domain-Specific Program Development*, pp. 1-12, 2008.

[4] S. Han, D.R. Wallace and R.C. Miller, "Code Completion from Abbreviated Input", *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, pp. 332-343, 2009.

[5] M. Bruch, M. Monperrus and M. Mezini, "Learning from Examples to Improve Code Completion Systems", *Proceedings of 7th European Conference on Software Engineering*, pp. 213-222, 2009.

[6] J. Brandt, M. Dontcheva, M. Weskamp and S.R. Klemmer, "Example-Centric Programming: Integrating Web Search into the Development Environment", *Proceedings of ACM Conference on Human Factors in Computing Systems*, pp. 513-522, 2010.

[7] M. Mooty, A. Faulring, J. Stylos and B. Myers, "Calcite: Completing Code Completion for Constructors using Crowds", *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 15-22, 2010.

[8] Snipmatch, Available at: http://languageinterfaces. com/, Accessed at 2017.

[9] B. Ellis, J. Stylos and B. Myers, "The Factory Pattern in API Design: A Usability Evaluation", *Proceedings of International Conference on Software Engineering*, pp. 302-312, 2007.

[10] M.A. Krishna Priya, "Trajectory Schema Service Frame Work for Software Development Organizations", *International Journal of Engineering and Technology*, Vol. 7, no. 3, pp. 616-620, 2018.

[11] M.A. Krishna Priya and Justus Selwyn, "Code Knowledge Acquisition for Knowledge Management Trajectory Framework", *International Journal of Recent Technology and Engineering*, Vol. 8, No. 3, pp. 1-12, 2019.

[12] M.A. Krishna Priya and Justus Selwyn, "Synthetization of Solution Knowledge Base", *International Journal of Analytical and Experimental Modal Analysis*, Vol. 11, No. 10, pp. 1-9, 2019.

[13] R. Software, "Abstract Syntax Tree (AST)", Available at https://support.roguewave.com/documentation/klocwork/en /10-x/ abstractsyntaxtreeast/, Accessed at 2018.

[14] J. Dean and S. Ghemawat, "Mapreduce: Simplified Data Processing on Large Clusters", *Communications of the ACM*, Vol. 51, No. 1, pp. 107-113, 2008.

[15] Developing Plugins, Available at https://docs.gradle.org, Accessed at 2018.

[16] Jquery, "Jquery: The Write Less, Do More, Javascript Library", Available at http://jquery.com/, Accessed at 2020.

[17] Multi Integrated Development Environment, Available at https://www.ghs.com/products/MULTI_IDE.html, Accessed at 2020.

[18] P. Miller, J. Pane, G. Meter and S. Vorthmann, "Evolution of Novice Programming Environments: The Structure Editors of Carnegie Mellon University", *Interactive Learning Environments*, Vol. 4, No. 2, pp. 140-158, 1994.

[19] S. Davis and G. Kiczales, "Registration-Based Language Abstractions", Proceedings of ACM International Conference on Object Oriented Programming Systems Languages and Applications, pp. 754-773, 2010.

[20] Jet Brains, "How to Check Your Regexps in Intellij Idea 11?", Available at http://blogs.jetbrains.com/idea/tag/regexp/, Accessed at 2019.

[21] Microsoft Magazine, "Custom Design-Time Control Features in Visual Studio.net", Available at http://msdn.microsoft.com/en-us/ magazine/cc164048.aspx, Accessed at 2018.