API Usability: An Empirical Study on Open Source Software

Mohammad Asif A. Khan
Department of Computer Science, University of Saskatchewan, Saskatoon, Canada
mak369@mail.usask.ca

***Abstract*—API usage is one of the major factors in terms of software development and maintenance. It has unparallel contribution towards software maintenance due to code reuse. This also prevents unnecessary increase of software size and complexity. Moreover programmers do not have to code from the scratch for a particular method which is provided by API. The productivity of the developer and the quality of software depends on how effectively and accurately APIs are used where understanding the proper usage comes into play. In this paper an empirical study has been conducted towards finding pattern of API usage in three different domains of compilers, game and text editors each containing different subject systems. A complete list of top APIs with frequencies being used is also identified. The granularity of the API usage such as standalone and block has also been considered.**

Keywords: Stand alone, API, TXL, Parsing

I. INTRODUCTION

With increase in availability of data and growing demand of service software development has become a challenging area. At the same time maintenance of software is also getting more of noticeable concern. Once software is developed and ready for commercial use it needs to be maintained at regular intervals. The maintenance task includes software upgrade, addition of functionality and fixing of bugs. So larger a software is greater the maintenance cost will be. Consequently the question is how large and complex software could be. The answer to this question depends on various factors ranging from type of the system, customer requirement, platform and the implementation style. Previously it was obvious that developing a system required considerable amount of coding from the scratch. However with rapid growth of software development technology and sophisticated programming platforms the task of coding has become very easy and simple. The programmers are able to quickly adapt themselves to the platform and carry on with the development process. One of the major breakthroughs in this technique is the support of interfaces allowing the developers to do coding through code reuse. Such reuse is provided by built-in interfaces known as API (Application Programming Interface). With rapid progress of software engineering reusable components, frameworks, APIs and libraries have been developed. As a result the process of system development has shifted considerably where the programmers can keep the size and complexity of the program small and simple without having to sacrifice the functionality. It has been found in a study [1] that Java Standard API consists of about more than 3000 classes and 20000 methods. The same study has revealed that only about 50% of the classes in the Standard API are used at all, and around 21% of the methods are used.

Large APIs [7] like Microsoft's .NET Framework or the Java APIs have grown to thousands of classes with tens of thousands of methods, and grow larger with each successive release. Previous studies show that Microsoft has created and supported many different application programming interfaces (APIs) that are in wide use today. The .NET framework APIs alone include more 140,000 methods and property fields and are shared by a collection of programming languages including C#, VB.NET and C++.

The usability of API is very important for the productivity of the programmers and software coding. The usability of API is of much talked issue nowadays. Though there are enormous numbers of APIs and libraries packaged within programming language software with diverse functionality there is still need of proper documentation of them for the programmers to understand and use them accurately. The programming skill, efficiency and performance of software depends how optimally a code is used to encompass the desired functionality. There comes the usability issue of API. Knowing how well to use APIs depends how well they are documented. Little research has been on to address the usability of API in terms of difficulty and the extent of the number of APIs used in the development of software. There has been a considerable amount of study and research on design, implementation and so on.

The main aim of this research is to study the extent of the API usage by the developers. The pattern of usage of APIs has also been taken into account where the top APIs are identified based on the frequency of use both as standalone and in blocks of If and While statements. This will help the designers to infer which APIs are mostly used and which of them are not.

Moreover the main reason behind using API is to enhance a programmers' productivity and at the same time increase the quality of the code. The question is does this really happen? The answer to this lies in the study of API usage that is presented in this paper.

The rest of the paper is organized as follows. Section II provides a brief overview of the API. Section III presents the study approach. Section IV describes the experimental setup. Section V describes the outcome of the experiment while Section VI gives an overview of related work followed by conclusion in Sections VII.

II. OVERVIEW

A. What is an API?

Application Programming Interface [23] is a set of commands, functions, and protocols which a programmer use to build an application. It provides a short cut way to the programmers for using the functionalities of the operating system instead of developing the same functions from the scratch to interact with the system. It serves as an interface between two software programs and facilitates interaction between the software unlike user interface which allows interaction between humans and computer. In general an API is a software to software interface. APIs run silently in the background and are completely invisible to the users. The vital role of an API is creating communication between the systems so that the users can get the required functionalities and information from the system.

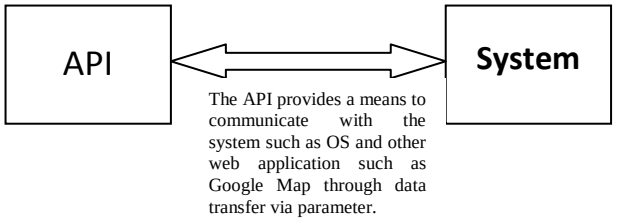

Figure 1: Functionality of API

B. Benefits of API

There are a number of benefits provided by the use of API. They are:

1. Provides an interface to the programmer to access certain functionalities without letting them to code from the scratch.
2. Allows them to save time through code reuse.
3. APIs does not expose the underlying code to the user. As a result change to the other parts of the program does not affect the codes associated with the API
4. APIs provide a media for accessing certain underlying functionalities that are not possible without them. Such functions include access to device drivers, handling interrupt service routine of the operating system, accessing windows registry and so on.

C. Types of APIs[23]

There are four types of API that are in use in software systems. They are:

1. General
2. Specific
3. Language dependent and
4. Language independent

General: These are the APIs that are found as a library to be used in the programming language by the developers. For instance, Java API, Standard Template Library in C++ which is a collection of functions, iterators, algorithms and containers.

Specific: In contrast to general APIs, there are APIs that are meant for a specific function such as google API, facebook API, twitter API and so on. These APIs allow developers to establish communication with other system.

Language Dependent: These are the APIs that are available in a particular programming language and that can be used within the specified language. For example in Pascal the API used for reading and writing from and to the console is “readln()” and “writeln()” while for C the APIs are “scanf()” and “printf()”.

Language independent: These types of APIs are not bound to a particular programming language, process or system rather they are open to be used and called form various programming languages. A developer can use the google API to take advantage of the information regarding maps and other relevant information. Facebook APIs are also very popular which allows developers to create their own applications.

D. Motivating Example of API

```
for (int i = 0; i < listOfFiles.length; i++)
{
  if (listOfFiles[i].isFile())
{
    if (listOfFiles[i].getAbsolutePath().endsWith(".txt"))
 {
Parser parser = new Parser(root,listOfFiles[i].getAbsolutePath());
   parser.parse();
    }
   }
    else if (listOfFiles[i].isDirectory())
    {
    mainFolder (root,folderpath+"\\"+listOfFiles[i].getName) ;
    }
  }
}
```

Figure 2: Example of API

The figure shows APIs that are used in the code. The codes marked in bold are the method calls made using the (.) notation

.

I. STUDY APPROACH

The primary objective of this research is to define a framework for identifying types and number of APIs used in software. Though there has been a lot of work on API design, implementation, migration there is little study done towards API usage except Ma. et. el.[1]. In their study they have identified the top used packages, classes and methods but did not provide any evidence of how and where those APIs are used in the corpus. However in this study a similar study has been done by proposing a framework to locate APIs in three different locations such as If block, Loop block and StandAlone (other than If and Loop block) in four different domains of game, database, compilers and text editors.

A. *Framework of the Study*

To conduct the study a series of steps is required starting from downloading of software till the reporting and analysis APIs contained in the systems. For this a framework for the experiment has been designed which is depicted in figure 1. The framework consists of three major steps a) mark b) extract and c) report.

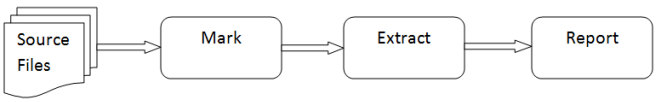

Figure 3: Framework of API study

B. *Mark*

This is the first step of the study of API usage. Here in this phase the source files from the systems are read and the desired APIs are marked accordingly. The marking phase is done using parser developed in TXL. TXL [22] is programming language designed specifically for source code analysis and transformation using rules and functions. A parser is developed with specialized rules, functions and grammar overrides. The desired APIs are therefore marked using the selective semantic markup strategy. TXL also provides some special features that are designed for agile parsing [2]. Each TXL program contains a base grammar based on which the program works. The main idea behind agile parsing is the replacement of non-terminals using the redefine clause.

1) *Grammar Override*

Figure 5 shows the scenario of grammar override [2] that is done for the development of mark phase. From the figure it is seen that reference has been overridden using API and EAPI. The notation (…) represents that previous definition will remain as it is followed by new added non-terminals as required for the task. Moreover the API and EAPI are the definition of the methods calls that we are interested in. Both API and EAPI are defined as [jdbc_name] [method_argument] where jdbc_name signifies the method calls that are to be marked. By method call it is meant that any call that is made using (.) notation. As for example in the code **file.getName(),** getName() is the method call that needs to be marked. So in this case any call followed by (.) notation must be marked which are identified as components with [component] notation. This [component] defines all the method calls contained within the (.) notation.

2) *Granularity*

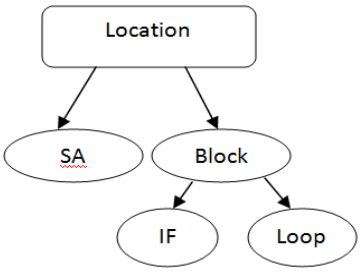

Figure 4: Granularity of Study

Granularity here means the location of the API we are interested in. The location in figure 4 is divided into SA and Block where SA stands for Stand Alone and Block in this case means code that contains IF and Loop. In this study we have considered APIs only in IF and While block.

Figure 6 depicts the code for the grammar overrides done to mark only those method calls within the If and the while block. Both the IF and the While definitions are overridden using redefine clause.

C. *Extract*

Extract is the second phase of the API detection. In the first phase the source files are marked using TXL. The marked files are then fed into the extractor which scans them and creates another file containing the extracted APIs, file name, file number and the location. The files are then further analyzed to study the use of the API is different systems. Figure 7 shows the output file generated during the mark phase for the extractor. All the method calls that are stand alone and within the If and while block are marked. The standalone APIs are marked using XML tag <API> and the blocked API are marked similarly using XML tags <EAPI>. The extractor then reads only those lines that are tagged and reports the method calls within it. The scenario of the extracting API is shown in the figure of the example from the code.

```
redefine reference
    ...
  | [API]
  | [EAPI]
end redefine

define API
    [jdbc_name] [method_argument]
end define

define EAPI
    [jdbc_name] [method_argument]
end define

define jdbc_name
    [component]
end define

define expression
    [assignment_expression] [API]
  | [assignment_expression] [EAPI]
  | ...
end define
```

Figure 5: Example of Grammar Override

```
redefine if_statement
    'if '( [expression] ')
    [guarded_statement]
    [opt else_clause]
end redefine

redefine else_clause
    'else
    [guarded_statement]
end redefine

redefine while_statement
    'while '( [expression] ')
    [guarded_statement]
end redefine
```

Figure 6: Example of Grammar Override for If and While Block

```
try {
    System.out < API >.print ("  Creating Statement...\n") < / API >;
    if (i == 9) {
        y = con < API >.createStatement () < / API >;
        System.out < EAPI >.print ("  Opening ResultsSet...\n") < / EAPI >;
        rset = stmt < API >.executeQuery (queryString) < / API >;
    } else {
        System.out < EAPI >.println () < / EAPI >;
        System.out < EAPI >.println ("  Row [" + ++ counter + "]") < / EAPI >;
        System.out < EAPI >.println ("  ----------------------") < / EAPI >;
    } int counter = 0;
    while (rset < API >.next () < / API >) {
        System.out < EAPI >.println () < / EAPI >;
        System.out < EAPI >.println ("  Row [" + ++ counter + "]") < / EAPI >;
        System.out < EAPI >.println ("  ----------------------") < / EAPI >;
        System.out < EAPI >.println ("      Name             -> " + rset < API >.getString (1) < / API >) < / EAPI >;
        System.out < EAPI >.println ("      Date of Hire     -> " + rset < API >.getString (2) < / API >) < / EAPI >;
        System.out < EAPI >.println ("      Monthly Salary   -> " + rset < API >.getFloat (3) < / API >) < / EAPI >;
    } System.out < API >.println () < / API >;
    System.out < API >.print ("  Closing ResultSet...\n") < / API >;
    rset < API >.close () < / API >;
    System.out < API >.print ("  Closing Statement...\n") < / API >;
    stmt < API >.close () < / API >;
} catch (SQLException e) {
    e < API >.printStackTrace () < / API >;
```

Figure 7: Example of a File Marked for Extraction

D. Report

This is the final and the crucial step of the API study. In this phase the following are measured to determine the API usage:

- Frequency of API usage in total: Denotes the number of times a particular API appears in the systems
- Frequency of API usage in Blocks: denotes the number of times a particular API is used in If and While blocks
- Frequency of API usage in SA: denotes the number of times a particular API is used in locations outside the blocks.

Table 1: Top APIs with Frequency

Text Editor		Game		Compilers	
API	**Frequency**	**API**	**Frequency**	**API**	**Frequency**
.add	8635	.add	807	.append	1217
.append	2055	.println	287	.equals	535
.equals	1713	.equals	274	.println	526
.put	1398	.drawImage	168	.output	133
.println	1344	.get	165	.acaoSemantica	110
.toString	878	.getImage	147	.elementAt	107
.get	791	.addActionListener	141	.addPrimop	106
.length	642	.read	134	.toString	106
.print	598	.getForegroundCharAt	122	.size	104
.write	578	.setText	104	.proximoToken	97
.setText	578	.close	94	.insereSintaxe	93
.getProperty	474	.setColor	92	.typeCheck	88
.close	468	.getWidth	89	.substring	84
.getText	467	.printStackTrace	87	.translate	78
.charAt	411	.append	85	.push	70
.substring	410	.toString	82	.add	64
.setEnabled	402	.setLayout	82	.getLinha	63
.printStackTrace	398	.drawString	81	.setUltimaAcaoSintatica	61
.size	396	.setMnemonic	81	.length	60
.getName	395	.isAllSetProperty	71	.getInstructionList	60
.getString	392	.setBackground	70	.markNeeded	58
.apply	366	.substring	67	.setNeeded	58
.testApplyRule	353	.getHeight	65	.nextToken	50
.parent	350	.setFont	62	.Evaluate	48
.addActionListener	344	.setVisible	61	.addElement	46
.setProperty	319	.size	61	.identicalTo	46
.next	317	.getPNG	59	.put	41
.hasNext	305	.setBounds	57	.get	38
.getMessage	293	.getResource	57	.getConstantPool	36
.make	288	.getString	55	.charAt	36
.read	283	.redrawArea	51	.getUnsignedInteger	35
.iterator	259	.createImage	51	.newLabel	35
.indexOf	252	.debug	48	.Error	34
.debug	252	.setSize	48	.run	32
.arraycopy	249	.length	47	.readShort	29
.remove	245	.showMessageDialog	46	.isNeeded	29
.setLayout	219	.writeBytes	46	.write	28
.setFont	214	.parseInt	46	.pushBack	27
.setBorder	212	.nextInt	45	.remove	27

Table 2: Top APIs with Frequency in Location

Text Editor				*Game*				*Compilers*		
API	**SA**	**Block**		**API**	**SA**	**Block**		**API**	**SA**	**Block**
.add	5721	2914		.add	96	711		.append	154	1063
.append	1033	1022		.println	79	208		.equals	0	535
.equals	0	1713		.equals	11	263		.println	182	344
.put	258	1140		.drawImage	69	99		.output	27	106
.println	589	755		.get	18	147		.acaoSemantica	63	47
.toString	0	878		.getImage	4	143		.elementAt	0	107
.get	0	791		.addActionListener	7	134		.addPrimop	0	106
.length	0	642		.read	3	131		.toString	0	106
.print	311	287		.getForegroundCharAt	0	122		.size	0	104
.write	253	325		.setText	24	80		.proximoToken	79	18
.setText	198	380		.close	25	69		.insereSintaxe	0	93
.getProperty	0	474		.setColor	29	63		.typeCheck	16	72
.close	117	351		.getWidth	9	80		.substring	0	84
.getText	13	454		.printStackTrace	17	70		.translate	34	44
.charAt	0	411		.append	19	66		.push	2	68
.substring	0	410		.toString	11	71		.add	13	51
.setEnabled	166	236		.setLayout	6	76		.getLinha	0	63
.printStackTrace	88	310		.drawString	22	59		.setUltimaAcaoSintatica	0	61
.size	0	396		.setMnemonic	1	80		.length	0	60
.getName	0	395		.isAllSetProperty	1	70		.getInstructionList	0	60
.getString	0	392		.setBackground	10	60		.markNeeded	51	7
.apply	344	22		.substring	1	66		.setNeeded	35	23
.testApplyRule	0	353		.getHeight	7	58		.nextToken	29	21
.parent	350	0		.setFont	9	53		.Evaluate	10	38
.addActionListener	17	327		.setVisible	29	32		.addElement	25	21
.setProperty	58	261		.size	15	46		.identicalTo	0	46

I. EXPERIMENTAL SETUP

In this section we provide a brief overview of the systems we have studied,

A. Subject Systems

The table 3 shows the summary of the domains and the number of subject systems. We studied four different domains of Text Editor containing 32 subject systems, Game with 31 subject systems, Compilers with 13 systems. Since the systems are coded in a variety of languages, only Java files have been considered in the study.

Table 3: Subject Systems

Domain	**No. of Systems**	**No. of Java Files**	**Line of Code**	**Total Size**
Text Editor	32	6917	1364917	366M
Game	31	1877	322227	177M
Compilers	13	863	294236	36M

The downloaded subject systems were coded in a mixture of different programming languages. Since we are interested to study only on Java files, the number of files and lines of code contained with each system were estimated using a customized script cloc-1.50.pl coded in python. It was observed that 64 projects contained files coded in Java with 10007 number of files and 2396071 lines of code. On further analysis it was found that 15 projects contained Java files less than 10. Those systems were ultimately removed from the study bringing down the number to 41. Similarly by applying the same technique of filtration on text editor, game and compilers we arrived at a lower number of systems of 32, 31 and 13 within each domain respectively.

I. RESULT

This section will provide results of the experiment on three different domains of compilers, game and text editors

The table 1 and table 2 shows the statistics of the frequency of APIs used in the systems of the domain compilers, game and text editors. From the table 2 it can be seen that append, equals, add, put, println, tostring, get, length are the APIs with the highest frequency. In case of game add, println, equals, drawimage, get, getimage are the APIs with greater frequency while for the compilers append, equal, println, output, tostring are the major used APIs.

Table 2 also shows the similar picture of the APIs used as standalone and in block. In case of Compilers it can be seen that much of the API usage takes place in the block (in the If and While statements) rather as a standalone. If we have a closer look at the text editor it can be observed that few APIs are used as standalone while all of them are used in the blocks with high frequencies. Moreover method calls like equals, toString, get, length, getProperty, charAt, substring, size, getName, testApplyRule are not used at all. The game, on the other hand shows different pattern of use. Here all of the method calls are used, but like others the frequency count shows that most of the API usage is concentrated towards the block.

II. RELATED WORK

There have been a good number of studies in API usability which can be classified as research in design issues, tool support, API migration, API usability measurement techniques.

Stylos et el. [15] in their research concentrated on the API design decisions. They have focused on defining what are the factors that must be considered in designing a good and powerful API. They have also proposed a mapping of API design decision space with the API quality attributes and identified different metrics for arriving at the decision. In another research Stylos et el. [14] raised the issue of combing the initial and the new users requirement for redesigning APIs. They performed a case study on SAP BRPlus which is a business rule engine. They did a usability evaluation on a user-centric design of API wrapper to assess the value of addressing specific use cases by the application-level developers. Similarly they also carried a comparative study on the usability of API design by considering the fact on the programmers' preference on using APIs with parameters or the one without parameters. On experiment they concluded that tough it was hypothesize that APIs with parameters are preferable but practically more preference was towards APIs without parameters.

Bartolumoi et el. [16] studied API migration between two different XML API. They have applied the process of API migration on two different Java platform XML APIs. These APIs were further investigated to find the differences with measurement and identify of classification based on usability.

Kawrykow et el. [17] did extensive research on the effective use of API. They identified the fact that though APIs are intended for simplicity and code reuse, however in some cases the programmers are not using the APIs built in the software package. They rather try to re invent the code for API and use them in their development as method. Such scenarios are defined as inefficient API usage. To point out this they have created a tool which can automatically detect such inefficient API pattern. On applying their tool on Java projects they found that around 4000 cases were among the victims of inefficient API usage that required improvement.

A good use of API by developers relies on how well the APIs are documented. Dekel et el. [19] did similar research on improving the documentation to convey the hidden information in it to the interested reader which may get ignored within the long text. They have built an eclipse plug-in called eMoose to highlight those hidden directives.

Feilkas et el. [18] worked on identifying whether a API client complies with the API developers' assumptions of the domain abstraction. They developed a framework for expressing the assumptions that restricts the clients in not complying with the expectations of the provider.

Henning [3], Arnold [4] pointed out that creating useful API depends on API design and human factors are also a major factor in it. Bloch [6] and McLellan [5] suggested a number of guidelines for building a good and usable API. On the other hand Ellis et el. [20] conducted a comparative study on the usability of factory patterns and the constructors on the instantiation of object. They found that factory patterns are harmful to API usability and concluded that more time is required by the user to construct object with factory.

Measuring API usability has also of great importance where Clarke and Becker [9] defined 12 cognitive dimensions for measuring usability of API while Bore et el.

[8] contributed in proposing 7 different measures in order to profile API usability.

Wu et el. [21] proposed a tool called CoDecent which can automatically link API documents and create a diagram for the users to understand the use of API. MAPO is another tool developed by Xie et el. [10] which mines API usage pattern while Stratcona [12] is another specialized tool that can aid in example matching between API source code and the source code repository containing API. Catch-up [11] and Diff Catch-up [13] provides support to the users during the code change caused during the evolution of the APIs contained in the code.

III. CONCLUSION

In this paper a study of API usage on three different subject domains of compilers, games and text editors have been done. To assist in the study process a framework for marking the API from the source file followed by extraction and reporting has been proposed. A listing of top 20 APIs used both as standalone and in blocks (If and While) has been made. From here it can be inferred which APIs are used most and in which location. In future more extensive study will be conducted to identify the packages and classes associated with the APIs. A more automated framework will be presented to report all associated APIs from the repository. Moreover a study on the API usage pattern will also be made on different releases of software. This will allow determining the API change pattern among the releases.

IV. REFERENCE

[1] H. Ma, R. Amor, E. Tempero. Usage Patterns of Java Standard API. In proceedings of 13th Asia Pacific Software Engineering Conference (APSEC '06).

[2] J .R. Cordy, "Generalized Selective XML Markup of Source Code Using Agile Parsing", Proc. IWPC 2003, IEEE 11th International Workshop on Program Comprehension, Portland, Oregon, May 2003, pp. 144-153.

[3] K. Arnold. Programmers are people, too. Queue, 3 (5): 54-59, 2005.

[4] M. Henning. API design matters. Queue, 5 (4): 24-36, 2007.

[5] S. McLellan, A. Roesler, J. Tempest, and C. Spinuzzi. Building more usable APIs. Software, IEEE, 15 (3): 78- 86, May/Jun 1998.

[6] J. Bloch. How to design a good API and why it matters. In OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pages 506-507, New York, NY, USA, 2006. ACM.

[7] J. Stylos and S. Clarke. Usability implications of requiring parameters in objects' constructors. In ICSE'07: Proceedings of the 29th International Conference on Software Engineering, pages 529-539, Washington, DC, USA, 2007. IEEE Computer Society.

[8] C. Bore and S. Bore. Profiling software API usability for consumer electronics. Consumer Electronics, 2005. ICCE. 2005 Digest of Technical Papers. International Conference on, pages 155-156, 8-12 Jan. 2005.

[9] S. Clarke and C. Becker. Using the cognitive dimensions framework to evaluate the usability of a class library. In Joint Conf. EASE & PPIG, Petre & D. Budgen (Eds), pages 359-366, 2003

[10] T. Xie and J. Pei. MAPO: mining API usages from open source repositories. In MSR '06: Proceedings of the 2006 international workshop on Mining software repositories, pages 54-57, New York, NY, USA, 2006. ACM.

[11] J. Henkel and A. Diwan. CatchUp!: capturing and replaying refactorings to support API evolution. In ICSE '05: Proceedings of the 27th international conference on Software engineering, pages 274-283, New York, NY, USA, 2005. ACM.

[12] R. Holmes, R. Walker, and G. Murphy. Approximate structural context matching: An approach to recommend relevant examples. Software Engineering, IEEE Transactions on, 32 (12): 952-970, Dec. 2006.

[13] Z. Xing and E. Stroulia. API-evolution support with DiffCatchUp. Software Engineering, IEEE Transactions on, 33 (12): 818-836, Dec. 2007.

[14] J. Stylos, B. Graf, D. K. Busse, C. Ziegler, R. Ehret, J. Karstens. A Case Study of API Redesign for Improved Usability. IEEE Symposiium on Visual Languages and Human-Centric Computing, 2008. VL/HCC 2008. Pages 189-192

[15] J. Stylos, B. Myers. Mapping the Space of API Design Decisions. 2007 IEEE Symposium on Visual Languages and Human-Centric Computing

[16] T. T. Bartolomei, K. Czarnecki, R. L¨ammel, and T. van der Storm. Study of an API migration for two XML APIs. In Postproceedings of Software Language Engineering (SLE 2009). LNCS, Springer, 2010.

[17] D. Kawrykow and M. P. Robillard. Detecting Inefficient API Usage. Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on

[18] M. Feilkas and D. Ratiu. Ensuring Well-Behaved Usage of APIs through Syntactic Constraints. Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on

[19] U. Dekel and J. D. Herbsleb. Improving API Documentation Usability with Knowledge Pushing. In proceedings of ICSE'09, May 16-24, 2009, Vancouver, Canada

[20] B. Ellis, J. Stylos, and B. Myers. The factory pattern in API design: A usability evaluation. In ICSE '07: Proceedings of the 29th International Conference on Software Engineering, pages 302-312, Washington, DC, USA, 2007. IEEE Computer Society.

[21] Y. Wu, L. W. Mar, H. C. Jiau. CoDocent: Support API Usage with Code Example and API Documentation. In Proceedings of 2010 Fifth International Conference on Software Engineering Advances.

[22] TXL- www.txl.ca

[23] http://en.wikipedia.org/wiki/API