An empirical study on the adoption of C++ templates:

Library templates versus user defined templates

Di Wu Lin Chen Yuming Zhou Baowen Xu

State Key Laboratory for Novel Software Technology at Nanjing University, Nanjing, China

nju.wudi@gmail.com lchen@nju.edu.cn zhouyuming@nju.edu.cn bwxu@nju.edu.cn

***Abstract*—Template is a powerful language feature to develop reusable programs, which has long been widely used by C++ programmers. Among various templates, library templates play a key role, as they provide programmers fault-free generic units. But to our surprise, little work has been done to study how these templates are adopted in real software. In this paper, we conduct an empirical study on the adoption of templates to understand how library templates influence programming compared with user defined templates. To this end, we use the historical revisions of ten open source systems, containing 34 million lines of C++ code, to conduct the experiment. The experimental results show that: (1) the growth of using library templates does not result in a substantial decrease of using user defined templates; (2) eight templates account for over 95% of all library template uses; and (3) user defined templates tend to be derived more frequently than library templates. Based on these results, we make two practical suggestions. For C++ novices, we suggest that they focus on learning the eight most commonly-used library templates. For library developers, we suggest that they develop more library templates that can be easily derived.**

Keywords- Programming Language, C++, Generic Programming, Templates, Empirical Study

I. INTRODUCTION

C++ template is a representative generic programming feature for developing highly reusable and efficient software components [1]. In the last decades, C++ experiences a long way to enhance the role of templates in generic programming by continuously supplementing new template libraries. In C++ 11 specification, more than one-third contents are devoted to standard template libraries [3]. Library designers hope that these well-designed library templates could enable common programmers to use general components at a higher level of abstraction rather than developing all code from scratch [5]. Due to the safety of library templates, they are often used as the basic elements to create compound data structures in a type safe manner.

Other than allowing directly adopting library templates in programming, C++ also supports programmers to create their own generic units, namely user defined templates. However, with the expansion of C++ libraries, library templates are more powerful and can be replacers of elementary user defined templates in more and more scenarios. In other word, a lot of generic units that were needed to be implemented by common programmers are now supported in new C++ libraries. For instance, smart pointers, which are frequently used to avoid dangling pointers and resource leaks, were always manually created by programmers in the past. However, C++ 11 [3] has formally introduced two new library templates for smart pointers, namely class std::shared_ptr for a pointer that implements the concept of shared ownership, and std::unique_ptr for a pointer that implements the concept of exclusive ownership or strict ownership [3]. Together with std::auto_ptr provided in C++ 98, these three kinds of smart pointer templates can now satisfy programmers' requirements in using smart pointers in most scenarios.

Since creating user defined templates brings heavy workload and are error-prone, programmers are recommended to make more use of library templates when they need [6]. Although most C++ programmers know the necessity of using library templates, there still exists a great amount of user defined templates in C++ programs. In view of this fact, there is a strong need to investigate how programmers apply templates and why they use or not use library templates. Surprisingly, few studies have been done in this area. This may lead to two problems. For library designers, they cannot know whether libraries they developed really benefit users in their programming work and which libraries are most welcomed by users. For common programmers, especially the novices, they do not understand which libraries are most helpful to them and need their attention to be paid on.

In this paper, we aim to empirically investigate how library templates and user defined templates are adopted in real software. The purpose of this is to understand how library templates influence programming compared with user defined templates. More specially, we want to investigate the following three questions:

- Whether more uses of library templates result in a decrease of using user defined templates?
- Whether programmers tend to use a few library/user-defined templates more frequently?
- Whether library templates or user defined templates are more frequently to be derived?

To answer these questions, we analyze the historical revisions of ten open source systems, containing 34 million lines of C++ code. The experimental results show that: (1) the growth of using library templates does not result in a substantial decrease of using user defined templates; (2) eight templates account for over 95% of all library template uses; and (3) user defined templates tend to be derived more frequently than library templates. Based on these results, we make two practical suggestions. For C++ novices, we suggest that they focus on learning the eight most commonly-used library templates. For library developers, we suggest that they develop more library templates that can be easily derived.

The rest of the paper is organized as follows. Section II introduces library templates and user defined templates in a nutshell. Section III describes the research questions, data sets, and analysis methods. Section IV reports the experimental results. Section V discusses related work. Section VI concludes the paper and outlines the direction for future work.

II. Templates and Their Usages: Library versus User Defined

Library templates are defined in C++ libraries, which include STL [7], Boost [23], Loki [24] and ATL [25], etc. The most well-known one, STL, is the standard template library recognized by the language specification [3]. It provides a large collection of classes that meet all kinds of needs, together with several algorithms that operate on them. Since all components of the STL are templates, they can be used for arbitrary element types. Containers, iterators, and algorithms are three key components in STL [5]. By contrast, user defined templates are explicitly defined in programs and are utilized by programmers themselves. These templates are used to implement functionality which is relatively specific to users' requirements. Generally, user defined templates are not as universal as library templates.

Even though library templates and user defined templates are defined in different sources, they have same ways of use. Both library templates and user defined templates can be divided into function templates and class templates. Here, class templates also include struct templates [4]. The way to use function templates is to invoke them as ordinary functions [4], resulting in a function instance called **specialized function**. Compared with function templates, more ways are available for using class templates. According to C++ language specification [3], class templates can be initialized, specialized, or derived. Here we demonstrate these concepts by excerpting the real usages of class templates in TortoiseSVN:

- Instantiation means to create a class instance by associating template parameters with arguments. The most commonly used template arguments are non-template types. For example:

```
quick_hash<CHashFunction> index;
```

This statement denotes to declare an object *index* with a class type *CHashFunction*, which is an instance of the user defined template *quick_hash*.

- Specialization, which can be created by a declaration introduced by "template<>", denotes to establish a separate implementation for a template. For example:

```
template <>
struct ice_not<true>
{
  BOOST_STATIC_CONSTANT(bool, value = false);
};
```

This is an example to provide the template *ice_not* with a specific implementation through specialization. All type instances with argument "true" will be instantiated from this special implementation.

- Derivation indicates to define a child template by explicitly extending the parent class template. For example:

```
template<class B, STREAM_TYPE_ID type>
class CInStreamImpl :public
CInStreamImplBase< CInStreamImpl<B, type> , B , type>
```

In the example, *CInStreamInmlpBase* acts as a parent template, and *CInStreamImpl* derives it in the public manner.

III. Research Method

In this section, we first formulate the three research questions about the adoption of templates. Then, we introduce the data sets used for investigating these research questions. Finally, we describe the analysis method.

A. Research Questions

We aim to compare the adoption of library templates to user defined templates in real software. The three research questions (RQ) are hence formulated as follows.

- **RQ 1:** *Will there be a decrease of adopting user defined templates with an increase use of library templates?* C++ library templates can help programmers get rid of hard work in constructing various elementary user defined templates. In other word, if library templates are more widely used, more user defined templates that have similar functionalities with existing library templates are unnecessary and thus can be replaced by using these library templates.
- **RQ 2**: *Are library/user-defined templates biasedly used? If so, which templates are the most commonly-used ones?* With so many library templates available, we intuitively conjecture that elementary library templates may be used more frequently. If this hypothesis is proven to be true, we can pick out the libraries containing commonly-used templates and give advices to programmers about their focus on these library templates.
- **RQ 3**: *Are library class templates less often derived than user defined class templates?* Similar to ordinary classes, class templates can also be used in the manner of derivation, which is an important feature in object oriented programming. If the answer of RQ3 is "Yes", it may indicate that user defined templates are more easily to be inherited than library templates. Further, there will be potential opportunities for library developers to produce more extensible templates to improve libraries' ability in supporting object oriented programming.

B. Data Sets

To investigate the proposed research questions, we analyze 10 projects of different categories listed on *http://sourceforge.net*. These projects are well logged and long aged, ranging from 4 to 15 years old. Due to the vast historical

revisions of the projects, it is too costly to investigate all of them. Additionally, some dump versions may also impact the result. Therefore, we regularly select the last revisions without dump in each season, that is, four revisions for a project each year since it started to be released. Finally, 317 historical revisions of the 10 projects are chosen. They totally contain over 34M lines of C++ code. Source files of these projects are obtained from open source repositories by using *svn* and *git clone* tools. The detailed information about these projects is listed in Table I.

TABLE I. OPEN SOURCE PROJECTS IN THE STUDY

Project	Age	C++ KLOC	Category
TortoiseSVN	11	405.806	Software Development
FileZilla	10	82.941	Communication
KeePass	4	56.364	Security & Utilities
MeshLab	9	247.556	Graphic
SAGA GIS	10	257.530	Science & Engineering
Bitcoin	4	66.359	Business & Enterprise
Console	7	20.959	System Administration
Cool Reader	7	232.239	Home & Education
Warzone 2100	9	180.517	Games
VLC	15	119.801	Audio & Video

C. Analysis Method

We analyze source code of the target systems by the following steps. At the first step, filter non-C++ source files. Our target files are C++ header files and source files. To filter out those redundant ones, we use the *C++ Strict* option in *Understand* and build an udb database for each selected revision. At the second step, detect the uses of template in C++ source files. To process the udb databases, we write Perl scripts invoking *Understand* APIs to find all program points related to templates. If a template definition is found, then the template is identified as a user defined template. If the namespaces like "std" and "boost" are found, then the template is recognized as a library template. At the third step, check the ways of template uses. In this step, we run a Perl script to check out different usages of all library function/class templates. The uses of user defined templates are examined with the same approach. At the fourth step, store the data. We write a Java program to automatically process the output of running Perl scripts and store the final data in xls files. We choose xls as the file format because they are easily to do statistical analysis and to plot figures.

IV. EMPRICAL RESULTS

In this section, we report in detail the experimental results and discuss the possible threats to the validity of our study.

A. RQ1: Will there be a decrease of adopting user defined templates with an increase use of library templates?

Based on the data obtained, we firstly measure the correlation between the density of library template uses and the density of user defined template uses. Here, density is calculated as the number of template uses divided by C++ KLOC. We choose the density (instead of the number of template uses) as the evaluation indicator because it reflects the frequency of template uses by eliminating impacts brought by the increase of code size of projects. In our context, one use of template is either a call to a function template or an instantiation, specialization, and derivation of a class template.

If an increase use of library templates causes a decrease in using user defined templates, a negative relationship will be shown by the two groups of data. To see if this is true, we use both observational and statistical methods. An observational result related to RQ 1 is shown in Figure 1. The figure plots the density of library template uses (red lines) versus the density of user defined template uses (blue lines). Inspecting both sub-figures for function template uses and class template uses, we cannot find a clear evidence supporting an increase of the density of using library templates results in a decrease of the density of adopting user defined templates. By contrast, we observe synchronous fluctuations of the two lines.

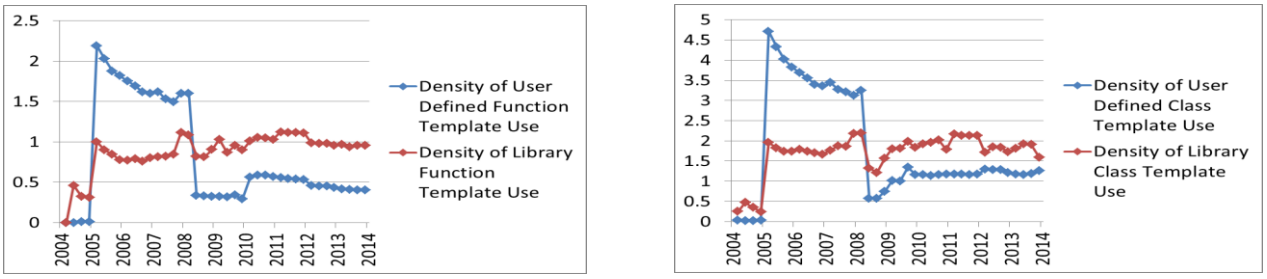
To precisely validate the relevance between the two densities, we use the Spearman Rank Correlation coefficient. Table 2 summarizes for each project the Spearman Rank Correlation coefficient between the density of library template uses and the density of user defined template uses. More specifically, the "Function Templates" column reports the correlation between library function templates and user defined function templates. The "Class Templates" column reports the correlation between library class templates and user defined class templates.

TABLE II. SPEARMAN RANK CORRELATIONS BETWEEN ADOPTION OF LIBRARY TEMPLATES AND ADOPTION USER DEFINED TEMPLATES

Project	Coefficient of Adoption of Templates (p-value)	
	Function Templates	Class Templates
TortoiseSVN	0.16 (0.31)	0.25 (0.12)
FileZilla	-	0.78 (0.00)
KeePass	-0.60 (0.02)	0.85 (0.00)
MeshLab	0.71 (0.00)	0.85 (0.00)
SAGA GIS	0.77 (0.00)	-0.33 (0.06)
Bitcoin	0.72 (0.00)	-0.18 (0.45)
Console	0.31 (0.13)	0.14 (0.50)
Cool Reader	0.81 (0.00)	0.17 (0.38)
Warzone 2100	0.49 (0.00)	0.68 (0.00)
VLC	0.45 (0.00)	0.71 (0.00)

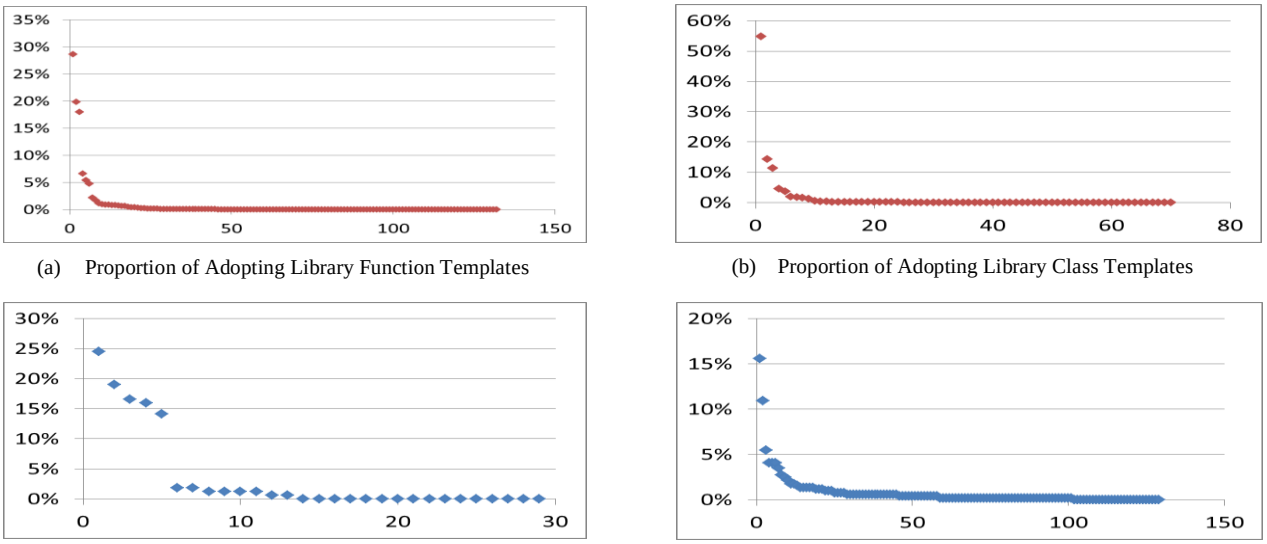
In the function templates group, seven of ten projects show significant results (p-value < 0.05). The only outlier is FileZilla, whose use number of user defined function templates is zero, thus the correlation value fails to be produced. Specifically, six of ten projects show positive relationships, including four high positive correlations (above 0.7) and two mild positive relationships (0.4 to 0.7). Only one project shows a moderate negative correlation (-0.60). To summarize, the data for function template uses generally shows that the tendencies of adopting library function templates and user defined function templates follow the similar pattern, indicating that an increase of library template uses does not lead to a corresponding decrease of user template uses when the templates are applied to create specialized functions. Regarding the adoption of class templates, the data basically does not validate strong relationships. To be specific, among the ten projects, only five

show a significant correlation ($p < 0.05$). The five significant results, however, all indicate relatively positive relevance, with four of them showing high positive relationships (above 0.7) and one showing mild positive correlation (0.4 to 0.7). Due to the low ratio of significances, this group of data cannot testify bound relevance between use density of library class templates and use density of user defined templates. Nevertheless, the consistent positive relationships shown by five valid samples can still demonstrate that with number of library class template uses rising, the number of adopting user template uses does not reduce.



(a) Adoption of Function Templates (b) Adoption of Class Templates

Figure 1. Density of Adopting User Defined Templates vs. Density of Adopting Library Templates *(the case in TortoiseSVN)*



(a) Proportion of Adopting Library Function Templates (b) Proportion of Adopting Library Class Templates

(c) Proportion of Adopting User Defined Function Templates *(TortoiseSVN)* (d) Proportion of Adopting User Defined Class Templates *(TortoiseSVN)*

Figure 2. Proportion of Adopting Different Templates *(Figure c and Figure d reflect adoption of user defined templates in TortoiseSVN)*

Overall, the function templates data group strongly indicates that **the increase of library template uses does not lead to the reduction of user defined template uses**, and the class templates data group validates this result slightly weaker. We deduce the reason for this fact lies in the effects of some irreplaceable user defined templates. That is, even though library templates (such as STL, Boost, and ATL) play an important role in programming, they are only elementary components to construct compound data structures but cannot replace user defined templates. Meanwhile, user defined templates are always needed in specific scenarios and sometimes more frequently used than library templates.

B. *RQ2: Are library/user-defined templates biasedly used? If so, which templates are the most commonly-used ones?*

Figure 2 shows the result for RQ 2. In the sub-figures, each red dot represents a library template, while each blue dot represents a user defined template. For every dot, the x-value is its rank among all templates in its group sorted by the numbers of their uses in descending order, while its y-value is the percentage calculated as the number of its uses divided by the total template uses. In our initial inspection, the dots basically distribute as a power curve. To precisely test our observation, we apply power functions to fit the dots and finally obtain good fitness (a large R^2 value) for all sub-graphs. The fitting result demonstrates the use rates of templates appear in power distributions, showing that **both library templates and user defined templates are high biasedly used**.

The result for RQ 2 signifies the adoption of a small proportion of templates accounts for a majority of overall template uses. Based on this knowledge, we pick out the most popularly-used library templates from the experimental data and manually query which libraries they belong to. Table III lists the names of these libraries and their proportions in use.

Here, we list eight libraries whose proportions are larger than 1%. These libraries totally account for over 95% of all library template uses.

TABLE III. MOST COMMONLY-USED TEMPLATE LIBRARIES

Library Name	Number	Proportion
Standard Container: Vector	1586	54.80%
Standard Container: List	416	14.37%
Standard Container: Map	328	11.33%
Standard Utility: Smart Pointers	152	5.25%
Standard Container: Set	132	4.56%
Standard Utility: Pair	58	2.00%
Language Support Library: Numeric Limits	49	1.69%
Standard Container: Deque	45	1.55%
Total	**2766**	**95.58%**

On account of this result, **we recommend C++ novices to focus on learning how to use the template libraries listed in Table III**. The reason is that these libraries are statistically proved to be frequently used by practitioners in developing open source projects and maybe helpful in their understanding of templates. For library developers, **we advise them to pay their attention on the safety and compatibility of these most frequently-used template libraries when they update C++ libraries**. The reason is that the changes or defects in these libraries may result in severe impacts on programs adopting these library templates.

C. *RQ3: Are library class templates less often derived than user defined class templates?*

As stated in Section 3, class templates are mainly used in three different ways: Instantiation, Specialization, and Derivation. Since RQ 3 focuses on derivation, we calculate the proportion of derivations in class template uses. Table IV reports the proportion of derivations among class template use. Among three kinds of usages of class templates, derivations are more likely to appear in the uses of user defined class templates. This is true for eight of ten projects. The average data also support this conclusion, with a proportion of 7.64% (derivations of user defined class templates) over 0.54% (derivations of library class templates). The result indicates that **compared with library templates, user defined templates are more frequently to be inherited to define child templates**.

Due to the poor ratio of derivations of library class templates, **we recommend library designers to develop more library templates that can be easily inherited to declare new class templates**, especially some elementary components which are easily to be used as base classes. In this manner, library templates could play more important role in object oriented programming and better facilitate programmers.

D. *Threats to Validity*

External Threats. Due to well-known accessibility reasons, our study only investigates open source projects. The research questions investigated in this paper may not compatible with regularities of template uses in industrial software, as different ways of managing software development probably make a difference in adoption of templates. Moreover, even though our study is performed on the historical revisions of 10 different kinds of projects, all of them are application software. Since no system software is included, the results may be skewed to some extent. Finally, since covering all historical revisions for a project is a hard work, we select representative revisions for each project. However, in this way, we probably miss several revisions that are milestones in the process of project development.

Internal Threats. The internal threat lies in the inclusion of test code in the open source projects. The test code may influence the accuracy of code size calculation. However, test code only accounts for a small proportion of the total code size. Therefore, we believe that our results are reliable.

TABLE IV. PROPORTION OF DERIVATIONS AMONG CLASS TEMPLATE USE

Project	Library	User Defined
TortoiseSVN	1.23%	**5.46%**
FileZilla	**0.27%**	0%
KeePass	0.99%	**16.72%**
MeshLab	1.12%	**5.16%**
SAGA GIS	0.97%	**10.39%**
Bitcoin	**0.315%**	0%
Console	0%	**0.84%**
Cool Reader	0%	**2.74%**
Warzone 2100	0%	**0.12%**
VLC	0.53%	**35%**
Average	0.54%	**7.64%**

V. RELATED WORK

Existing studies about C++ templates mainly focused on STL. For example, Josuttis [5] comprehensively summarized the adoption of STL in implementing an amount of useful libraries. Austern [6] studied in depth on the design ideas and infrastructures of STL and discussed the detailed specifications of STL. Other studies on C++ templates include how to refine design of built-in templates among the C++ standard [13], how to identify idiom usages of generic libraries [14], how to effectively compile templates at runtime [15], and how to compare C++ templates with the generic features in other languages [16]. The only empirical study about adoption of C++ template was undertaken by Basit *et al.* [8]. They conducted a case study to observe whether templates can reduce clone code in STL. As a result, they found out many code duplications that cannot be eliminated by templates.

With more and more open source codes available through the Internet, researchers have been gradually starting to make use of empirical techniques such as data mining to access useful information about language adoption. These techniques can not only help to understand adoption of different languages, but also supply objective evidence to evaluate features in various languages. Existing studies in this area include predicting future trends of language evolution by comparative study among languages [17], mining software repositories to investigate language features [11] and language adoption [18] [19] [20], and understanding language elements

with data mining techniques [21] and artificial experiments [22].

Among all previous studies, the work done by Parnin *et al.* [9] [10] is most related to our work. They deeply investigated the adoption of Java generics in open source projects. The papers report that: (1) Use of Java generics sometimes shows negative relationship with the number of type casts; (2) Java generics can prevent some code duplications; (3) Generics are usually adopted by several contributors in a project rather than all committers; (4) Developers do not always convert raw types to generic types; (5) Different abilities of IDE in supporting generics do not influence the adoption of generics.

Due to distinct characteristics between C++ and Java, it is improper to investigate C++ templates by following the same way in studying how Java generics are adopted. The reasons are two-fold. First, Java has a super-type "Object", which is always applied to create generic data structures before generics are introduced. However, C++ does not have such a super-type. Therefore, we cannot study conversions from using such language feature to using templates. Second, since most C++ open source projects are not as well logged as Java projects, it is difficult to obtain enough information about the behaviors of committers.

VI. Conclusion and Future Work

In this paper, we explore how C++ templates are adopted in open source projects by comparing library templates to user defined templates. The whole study is undertaken by investigating the uses of both function templates and class templates in open source systems. As a result, we find that an increasing use of library templates does not reduce the use of user defined templates, both library and user defined templates are biasedly adopted, and user defined templates are more likely to be derived compared to library templates. Based on these results, we give practical suggestions to both library developers and common programmers. For example, programmers are advised to pay attention on learning those commonly-used libraries which we pick out through empirical study, and we recommend library developers to develop more extensible library templates which may increase libraries' ability to support object oriented programming.

In the future work, we will replicate the study to validate the above-mentioned findings on more systems. Further, an investigation is being done by us to uncover factors that influence the adoption of templates and practical benefits of applying templates. In addition, new features introduced into C++ 11 are also our concerns.

Acknowledgment

This work is supported by the National Key Basic Research and Development Program of China (2014CB340702), the National Natural Science Foundation of China (91318301, 61321491, 61170071), and the Natural Science Foundation of Jiangsu Province (BK2011190). We especially thank Yibiao Yang from ISE group at Nanjing University for his valuable suggestions on our experiments.

References

[1] ISO. Information Technology—Programming Languages—C++. ISO/IEC 14882-1998. *ISO/IEC*. 1998.

[2] ISO. Information Technology—Programming Languages—C++, Second Edition. ISO/IEC 14882-2003. *ISO/IEC*. 2003.

[3] ISO. Information Technology—Programming Languages—C++, Third Edition. ISO/IEC 14882-2011. *ISO/IEC*. 2011.

[4] D. Vandevoorde, N. Josuttis. C++ Templates: The Complete Guide. *Addison-Wesley*. 2002.

[5] N. Josuttis. The C++ Standard Library: A Tutorial and Reference - Second Edition. *Addison-Wesley*. 2012.

[6] M. Austern. Generic Programming and the STL. *Addison-Wesley*. 1999.

[7] A. Stepanov, M. Lee. The Standard Template Library. Presentation to the C++ standards committee, March 7, 1994.

[8] H. Basit, D. Rajapakse. and Jarzabek, S. An Empirical Study on Limits of Clone Unification Using Generics. *SEKE'05*, 109-114, 2005.

[9] C. Parnin, C. Bird, and E. Murphy-Hill. Java Generics Adoption: How New Features are Introduced, Championed, or Ignored. *MSR'11*, 3-12, 2011.

[10] C. Parnin, C. Bird, and E. Murphy-Hill. Adoption and use of Java generics. *Empir Software Eng*, 18(6):1047–1089. 2013.

[11] M. Hoppe, S. Hanenberg. Do Developers Benefit from Generic Types? An Empirical Comparison of Generic and Raw Types in Java. *OOPSLA'13*, 457-474, 2013.

[12] J. G. Siek, A. Lumsdaine. A language for generic program-ming in the large. *Science of Computer Programming*. 76 (5): 423–465, 2007.

[13] D. Groger, J. Järvi, J. Siek, B. Stroustrup. G. Dos Reis, and A. Lumsdaine. Concepts: Linguistic Support for Generic Programming in C++. *OOPSLA'06*, 291-310, 2006.

[14] A. Sutton, R. Holeman, and J. I. Maletic. Identification of Idiom Usage in C++ Generic Libraries. *ICPC'10*, 160 – 169, 2010.

[15] M. J. Cole, S. G. Parker. Dynamic compilation of C++ template code. *Journal Scientific Programming*, 11(4): 321 - 327. 2003.

[16] R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. An Extended Comparative Study of Language Support for Generic Programming. *Journal of Functional Programming*, 17(2): 145 - 205. 2007.

[17] Y. Chen, R. Dios, A. Mili, and L. Wu. An Empirical Study of Programming Language Trends. *Software, IEEE*. 22(3): 72-79, 2005.

[18] R. Dyer, H. Rajan, H. N. Nguyen, and T. N. Nguyen. Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features, *ICSE'14*. 2014.

[19] S. Karus, H. Gall. A Study of Language Usage Evolution in Open Source Software. *MSR'11*, 13-22, 2011.

[20] L. A. Meyerovich, A. Rabkin. Empirical Analysis of Programming Language Adoption. *OOPSLA'13*, 1-18, 2013.

[21] O. Callaú, R. Robbes, and D. Röthlisberger. How Developers Use the Dynamic Features of Programming Languages: The Case of Smalltalk. *MSR'11*, 23-32, 2011.

[22] S. Kleinschmager, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik. Do Static Type Systems Improve the Maintainability of Software Systems? An Empirical Study. *ICPC'12*. 153 – 162, 2012.

[23] Boost C++ Library. http://www.boost.org.

[24] Loki Library. http://loki-lib.sourceforge.net.

[25] Introduction to ATL. http://msdn.microsoft.com/en-us/library/hdf7fy18.aspx.