What Makes APIs Difficult to Use?

Minhaz Fahim Zibran

University of Calgary, 2500 University Drive NW, Calgary, Canada

Summary
Use of APIs is an inseparable part of software development today. But programmers often find difficulties in using those APIs in client code [27]. This reduces programmers' productivity as well as quality of the client code [30]. Therefore, APIs should be implemented to have high usability, and to this extent a good understanding of what makes APIs difficult to use demands the foremost importance. I reviewed existing literature in this area and identified significant factors that put barriers on usability of the APIs.

In presenting the findings this paper makes two contributions: (1) The results would be useful for the API designers and developers to produce more usable APIs for the community, (2) The findings indicate further research scope in this area, which would be interesting to the researchers.

Keywords:
Application programming interface, API design, API usability

1. Introduction

In the '70s and '80s if a programmer wanted to create software, he or she had to write pretty much everything from the scratch. But with the progress of technology and software engineering reusable components, frameworks, and API libraries have been developed. As a result, the process of creating software has changed considerably: instead of creating functionality, much of today's software engineering is about integrating existing functionality or repacking it using the APIs [14].

Programmers often find difficulties in using APIs, which reduce their productivity [30] and deteriorate quality of the client code. So, usability of APIs demands increasing interest these days than in the past. A major purpose of APIs is to increase reusability of codes. But, if an API is not usable, it cannot be reusable. There are thousands of times more people using an API than designing it [9]. Once an API is deployed, it is hard to change, because any change may break the client code necessitating corresponding changes in all applications that calls that API. Practically the number of such applications is not small. Hence, the design phase of an API is the most appropriate time to take into account the usability issues. Joshua Bloch [4] expresses more conservative argument in this regard, "Public APIs are forever – one chance [before release] to get it right". Therefore, API designers and developers need a good understanding on what makes the APIs difficult to be used by programmers (users).

Not much work has been done in the area of API usability [5]. Most of the works done till date are focused on design of more sophisticated APIs aiming to increase their power [28], modular implementation [10], tailorability [8] and so on. Some design guidelines are also proposed [4, 14]. However, a few researchers focused to address the API usability issues [1, 18]. Studying the existing literature, I identified a set of important factors that affects the usability of APIs. This paper presents all these findings. A complete review of existing literature is very hard, if not impossible. Therefore, the set of factors presented in this paper may not be complete. Further research in the area should be useful to add more. Identification of potential factors affecting usability of APIs would be useful for API designers and developers to implement more usable APIs.

The remaining of the paper is organized as follows. In section 2, I discuss previous works relevant to API usability. Findings of my study are presented in section 3. Finally, I conclude the paper in section 4 with some remarks and future research direction.

2. Related Work

Works previously done in the area of API usability can roughly be classified into three categories: some proposed design recommendations, some proposed API usability measurement techniques, and some designed or implemented third party tools to help using APIs.

2.1 Design Guidelines

Michi Henning [14] using relevant examples showed that API design matters in producing useful and usable APIs. Ken Arnold [1] also addressed the importance of usability of APIs reminding that programmers who use the APIs to write client code are also human beings, and so human factors also apply to APIs. McLellan and et al. [18] conducted a case study to observe how usability of APIs affects programmers' job. Using the lessons learned from the literature, field observations, and the case study they proposed some guidelines for building more usable APIs. Joshua Bloch [4] presented the importance of designing good APIs and proposed a number of guidelines to attain

Manuscript received April 5, 2008
Manuscript revised April 20, 2008

this goal. Jeffrey Stylos and Steven Clarke [27] conducted a comparative study to asses the usability of parameterized constructors as opposed to parameterless default constructors. Brian Ellis and et al. [9] performed a quantitative user study comparing the usability of factory pattern and constructors in instantiating an object. Architectural design guidelines [3, 8, 22, 23, 25, 29] for frameworks and applications proposed by researchers also indicate some API usability issues.

2.2 Measuring API Usability

Steven Clarke and Curtis Becker [6] proposed 12 cognitive dimensions for describing and measuring API usability. Chris Bore and Sarah Bore [5] proposed 7 measures for profiling usability of APIs. Jeffrey Stylos and Brad Myers [28] map the space of API design decisions and API quality attributes.

2.3 Tool Support

In order to help developers understand API usages and write client code more effectively Tao Xie and Jian Pei developed an API usage mining framework and its supporting tool called MAPO [30]. Reid Holmes and et al. presents Strathcona [16], a tool, which provides useful examples matching heuristically the structure of the source code (that the developer is writing) to a repository of source code that uses the API. David Mandelin and et al. presented a tool, PROSPECTOR [17], which automatically synthesizes API client code to help programmers use APIs more easily. CatchUp! [13] and Diff-CatchUp [31] are tools developed to minimize the difficulties of changing client code due to change or evolution of the underlying APIs. Another such tool is presented by Ittai Balaban and et al. [2].

3. Findings

A good API makes doing simple things easy and difficult things possible. A usable API should be easy to use and hard to misuse [4]. Functional correctness, functional coverage and performance (in terms of resource utilization, speed, etc.) are the prerequisite for any API to be useful. A useful API is usable if it has 5 characteristics: (1) easy to learn, (2) easy to remember, (3) easy write client code, (4) easy to interpret client code, and (5) difficult to misuse. Factors affecting these five characteristics and consequently usability of APIs are discussed below.

3.1 Complexity

Complex APIs are difficult to learn, remember, and use correctly. If the API provides a huge set of functions, programmers find difficulty in choosing the appropriate one to use. For instance, Unix kernel provides wait(), waitpid(), wait3(), wait4(). But wait4() function is sufficient as it can be used to implement functionality of other three. Another good example may be the ‘string’ and ‘cstring’ classes of C++ [14].

However, too less functionality also limits the usability of the APIs. Use of abstraction reduces complexity and transparency, while decreases flexibility [23]. So, insensible balance between complexity and flexibility reduces usability. In this respect the recent idea of progressive disclosure [1, 19] may be useful. Conceptual complexity and trickiness in the architecture of the APIs also appear difficult for developers to learn and use them. For example, the concerns about multiple inheritance, copy constructors, destructors, friend functions, virtual functions, virtual classes, and such so many tricky concepts make C++ harder to learn and use compared to simple C or Java.

3.2 Naming

The naming scheme used in APIs much affect its usability. If appropriate names of methods, classes or variables are not consistently used throughout the API, or if the names are not self- documenting, then the API becomes difficult to use [4, 14]. Abbreviated names (Hungarian notations) are less comprehensive than camel case names. Abbreviated names don't make much sense to a programmer new to the domain or code-base. For this reason Microsoft's MFC library is difficult to use, as opposed to Java APIs. Moreover, the naming conventions recommended for Java programmers, such as, a class name should start with a capital letter, if consistently used, increases readability of the code. Because seeing a name beginning with a capital letter, a programmer new to the code-base can easily interpret that as the name of a class. Further, a Java source file name must be same as the public class defined in it. This also prevents programmers from mistakes by forcing them to use consistent names.

APIs should use the same name to mean the same things and different things should look different [4]. Otherwise programmers face difficulty in using them correctly. For instance, Perl 5's “string eval” and “block eval” may be an example of using inappropriate names, which has been fixed in Perl 6, where “block eval” is now spelled “try”.

3.3 Ignorance of Caller’s Perspective

APIs should be designed from the perspective of the caller [14]; otherwise it may loose usability. An example [14] should be explanatory here. Suppose, an API designer is implementing a function to create a TV: black&white or color, and CRT or flat-screen. The method signature he designs is as follows.

```
 void makeTV (bool isBlackAndWhite, bool isFlatScreen);
```

Such a design apparently looks fine as it uses explanatory names. But here the caller's perspective is ignored. To create a color flat-screen TV the client code will make a call as

```
makeTV ( false, true );
```

Such a call reduces readability of the client code. Without reading the documentation, just looking at the code it is hard to understand the meaning of the parameters. Taking caller's perspective into account may require little more work such as, adding enum definitions with a different function signature:

```
enum ColorTypef {Color, BlackAndWhite };
enum ScreenTypef {CRT, FlatScreen };
void makeTV ( ColorType col, ScreenType st );
```

Now the client code for a call will be

```
makeTV ( Color, FlatScreen );
```

Looking at such a more readable code a programmer would easily understand what the parameters mean.

3.4 Documentation and Code Examples

A usable API is expected to be called involving the least consultation with documentation. However, documentation is a part of APIs [14], and large APIs without good documentations are usually difficult to use [4]. Incomplete or unclear documentation reduces the usability of APIs. The notion of progressive disclosure [1, 19] may also be useful in this regard. Often open source APIs are more usable since the developers can investigate the source codes, and modify if needed. But for such APIs, absence of inline documentation in the source codes makes it hard to investigate them.

Programmers often look for example codes demonstrating common use the APIs. Absence of useful examples in the documentation often makes the APIs difficult to use [4]. Scott Henninger [15] and Lisa Rubin Neal [20] emphasized more on examples and proposed example based programming environments.

3.5 Consistency and Conventions

Inconsistencies in the design of APIs makes them difficult to learn and use correctly [4, 14]. For example, the read() and write() Unix system calls place the file descriptor first, but the Unix standard library I/O calls such as fgets() and fputs(), place the stream pointer last, except for fscanf() and fprintf(), which place it first [14]. This inconsistent order of method parameters makes them difficult to remember. Another type of inconsistency found in the strtok() function of ANSI C, where the first call has different semantics to subsequent calls.

If APIs don't respect programmers' common practices and conventions, they become difficult to use. While conducting a user study McLellan and et al. [18] found that programmers were looking for a counterpart of the function GetCommonType(), but the API did not have any such SetCommonType() or PutCommonType() functions. During the same study they found that the programmers needed clarification about the use of the term \template" in RODE library, which meant something very different for the C++ programmers.

3.6 Conceptual Correctness

Conceptual incorrectness in the design of APIs makes them difficult for the developers to learn and use correctly. For example, the .NET socket Select() function in C# takes as arguments three lists (IList) of sockets that are to be monitored. But, conceptually the function is expected to receive three sets of sockets. Using lists to model sets is incorrect: it created semantic problem because lists allow duplicates while sets don't. Unfortunately .NET collection classes do not include set abstraction [14]. Use of lists as parameters allows developers pass duplicate sockets, which is incorrect use of the function. Another design flaw in the C# v1.0 API was that Environment.HasShutDownStarted was an instance property of the static class Environment, but its constructor was private. So, developers had no way of instantiating the class to access the HasShutDownStarted property. However, this design mistake is corrected in v1.1 [19].

Another common design aw is implementation of inheritance where there is no subset (is-a) relationship [4, 12]. Such insensible inheritance hierarchies also incur difficulties to learn and correctly use the APIs.

3.7 Method Parameters and Return Type

Functions taking many parameters, specially those with multiple consecutive parameters of the same type are difficult to use [4]. For instance, the CreateWindow() function of Win32 API takes eleven arguments including four consecutive integers. Another example is the (currently deprecated) constructor of Date class in Java, namely Date(int day, int month, int year). It is very easy to make mistakes in following the input order while calling such functions. Moreover, if a function requires ten parameters, five of which are irrelevant for majority of use cases, callers pay the price of supplying them every time they make a call [14]. Joshua Bloch [4] suggests that functions requiring three or fewer parameters are ideal. Inconsistent ordering of parameters across similar functions (as mentioned in section 3.5) also makes a developer's job difficult in using them [4].

Developers often face difficulties when function's return type or value does not indicate whether it successfully accomplished its job. For instance, the .NET socket Select() function in C# (as mentioned in section 3.6) returns when no socket becomes ready within the specified timeout. But the function has void return type - that is, it does not indicate on return whether any sockets are ready [14]. Moreover, functions with return values that demand exceptional processing also impose difficulties on the developers. Returning zero-length array or empty collection is better than returning null [4].

3.8 Parameterized Constructor

Parameters in the constructors are commonly used to instantiate objects and set certain attributes at the same time. But, conducting a user study Jeffrey Stylos and Steven Clarke found that programmers preferred calling parameterless constructors followed by setter methods to instantiate objects and set certain attributes, as opposed to calling parameterized constructors [27]. Their study suggests that required constructor parameters interfere with common learning strategies, causing undesirable premature commitment.

3.9 Factory Pattern

‘Factory’ design pattern is commonly used in APIs. But, Brian Ellis and et al. [9] using a user study found that getting a desired class's instance from a factory often imposes difficulties on the programmers. They found that programmers naturally expect to use constructors to instantiate objects. They suggested that the use of factories can and should be avoided in many cases, where other techniques such as constructors or class clusters may serve the purpose.

3.10 Data Types

Improper choices of data types also may cause the developers do extra work of explicit casting, which is inconvenient [9] and may cause loss of precession. Joshua Bloch [4] suggests avoiding use of strings if a better type exists, because strings are cumbersome, error-prone, slow, and often make the user convert it to numeric values. He further suggests to prefer using double (64 bits) over float (32 bits), arguing that the precision loss due to the use of float type may be significant to the users and cause difficulty in using the concerned API.

3.11 Use of Attributes

Steven Clarke[1] conducted a user study with respect to attributes and functionality. He found that developers face difficulty when the API requires them to configure multiple attributes to achieve specific functionality, for example, when a change in application behavior needs modifying more than one attributes at a time. Developers often find difficulty in understanding the relationships between attributes. The reasons are often low visibility of concerned attributes and unclear interaction among them.

3.12 Concurrency

Often elements of APIs are designed without anticipating concurrent access in mind, and consequently documentation misses necessary information about thread safety. Concurrent access to such elements often causes unexpected behavior of the client code and even catastrophe. Ben Pryor [24] proposed four guidelines as a starting point while deciding threading policy of a class or class-member. Exposure of much mutable elements to the developers also imposes on them the extra burden to take care of thread-safety in case of concurrent use [4].

3.13 Error Handling and Exceptions

Pitfalls and back doors exposed to the users allow them doing wrong things with the API and consequently face difficulties. Class members should be private unless there is good reason to expose them [4]. Use of constants, as well as final methods and classes (where appropriate) protect the users from erroneously misusing them [1, 12].

Improper error handling also reduces usability of APIs. For example, the .NET Receive() API commits the crime for non-blocking sockets: it throws an exception if the call worked but no data is ready, and it returns zero without an exception if the connection is lost, which is just the opposite of what callers need [14]. Layered APIs become difficult to use if the exception is thrown far from where it occurred. Another common design flaw - namely exceptions for expected outcomes - also imposes difficulty in use [14]. Overuse of checked exceptions causes boilerplate [4]. Moreover, error messages with insufficient information for diagnosis and repair or recovery reduce usability of the APIs [4, 14].

[1]This information is collected from Steven Clarke's WebLog at http://blogs.msdn.com/stevencl/archive/2004/05/12/130826.aspx, http://blogs.msdn.com/stevencl/archive/2004/10/08/239833.aspx

3.14 Leftovers for Client Code

An API with high usability requires the user to type as few as possible. If the API can implement a functionality that the user needs, it should not be left for the users to implement [4], because this extra work on the users' side may make it difficult for them to use the API. For instance, simple operations like opening a file, writing a string, and closing it using the raw Windows API may take a page of code, while in Visual Basic similar operations can be implemented using just three lines of code [26]. The .NET socket Select() function in C#, as mentioned in section 3.6, would be illustrative. The function overwrites its arguments, so the caller must make a copy of each list before passing it to it, which is inconvenient. But the problem is IList does not inherit from IClonable, so there is no convenient way to copy an IList, except doing the job element by element. Moreover, the function takes an integer as the time-out parameter in microseconds. So, the maximum possible wait time is Int.MaxValue (2^{31} - 1), which is little more than 35 minutes. The function does not provide any way to wait longer or indefinitely. To implement such needs, the users may write wrapper over Select(), which may cause them write 100 lines of code [14]. Consequently the API becomes less usable.

3.15 Multiple Ways to Do One

If the APIs provide the users multiple different ways to do the same thing, developers often become puzzled to choice one from the different alternatives available. For instance, a Java programmer may create a thread in two ways: writing a class extending the java.lang.Thread class, or implementing the java.lang.Runnable interface. Since, java does not support multiple inheritance, creating a thread extending from Thread is not possible when the class already inherits another class. In such situations the only possible means is implementing the Runnable interface. Bit, in other situations often developers get puzzled to decide which way to create a thread.

Another example is the .NET socket Select() function in C#. As mentioned in section 3.6, callers pass to it three lists of sockets that they want to be monitored. In this case, there are two legal parameter values for one and the same thing: both null and empty list indicate that the caller is not interested in monitoring one of the passed lists. Here, the caller may be puzzled wondering if there is any difference between the two means. Moreover, the problem becomes severe causing difficulty in reusing Select(), for instance, writing wrapper over it [14], as indicated in section 3.14.

3.16 Long Chain of Reference

If the underlying architectures of the APIs have long complex inheritance hierarchies, the users often find difficulty in understanding them. Therefore, Gurp and Bosch [29] recommend use of delegation over inheritance. They argued that the flatter structure of the inheritance hierarchy, when using delegation, is easier to understand than vertical inheritance hierarchy. However, for method invocation, they argued that long chain of method delegations is also difficult to track, and so they further suggested preferring loose coupling (in the form of event mechanism) to delegation.

However, when a single functionality scattered over a number of objects, it is naturally difficult to track [27], no matter whether loose coupling or method call chain is used.

3.17 Implementation vs. Interface Dependency

Interface dependencies between components are more flexible and should be preferred over implementation dependencies [29]. Joshua Bloch [4] also suggests preferring interface types to classes for input. When a method needs to take an object as parameter, using the class of the object as parameter type causes implementation dependency. In such implementation an object of that specific class or any of its subclasses can be passed to the method. On the contrary using an interface as the parameter type allows passing any object implementing the interface or its sub-interface. This gives more flexibility to the users of the function.

3.18 Memory Management

APIs that leave the responsibility of memory management and garbage collection on the user are less usable. For example, a C++ programmer needs to take care of possible memory misuse, when he or she uses a pointer, dynamically allocates memory invoking functions like malloc(), or creates, destroys or passes objects as parameter to functions. This makes C++ difficult to use compared to Java and .NET, which take care of memory management and garbage collection overheads. The CORBA C++ mapping requires callers to fastidiously keep track of memory allocation and deallocation responsibilities; the result is a less usable API making it easy to corrupt memory [14].

3.19 Technical Mismatch

Use of more than one programming languages or frameworks in developing a single application is a common practice today. But sometimes, APIs are designed making wrong assumptions about the control model, data model or protocols, which consequently make those APIs less inter-operable with others [11]. Such a common mistake is, when a framework is assumed to have the main control of execution, which may not be true in situations. Moreover, an API may work fine on a platform while malfunctions on another. Such incompatibilities and technical mismatches negatively affect usability of the APIs.

3.20 API Evolution or Change

Sometimes evolution or change in APIs breaks the client code. APIs without support of backward compatibility are typically difficult to use specially in maintaining legacy software, or those evolving through a number of releases over time. Since Microsoft Visual Basic .NET is not backward compatible to VB 6.0, applications developed using VB 6.0 has become difficult to migrate to VB .NET. Such difficulties increase when the APIs don't come with complete documentation about the changes made. As mentioned in section 2.3 researchers introduced third party tools to mitigate such difficulties.

Moreover, deprecation of commonly used classes and methods in the newer version of API often surprises the users. They often face difficulty in understanding the reason of deprecation and finding the proper alternatives.

3.21 API Aging

The notion of software aging [21] also applies to APIs. Many programmers are using APIs provided my Microsoft, while many others are reluctant to use those. Without keeping backward compatibility Microsoft introduces completely new APIs, and their new operating systems (OS) often don't support old APIs. Applications using Win32 APIs don't work properly on OS versions later than Windows 98. Later they introduced MFC library. Instead of adding features to Win32 APIs they introduced WinFX, Avalon, XAML. If one developed GUI applications using Microsoft's programming environment, WinForms, he had to start over again in two years to support Longhorn and Avalon [26]. Such frequent replacements of Microsoft's APIs reduce motivation of many programmers to use them.

3.22 Intelligibility of Source Code

This factor applies to open source APIs. Low readability of source code due to large source files, large methods, lack of inline documentation, improper indentation, split implementation of classes (i.e. C++ classes implemented using header and implementation files), etc. imposes difficulties on the users when they need to investigate the source code of the concerned APIs.

4. Conclusion and Future Work

An API is a user interface to the underlying programming model (i.e., frameworks, components, etc.) that is presented to the user (client code developer) [1]. Essentially there are two aspects of a good API design. From the perspective of the API designer architectural elegance is important to ensure efficiency, usefulness and changeability of the API. From the perspective of client code developer usability of a useful API is significant. Usability problems in APIs reduce programmers' productivity [30]. Damian Conway states, “The most important aspect of any module is not how it implements the facilities it provides, but the way in which it provides those facilities in the first place” [7]. Many software engineering techniques have been introduced to guide elegant architectural design. But comparatively few works have been done addressing the usability issues.

From the study of existing literature, I pointed out significant factors that make APIs difficult to use. The findings presented in this paper would help API designers to gain better understanding on API usability and develop more usable APIs.

However, factors leveraging changeability and reusability of components may conflict with usability of the APIs. For example, large components composed of smaller components increase reusability of those smaller components. As small components are easier to comprehend [29], this would help API developers ease in maintaining and changing the implementation. But on the other hand, such implementation may require the client code developers track long chain of method delegation, which is inconvenient.

Hence, future work in this regard may include study of factors that enhance architectural elegance to leverage reusability and evolution, as well as identifying those factors, which conflict with usability issues. From such study valuable guidelines may be introduced for developing evolvable, useful, and usable APIs making sensible balance among such conflicting factors.

References

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

[2] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. SIGPLAN Not., 40 (10): 265-279, 2005.

[3] T. Biggerstaff. The library scaling problem and the limits of concrete component reuse. Software Reuse: Advances in Software Reusability, 1994. Proceedings., Third International Conference on, pages 102-109, 1-4 Nov 1994.

[4] 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.

[5] 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.

[6] 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.

[7] D. Conway. Ten essential development practices. 14 July 2004.

[8] S. Demeyer, T. D. Meijler, O. Nierstrasz, and P. Steyaert. Design guidelines for tailorable frameworks. Commun. ACM, 40 (10): 60-64, 1997.

[9] 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.

[10] M. Fayad and D. C. Schmidt. Object-oriented application frameworks. Commun. ACM, 40 (10): 32-38, 1997.

[11] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In ICSE '95: Proceedings of the 17th international conference on Software engineering, pages 179-185, New York, NY, USA, 1995. ACM.

[12] E. R. Harold. XOM design principles. Extreme Markup Languages, 2-6 August 2004.

[13] 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.

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

[15] S. Henninger. Retrieving software objects in an example-based programming environment. In SIGIR'91: Proceedings of the 14th annual international ACM SIGIR conference on Research and development in information retrieval, pages 251-260, New York, NY, USA, 1991. ACM.

[16] 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.

[17] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pages 48{61, New York, NY, USA, 2005. ACM.

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

[19] Microsoft Developer Network. Designing .NET class libraries: Designing progressive APIs. MSDN Online Chat, 02 March 2005.

[20] L. R. Neal. A system for example-based programming. In CHI '89: Proceedings of the SIGCHI conference on Human factors in computing systems, pages 63-68, New York, NY, USA, 1989. ACM.

[21] L. R. Neal. A system for example-based programming. In CHI '89: Proceedings of the SIGCHI conference on Human factors in computing systems, pages 63-68, New York, NY, USA, 1989. ACM.

[22] D. L. Parnas. On the criteria to be used in decomposing systems into modules. Commun. ACM, 15 (12): 1053-1058, 1972.

[23] D. L. Parnas and D. P. Siewiorek. Use of the concept of transparency in the design of hierarchically structured systems. Commun. ACM, 18 (7): 401-408, 1975.

[24] B. Pryor. Simple concurrency guidelines. Ben Pryor's Blog, March 2008.

[25] Y. Smaragdakis and D. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. ACM Trans. Softw. Eng. Methodol., 11 (2): 215-255, 2002.

[26] J. Spolsky. How Microsoft lost the API war. In Business of Software 2008, a JOEL ON SOFTWARE Conference, Boston, MA, United States, 13 June 2004.

[27] 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.

[28] J. Stylos and B. Myers. Mapping the space of API design decisions. Visual Languages and Human-Centric Computing, 2007. VL/HCC 2007. IEEE Symposium on, pages 50-60, 23-27 Sept. 2007.

[29] J. van Gurp and J. Bosch. Design, implementation and evolution of object oriented frameworks: concepts and guidelines. Softw. Pract. Exper., 31 (3): 277-300, 2001.

[30] 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.

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

Minhaz Fahim Zibran received the B.Sc. in Computer Science and Information Technology in the year 2002 from the Islamic University of Technology, Bangladesh. He earned M.Sc. in Computer Science in 2005 from the University of Lethbridge in Alberta, Canada. Currently he is a Ph.D. student in the Laboratory for Software Modification Research at the University of Calgary in Alberta, Canada. He has research experience in the area of combinatorial optimization. His current research is focused on evolutionary software engineering.