CHAPTER TWENTY-NINE

# How Usable Are Your APIs?

*Steven Clarke*

In November 1999, I started working as a user experience researcher on the user experience team for Visual Studio. I was excited to be part of a team that designed the product I had used in my previous job. While working as a software developer at Motorola, I had always wanted to be able to make big improvements to the user experience of Visual Studio, and now I had my chance. But, although it seems obvious now, it took me almost a year to figure out that the way to improve the developer's user experience wasn't necessarily to focus exclusively on the development tools.

When I joined the team, I focused exclusively on achieving a deep understanding of the way that developers used Visual Studio. Together with my colleagues, we spent countless hours observing the experiences developers had creating new projects in Visual Studio: adding new classes to existing C++ projects, debugging web services, and many other experiences. We learned a great deal of useful information that enabled us to better understand our customers and how to apply that understanding to designing and building a great development tool.

But there was something missing. Although the developers' experience with the development tool was getting better all the time, the overall developer experience wasn't improving to the same extent. We needed to change focus. Instead of focusing on things that developers rarely do (how often do you create a new project compared to everything else that you do?), we needed to focus on the things that developers often do. And that is reading, writing, and debugging code that uses many different APIs. So we started the long journey of measuring and improving the user experience of the class libraries and APIs that developers use.

We didn't get it right on the first try. But through perseverance and a strong desire to improve, we managed to achieve a good understanding of what it means for an API to be considered usable. We learned many lessons along the way, some of which fundamentally questioned our assumptions about how developers write code. In this chapter I share those stories and the journey we took.

## Why Is It Important to Study API Usability?

Unlike graphical user interfaces, developers don't interact with APIs in a direct manipulation style through widgets or UI controls. Instead, the developer types some text into a text editor and then, some time later, compiles and runs the program. The developers don't interact directly with the API, but rather interact directly with the tools they use to write and run the code. Obviously the usability of these tools will have a great impact on the developer's ability to write and run code that uses the API.

But is it OK to stop there? What if, in an ideal world, the tools were the perfect tools for a developer? Would the experience that the developer has writing and running code that uses an API also be perfect? Let's consider the impact a badly designed API has on a developer trying to use it, even with the ideal set of tools.

APIs are supposed to increase a developer's productivity by providing components that perform common tasks. Indeed, many times a developer must use an API to accomplish some task because doing so is the only way to access encapsulated system functionality that can't be implemented from scratch. Even when the operations can be executed without the special privileges to reach into hidden parts of the system, using the API saves the developers the time and effort required to create their own implementations of these components. Unfortunately, when an API is badly designed, it can be difficult to determine whether the time saved in using the API is worth the investment of effort in using it.

Alan Blackwell presented a model for describing how to determine whether the investment in programming a custom solution instead of using a predefined API is worth the effort [Blackwell 2002]. In the model, the developer estimates the cognitive cost of using the API as well as the cost of figuring out how to use the API. He then weighs these costs against the payoff, the potential savings in time and effort that he will accrue by using the API. The model takes into account the possibility that no such payoff will result.

To perform such an analysis, the developer uses cues in the API design such as class names, method names, and documentation to estimate the cost involved. Particular design patterns employed in the API might be recognized by the developer, and these might help him better understand how to use the API. The developer may have had experience with other parts of the API, which might suggest a particular approach to take. Additionally, the developer's own understanding of the domain in which he is working provides another perspective on the API. The combination of API cues, design patterns, prior experience with the API, and domain

understanding helps the developer estimate the effort involved to implement a solution. From this, he can decide whether or not to use the API.

However, many things can prevent the developer from making an accurate estimate of the effort required:

- Class names and documentation obviously can be misleading and inaccurate. They can suggest that the API behaves in one way, even if it actually behaves in another. Martin Robillard indicated that one of the biggest obstacles to learning an API is the lack of good resources that describe the API and how it works [Robillard 2009].
- Another common problem with names is that they can have multiple meanings. Determining the specific meaning often requires an understanding first of the conceptual model underlying the design of the API.
- Prior experience with other parts of the API might not indicate how other parts of the API will work, particularly if design guidelines have not been followed consistently throughout the design.
- The way that the developer thinks about the domain may well be very different from the way that the API represents the domain.

Unfortunately, the developer can rarely tell right away that an API is inconsistent, has poorly named classes and methods, or represents the domain differently from the way he expects. A big problem in estimating the costs involved in using an API is that the developer often has to go through the effort of creating and running a small program that uses the API before he can estimate the costs involved in creating and running a larger program that uses the API. Only once he has started writing some code that uses the API do all the idiosyncrasies and quirks in the API's design become apparent.

Crucially, this effort is an added burden to the effort the developer had already estimated he would spend in using the API, and can quickly exceed the estimated costs of implementing a custom solution. Thus, developers may feel that instead of increasing their productivity, the API has actually made them less productive.

Hence the explosion of online forums, websites, and blogs devoted to helping developers understand the costs involved in using an API. Sites such as StackOverflow.com allow the developer to post questions and read answers about performing specific tasks with an API.

The end result is that developers often can't figure out how to use the API, and instead of writing code and being productive, they end up spending their time browsing online forums, posting questions and reading answers. Their self-esteem takes a hit. The developer feels that all she is doing is repeatedly crafting search queries for online forums and copying and pasting code she finds online. It doesn't really feel like she is "programming" anymore.

Even perfect tools can't compensate for a badly designed API. Tools can help developers write and run code, but a lot of the effort in using a badly designed API is cognitive effort. The developers have to think about the best way to translate their intentions into a particular

incantation of different classes and methods. Debuggers and other such analysis tools can help the developers recognize more quickly that there is a mismatch between the way that they understood or expected the API to work and the way that it actually works. But the developers still need to process the information from the tools. And the more mismatches that occur between the API and the developers' expectations, the more mental processing the developers need to do to overcome them.

In contrast, a well-designed API allows developers to make accurate estimates of the investment required to use the API. The names used in the API effectively communicate the purpose of the classes and methods to the developers. The classes and methods in the API map well to the classes and methods that the developers expect to find in the API. The way that these classes and methods work is consistent with the way that other parts of the API work. The developers are able to get into "the zone," and the code that they write flows naturally from their thoughts to the screen. Even if the tools that the developers use aren't optimal, they can still have a great experience with the API because the developers don't depend as much on the tools as they do when working with a badly designed API. In an ideal world, the tools really just let the developer type in and run their programs.

## First Attempts at Studying API Usability

It seems obvious now that the usability of an API plays a critical part in the overall developer experience. But at the time that I joined the Visual Studio user experience team, it simply didn't occur to me.

As a developer who had used Visual Studio before joining the Visual Studio team, I was often frustrated by the tools I used. Compiler settings were difficult to find and set. Configuring library paths seemed cumbersome. These were the types of usability issues that I wanted to fix when I joined the Visual Studio user experience team.

It's not that I hadn't experienced any difficulties with the Microsoft APIs. On the contrary, some of the biggest difficulties I had faced were related to understanding how the Microsoft Foundation Classes (MFC) APIs worked. But in stark contrast to the way I responded to the tools when they didn't work as I expected them to, I blamed myself for not being able to figure out how to use MFC. Not once did I think that the problem could have been with the design of the API.

This reaction to a badly designed API can be quite common. Many developers explain away the difficulties they are having with an API by putting it down to their lack of experience with the API. If they only had enough time, they could work it out. I have heard many developers say that the problem is with them, that they aren't smart enough to figure out how to use an API.

So, even though I was part of the Visual Studio user experience team, the insight that we can study the usability of an API didn't come from me. Instead, it came from a team I was working with that was developing a brand-new API called ATL Server.

ATL Server was a C++ API for building web applications. Many of the developers, testers, and program managers working on the ATL Server API team had also worked on creating the MFC API. They had learned from their experience working on MFC that they didn't always know the best way to design the API. They knew there were some usability issues in the API that were related to design decisions they had made during the development of the API. But now that the MFC API had shipped and was being used by customers, it was too late to make significant changes, since these would run the risk of breaking customers' code. Now that the team was working on a brand-new API, the team had the opportunity to find out whether they were making the right design decisions before shipping it.

Since I was the usability engineer assigned to their team, they asked me for help. I set about trying to figure out how to study the usability of an API.

My first thought was: how could I possibly study the many different ways that the API can be used? There are many tasks that a developer could perform with this API. How could I possibly hope to study each of these? Instead, the approach I took in designing the study was to focus on the architecture of the whole API. I thought it would be important to learn whether the classes' organization made sense to developers. This would help me address the issue of scale and would allow me to measure how closely the developers' domain model for ATL Server mapped to the API model. My hope was that I would be able to identify the areas where mismatches occured so that the ATL Server team would be able to correct the design of the API.

To that end, I set about running a card sort study on the API. In a card sort study, participants are given a stack of physical cards. A description of some concept or the name of some object (e.g., a web page, product, method) is written on each card, one name per card. Participants read each card one by one and group the cards into piles that they think are most similar. Participants can group cards according to any criteria they choose. Once all participants have created their groups, a cluster analysis is performed over the collection of groupings to measure similarities between pairs of cards. The end result of the analysis is a graph known as a *dendogram* that shows the most common groupings for each card. A card sort study seemed the most appropriate way to gather data about developers' conceptual model.

## Study Design

Fourteen experienced C++ developers participated in this study. All participants said that they developed C++ code for more than 20 hours per week at work, using Visual C++ v6.0 (the study was performed in March 2000). None of the participants had extensive experience developing for the Web. One of the objectives for ATL Server was to make it appropriate for developers who did not have significant prior experience building web applications. So the participants' lack of web development experience was fine for the purposes of this study.

Twenty-four classes were identified as being core to ATL Server and were the focus of this study. These classes are listed in Table 29-1.

*TABLE 29-1. Core ATL Server classes*

| | | | |
|---|---|---|---|
| CStencil | CMultiPartFormParser | CWriteStreamHelper | CHtmlStencil |
| CISAPIExtension | CHttpRequestParams | CValidateObject | CStencilCache |
| CValidateContext | CHttpRequestFile | CDBSessionDataSource | CHttpResponse |
| TServerContext | CPersistSessionDB | CHttpRequest | CCookie |
| CSessionNameGenerator | CDefaultErrorProvider | CHtmlTagreplacer | CSessionStateService |
| CSessionState | CRequestHandlerT | CPersistSessionMem | CDllCache |

Short, succinct descriptions of the functionality exposed by these classes were created by the technical writers on the team. These descriptions were typically one or two sentences long. Each description was printed onto an index card, one description per card. For example, put and get methods for setting and retrieving class member variables would map to one card containing the description "Manage variable." Common methods shared by all classes, such as constructors and destructors, were not listed.

Fifty-nine cards were prepared describing the functionality implemented by these classes. None of the cards contained the name of the class that exposed the functionality described on the card.

As an example, the `CHttpRequest` class exposes four main pieces of functionality. Descriptions of each of these were written onto four cards:

- Get the request data (name-value pairs, form fields) from the HTTP query.
- Initialize object with information about the current request.
- Maintain the request buffer.
- Retrieve request and server variables.

Finally, a short document presenting an overview of the ATL Server framework was provided to participants as reference material throughout the study.

Participants were given 90 minutes to use their own judgment and understanding of the framework to sort the 59 description cards into what they felt would be appropriate classes, or groups. The minimum number of groups created by any participant was eight, and the maximum was fourteen. The participant's groupings were recorded in a spreadsheet, and we performed a cluster analysis of the data.

## Summary of Findings from the First Study

The cluster analysis showed that responses fell into three common groups. One group contained two descriptions, the second group contained eight descriptions, and the third contained the rest (minus two that were ungrouped).

The descriptions contained in each group had very little in common. The research team expended a good deal of time trying to determine what, if anything, could be learned from the groupings, but without success. Because the cluster analysis showed three distinct groups among the participants, we concluded that they had no conceptual model in common.

At first, the results were very disappointing. The three groups that the analysis had created weren't at all meaningful and could not be used by the team to determine any specific corrective action. It seemed like the effort involved in designing, running, and analyzing the study had been for nothing.

However, in the long run, the study taught us a very valuable lesson, totally apart from its formal outcome. The experience that we gained designing, running, and analyzing this study taught us that API usability isn't a function of the whole API. Studying the architectural design of an API doesn't tell you everything you need to know about its usability.

I had been attracted to a card sort study because I had been concerned with the issue of scale: how could I account for the many different ways that the same set of classes and methods in an API can be used together to accomplish different tasks? But after running the study, I realized that was the wrong thing to focus on. In real usage, very few developers ever concern themselves with the whole API. They concern themselves with the task they are trying to perform.

Many people can have different experiences with an API due to the different tasks that they are performing, the different backgrounds that they have, and the different work styles that they follow. Attempting to determine the usability of an API without taking all of that into account will likely end up with the inconclusive result that we reached in the card sort study.

Even if the card sort study had been able to identify a common grouping, it would not be clear what to do with such a result. If the API design had been altered so that the classes more closely mapped onto such a fictional grouping, how would we measure the impact of changing the design on the ability of developers to use the API? We would know that the API now mapped to the conceptual model that participants had, but would that be a good thing? Is it really the case that we should map the API design to the conceptual model of developers who have not used the API?

Effectively, that is the argument I had been making in designing and running the card sort study, but in retrospect the argument lacks credibility. The card sort study convinced me that if we really want to understand API usability, we need to try to study the API when it is being used by real developers solving real tasks.

Thankfully, the culture at Microsoft is such that, even having produced no useful result from my first API study, I was encouraged to continue exploring ways to measure and describe API usability.

## If At First You Don't Succeed...

The next opportunity to study API usability came 18 months later. With my colleagues in the Visual Studio user experience team, I worked on a study that investigated whether experienced Visual Basic 6 developers could write .NET applications using the Visual Basic .NET language.

Visual Basic .NET is the latest version of the Visual Basic programming language, and it departs radically from earlier versions of Visual Basic, such as Visual Basic 6. Most notably, the .NET framework used to write programs in Visual Basic .NET provides similar functionality to Visual Basic 6 frameworks, but is composed of a completely different set of classes and namespaces. There are other differences as well. For example, in Visual Basic .NET, the default lower bound of every dimension of an array cannot be changed from 0 to 1 as it can be in Visual Basic 6.

Our team wanted to evaluate the experience of developers using Visual Basic .NET for the first time to perform common tasks such as writing to and reading from text files on disk. Having learned our lesson from the card sort study on the ATL Server API, we decided that in this study, we would provide participants with a specific task and observe them in our usability labs performing those tasks. We would videotape each participant and ask them to talk out loud as they were performing the task. We would then analyze each video to look for common patterns of behavior, interesting highlights, and any evidence that could help us better understand the experience that participants had with both Visual Basic .NET and the .NET framework.

### Design of the Second Study

We recruited eight developers, all of whom said that they spent at least 20 hours per week writing Visual Basic 6 code at work. We asked them to use Visual Basic .NET and the .NET framework to write a program that would write to and read from text files on disk. Each participant was given two hours to complete the task. While working on the task, they were free to browse the documentation available for any of the classes in the .NET framework. Each participant tackled the task individually in the usability labs on our campus in Redmond, WA. A usability engineer was behind the one-way mirror in the usability lab to operate the video cameras and to take notes. Other than helping the participant recover from any bugs or crashes that came up with the prerelease version of Visual Studio, the usability engineer did not help the participant with any of the tasks.

## Summary of Findings from the Second Study

In contrast to the results from the ATL Server card sort study, the results from this study were very enlightening. The headline result was that not one participant could complete any of the tasks successfully in the two hours given.

This was quite unexpected. We didn't think every participant would be completely successful, but we also hadn't expected such a complete bust. Our first reaction was that the participants simply had not satisfied the study recruiting profile and were in fact not experienced programmers at all. But it was clear from the video recording of each session that this was not the case. Each participant was clearly an experienced programmer who was not daunted at the prospect of picking up and using a new programming language and environment.

So what was the reason for the poor success rate? We reviewed the videos from each session in detail soon after the study. We had already reviewed each session to create highlight video clips that we distributed internally around the Visual Studio product team almost immediately after the last session had ended. These highlights showed the difficulties that participants had performing each of these tasks, so we had a few hypotheses about the reason for the difficulties.

The first hypothesis we tested was whether the documentation was to blame. This hypothesis was inspired by watching all of the participants spend most of their time in the usability lab searching and hunting through the documentation, looking for some class or method that would help them write to or read from a file on disk. At the time we ran the study (summer 2002), the documentation did not contain any code snippets for common tasks such as writing to or reading from a file on disk.

For example, the transcript of one participant browsing the documentation for the System.IO namespace is typical:

> PARTICIPANT:
>
> *[Reading names of classes]*
>
> Binary reader. Buffered stream. Reads and writes....
>
> *[Pauses. Scrolls through list of classes].*
>
> So let me just...stream writer. Implements a text writer for writing characters to a stream in a particular encoding. Umm...stores an underlying string. String builder. Text...hmmm. Text reader, text writer. Represents a writer that can write a sequential series of characters. This class is abstract.
>
> *[Pause]*
>
> Ummm....
>
> *[scrolls up and down through list of classes]*
>
> So it, you know, it sounds to me like it's, you know, it's more low-level kind of stuff. Whereas I just want to create a text file. Umm.

*[Points at the description of one of the classes]*

Characters to a stream in a particular encoding. I'm not sure what...obviously a text writer for writing characters to a stream.

*[Clicks on the link to view more details about the TextWriter class. Then looks at list of classes that derive from TextWriter]*

System dot IO dot StringWriter. This seems too low-level to me to be a text writer thing but maybe I am wrong. Umm....

*[scrolls through description of the TextWriter class]*

Text writer is designed for character output, whereas the stream class is designed for byte input and output.

*[Sigh. Clicks on link to view TextWriter members]*

Probably going where no man should have gone before here.

The associated video clip is very compelling and is the one we used extensively to publicize the results of the study throughout the Visual Studio product team. Along with other critical video clips, it generated huge empathy for the customer on the part of the product team and a massive desire to figure out what was broken and how to fix it.

In another video clip, a participant is struggling with a compiler error:

PARTICIPANT:

OK. We are back to one error.

*[Reads the compiler error message]*

Value of system. Value of type system dot io dot filestream cannot be converted to system dot io dot streamwriter. No clue....

*[pause]*

Are you gonna make me suffer through this?

Since a large proportion of each participant's time was spent browsing the documentation, many people in the product team latched onto that and believed that the best way to improve the experience would be to modify the documentation. If the documentation could provide clear pointers to sample code and code snippets, developers could copy and paste those snippets into their own code.

However, we wanted to be sure that the lack of code snippets truly was the underlying cause of the problems we observed in the usability lab. Although modifying the documentation at this stage (three-quarters of the way through the project life cycle) would be a less costly fix than modifying the APIs, it would be wasted energy if it turned out to have no relationship to the underlying cause.

In order to be confident in our analysis of the problems, we needed a framework or a language upon which to base our analysis. Without such a framework, we would have difficulties explaining and justifying our analysis to a product team that was already convinced by the video clips that the problem was the documentation. We ended up using the Cognitive Dimensions framework [Green and Petre 1996].

## Cognitive Dimensions

The Cognitive Dimensions framework is an approach to analyzing the usability of programming languages and, more generally, any kind of notation, such as an API. The framework consists of 13 different dimensions, each describing and measuring a specific attribute that contributes to the usability of the notation.

Each dimension is effectively a probe into the design of the notation, measuring one attribute of its usability. Each measurement can be compared to a baseline (for example, the preferences of a set of developers), and the results of the measurements and comparisons can then be used to determine any corrective action that needs to be taken to ensure that the notation maps well to the baseline.

The measurements that result from this kind of analysis aren't numerical or even necessarily absolute measurements, but are instead more like ratings on a scale, defined by each dimension. Although this may introduce some uncertainty about the accuracy of each measurement, the beauty of the framework overall is that it identifies and provides a vocabulary with which to discuss the detailed attributes of the usability of a notation. Without such a detailed vocabulary, any analysis of the usability of a notation is fraught with difficulty, because the lack of labels and definitions for each of the different attributes of the usability of a notation makes it close to impossible to reach a shared understanding of what important attributes need to be considered.

We made some modifications to the names to make the framework more relevant to API usability [Clarke and Becker 2003]. A short description of each dimension follows:

*Abstraction level*
: The minimum and maximum levels of abstraction exposed by the API, and the minimum and maximum levels usable by a targeted developer

*Learning style*
: The learning requirements posed by the API, and the learning styles available to a targeted developer

*Working framework*
: The size of the conceptual chunk (developer working set) needed to work effectively

*Work-step unit*
: How much of a programming task must or can be completed in a single step

*Progressive evaluation*

To what extent partially completed code can be executed to obtain feedback on code behavior

*Premature commitment*

The number of decisions that developers have to make when writing code for a given scenario and the consequences of those decisions

*Penetrability*

How the API facilitates exploration, analysis, and understanding of its components, and how the targeted developers go about retrieving what is needed

*API elaboration*

The extent to which the API must be adapted to meet the needs of targeted developers

*API viscosity*

The barriers to change inherent in the API, and how much effort a targeted developer needs to expend to make a change

*Consistency*

How much of the rest of an API can be inferred once part of it is learned

*Role expressiveness*

How apparent the relationship is between each component exposed by an API and the program as a whole

*Domain correspondence*

How clearly the API components map to the domain, and any special tricks that the developer needs to be aware of to accomplish some functionality

We were attracted to the framework based on claims about its ease of use. Green and Petre claim that the Cognitive Dimensions framework can be used by people without any formal training or background in cognitive psychology [Green and Petre 1996]. We believed that it was vital for the development team reviewing the results of our study to view those results in the context of a framework or language that made sense to them.

We used the framework to help describe the experience that participants had in each task. We reviewed the video recordings of each session and made video clips for each interesting event we observed. We then tagged each event with the set of specific cognitive dimensions that we believed were relevant, and reviewed the video clips in the context of the dimensions associated with them. Thus, we reviewed all the abstraction level clips together, all of the viscosity clips together, etc. This approach gave us a different perspective on the video recordings of each session. We used that perspective to develop and articulate a clearer explanation for the difficulties observed.

Crucially, we were able to use the Cognitive Dimensions framework to convince the product team that fixing the documentation alone would not suffice. Although it was necessary, our analysis told us it would not be sufficient.

The Cognitive Dimensions analysis suggested that the critical underlying problem was related to the abstraction level of the APIs used. To explain this explanation in more detail, let's consider the code snippet that a participant had to write in order to read code from a text file:

```
Imports System
Imports System.IO

Class Test
    Public Shared Sub Main()
        Try
            ' Create an instance of StreamReader to read from a file.
            Dim sr As StreamReader = New StreamReader("TestFile.txt")
            Dim line As String
            ' Read and display the lines from the file until the end
            ' of the file is reached.
            Do
                line = sr.ReadLine()
                Console.WriteLine(Line)
            Loop Until line Is Nothing
            sr.Close()
        Catch E As FileNotFoundException
            ' Let the user know what went wrong.
            Console.WriteLine("The file could not be found:")
            Console.WriteLine(E.Message)
        End Try
    End Sub
End Class
```

Writing to a file is very similar:

```
Imports System
Imports System.IO

Class Test
    Public Shared Sub Main()
        ' Create an instance of StreamWriter to write text to a file.
        Dim sw As StreamWriter = New StreamWriter("TestFile.txt")
        ' Add some text to the file.
        sw.Write("This is the ")
        sw.WriteLine("header for the file.")
        sw.WriteLine("-------------------")
        ' Arbitrary objects can also be written to the file.
        sw.Write("The date is: ")
        sw.WriteLine(DateTime.Now)
        sw.Close()
    End Sub
End Class
```

This code doesn't appear particularly complex. The `StreamReader` or `StreamWriter` constructor takes a string representing a path to a file. Then, the reader or writer is used to read from or write to the disk accordingly.

If you already know the solution to these tasks, and you watch the videos of participants struggling to write this code, it is natural to leap to the conclusion that the problem lies with the documentation not showing a code snippet for this task. After all, how could this code be made any simpler?

However, when we analyzed the video clips and the associated talking commentary provided by each participant, we knew that the problem is not the lack of code snippets. For example, reread a portion of the transcript that I quoted previously:

> PARTICIPANT:
>
> System dot IO dot StringWriter. This seems too low-level to me to be a text writer thing but maybe I am wrong. Umm....
>
> *[scrolls through description of the TextWriter class]*
>
> Text writer is designed for character output whereas the stream class is designed for byte input and output.
>
> *[Sigh. Clicks on link to view TextWriter members]*
>
> Probably going where no man should have gone before here.

When the participant says that this seems too low-level, he is referring to the abstraction level of the classes he is browsing. Instead of a very flexible, but somewhat abstract, stream-based implementation, participants expected something less abstract and more concrete. The representation of the task in the API was far removed from the way participants expected the task to be represented. Instead, participants had to change the way they thought about text files in order to recognize that the `StreamWriter` and `StreamReader` classes were the correct classes to use.

We made this same observation among the majority of the participants. Even when they were browsing the documentation for the class that would allow them to complete the task (`StreamReader` or `StreamWriter`) they would dismiss it, because it didn't map to the way they expected reading and writing to work.

So the problem wasn't so much that the documentation lacked appropriate code snippets. It was more that participants were searching for something in the documentation that didn't exist. They expected to find a concrete representation of a text file, but found only representations of file streams.

The Cognitive Dimensions framework gave us a language with which to analyze and better understand the problems we observed in the study. With this analysis, we convinced the team that the best way to fix the problem, in addition to addressing the need for better documentation, would be to design and create a new class that concretely represented the filesystem and files on disk.

Six months later, the team had designed and implemented a prototype version of this new class. The code required to complete the same tasks using this new class (`FileObject`) follows:

```
Dim f As New FileObject
f.Open(OpenMode.Write, "testfile.txt")
f.WriteLine("A line of text")
f.Close()
```

In this case, a file is represented by an instance of the `FileObject` class. The `FileObject` class can exist in different modes (in this case, `OpenMode.Write` opens the file for writing). This contrasts starkly with the earlier code examples that use different classes to represent the different modes (`StreamWriter` and `StreamReader`).

We ran a study like our previous one on the new FileObject API, using the same tasks and the same profile of participants (although not the same participants). This time the results were markedly different. Within 20 minutes and without browsing any documentation, all participants were able to complete each task.

## Adapting to Different Work Styles

Our study not only helped us add useful new classes to a single (albeit important) API, but also led to a new respect and zeal for studies throughout Microsoft. The API review process that every API had to go through in order to ship with the .NET framework was updated to include an API usability process.

Although we found it very encouraging to see the increased level of interest in our way of generating usability information, it wasn't clear at the time if API usability meant something different to different developers or if it was a universally agreed upon quantity. The developers who had participated in our file I/O studies all shared the same profile: experienced Visual Basic developers who used Visual Basic at work for more than 20 hours per week. But would developers with a different profile have responded the same way?

We had a hunch that they wouldn't. The developers who had participated in the studies were representative of one of the three developer personas that we used at the time to design the Visual Studio IDE:*

*Opportunistic developers*
: Characterized by a habit for rapid experimentation, a task-focused approach, and extensive use of high-level, concrete components

*Pragmatic developers*
: Characterized by a code-focused approach and use of tools that help them focus on the robustness and correctness of the code that they write (refactoring tools, unit testing tools, etc.)

* For a description of how we developed these personas, see [Clarke 2007].

*Systematic developers*

: Characterized by a defensive approach to development and their need for a deep understanding of any technology before they start working with it

The developers who had participated in the file I/O studies were very much in the opportunistic camp. The way they used the `FileObject` class in the second study was almost a perfect example of opportunistic programming. The contrast with the more abstract classes they had to use in the first study was stark. Indeed, one of the main findings from the file I/O studies was that without such high-level concrete components, opportunistic developers are unlikely to be successful.

With the raised levels of interest in API usability, we now needed to determine what made an API usable for pragmatic and systematic developers as well.

Having developed the three personas through observations of developers during many studies that had taken place earlier, we had a large body of data we could return to. A colleague and I reviewed that data and used the cognitive dimensions to define API usability relative to each persona.

To begin with, we defined a scale for each dimension in the framework. For example, for the abstraction level, we defined a scale ranging from "Primitives" to "Aggregates."

APIs that have a primitive abstraction level expose a collection of low-level components. In order to complete a task with a primitive API, developers select a subset of these primitive components and use them together to achieve some effect.

For example, a primitive file I/O API might represent a file as a stream. To read data from the stream, the API might expose a `StreamReader` class. The developer needs to combine these classes together in order to read data from the file. For example:

```
File f = new File();
StreamReader sr = f.OpenFile("C:\test.txt");
Byte b = sr.Read();
string s = b.ToString();
f.CloseFile();
```

In such an API, developers need to figure out how to combine the primitives to achieve the effect they desire (in this case, reading from a text file).

In contrast, an aggregate API might contain a class that represents a text file and exposes all of the operations that can be performed on a text file. For example:

```
TextFile tf = new TextFile("C:\text.txt");
string s = tf.Read();
tf.Close();
```

Having defined the endpoints for each of the dimensions, we then used the evidence from our usability studies to determine the preferences of each persona with respect to each dimension.

After this determination, we reviewed the videotapes from each study and watched participants use different APIs. We knew the persona of each participant because we matched participants to our personas during our recruitment interviews. Thus, we were able to estimate persona preferences for each dimension. Although we didn't do any quantitative analysis of the events we observed in each study, we were confident that our subjective assessments of persona preferences would be reasonably accurate, given the large number of studies we were able to review. And we would often be helped by the participant's verbal commentary recorded during each session, which often made their preferences *very* clear.

As an example of the preferences we observed, we noted that most opportunistic developers seem to prefer an API with an aggregate abstraction level. Systematic developers, on the other hand, prefer the flexibility that primitive components offer. Pragmatic developers favor what we call *factored components*, which are effectively aggregate components with all the "modes" of operation factored out into different components.

We performed this analysis for each dimension until we had a profile for each persona. We then graphed out the profiles as shown in Figure 29-1.

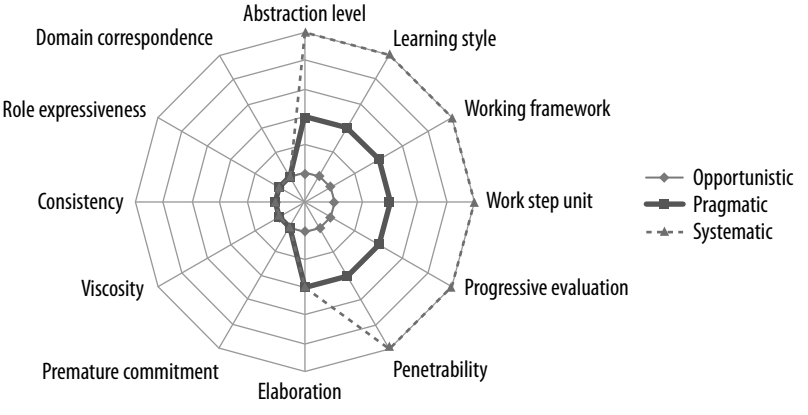


*FIGURE 29-1. Cognitive dimensions profile for each persona*

The graph makes it clear that each persona has a different view of API usability. The abstraction level preferences of opportunistic and systematic developers differ along the same lines as their learning styles, working frameworks, etc.

The graph was immediately useful at Microsoft in making clear that it would be very difficult, if not impossible, to design and build a usable API that targeted all three developer personas. This posed quite a challenge, however. Most of the teams designing and building APIs believed, correctly, that they *should* be targeting all three of the personas. If they wanted each persona to have a great experience with their API, what should they do? Should they build three versions of the API, one for each persona? The final piece of the jigsaw puzzle, scenario-based design, needed to be fit into place.

## Scenario-Based Design

When we refer to an API, sometimes we mean the complete set of classes that form the interface between the application and the programmer. In other words, every class that the developer could possibly use, regardless of whether she actually uses them, is considered part of the API. When the API design teams said that they were building an API that all three personas would use, it was in this sense that they were using the term.

The experience we learned from the ATL Server card sort study was that API usability is not determined by analyzing all of the classes in an API. The complete set of classes is what the API team thinks about, but the developers think about the scenarios in which they will use the API and the experience they have during those scenarios.

We saw this in action during the file I/O study. In that study, the participants didn't concern themselves with every single class in the .NET framework. Instead, they focused on the classes that would help them complete the task. Furthermore, we observed them dismiss certain classes that weren't at the expected abstraction level. The combination of the scenario and developer expectations served to filter out which classes they considered using.

So one way to deal with the problem of designing an API for all personas is to focus on the scenarios in which the API is expected to be used and the type of developer who will perform those scenarios. In many cases, the scenario will determine the type of developer and thus the type of API that should be designed for that scenario.

For illustrative purposes only, consider a general I/O API. Some of the scenarios the API supports might be:

- Reading a line of text from a text file
- Writing binary data to a file
- Performing memory-mapped I/O

It is unlikely that each of the three personas would perform all these scenarios. For example, opportunistic developers might be more likely than systematic developers to interact with text files because text files allow for rapid prototyping and experimentation (text files are quick and easy to create and modify in a text editor, so test cases, sample data, etc. can be created easily).

Pragmatic developers, on the other hand, may be more likely to work with binary data, since the mapping between the code that they write and the representation of data on disk is more direct. Being more code focused, pragmatic developers are attracted to frameworks that reduce the number of representations of data they need to work with.

Finally, systematic developers might be more likely to use memory-mapped file I/O, due to the flexibility that such an abstraction offers. With memory-mapped I/O, they can share memory between multiple processes and exercise full control over who has access to the memory.

In this simplistic example, the I/O API is indeed used by each of the three personas—but crucially, different parts of the API. Those different parts can be designed according to the persona that uses them. The text file component might expose aggregate components, for example, whereas the memory mapped component might expose primitives.

In scenario-based design, the API designers write out the code that they would expect a developer to write to accomplish each scenario. The designers take into account the preferences of the developer, in terms of the cognitive dimensions, as they write the sample code. In effect, they step into the shoes of the developer and try to behave as that developer would. At each step of the way, the API design team can use the cognitive dimensions framework as a checklist for what they are designing. They can review each code sample they have written and ensure that it truly meets the preferences of the developer to whom the scenario is targeted. Only when the design team is happy with the scenario do they start to design and build the API to enable the scenario.

Of course, once implementation begins, unanticipated questions arise about the design. On such occasions, the team refers to the set of scenarios they are building and considers the impact that the design question might have on the scenario. In doing so, the design team focuses on the *user experience* of the multiple APIs instead of the *architecture* of the total API. By striving to deliver a great set of experiences around the scenarios, design decisions can be made throughout the development of the API that are grounded in a good understanding of what it means for an API to be usable.

A scenario-based design approach to API design focuses the API designers' minds on what the API is for instead of what the API is. It makes it clear that designers are designing experiences that a developer has with a set of classes, rather than the architecture of the API. When considering API design decisions, the designers can focus on the scenarios in which the API is used and the type of developer using it.

## Conclusion

When we started looking at API usability, we didn't know how to do it or whether the effort would be worthwhile. We can now unreservedly state that it is absolutely possible and absolutely vital. We have learned a lot about how different developers approach APIs and the different expectations they have of those APIs. We have been able to utilize that understanding both to design new APIs and to discover and fix usability issues in existing APIs.

For example, the scenario-based design approach has been used at Microsoft for a few years now. Our design guidelines and API review process reinforce the need to do scenario-based design [Cwalina and Abrams 2005] and usability studies on APIs.

But this is only the beginning. There is much work still to be done. Given how critical it is to find and address API usability issues before the API ships, we need to develop additional techniques that can be used as early in the API development process as possible. For example, Farooq et al. describe a method for evaluating the usability of an API that does not require a usability study [Farooq and Zirkler 2010].

It's also important to understand how trends in application development impact the usability of APIs. For example, there is a trend toward the use of design patterns such as dependency injection [Fowler 2004]. There are very clear and sound architectural reasons for such patterns, but how do they affect usability? Jeff Stylos has investigated these issues and learned that there can be surprising usability consequences associated with some popular design patterns [Stylos and Clarke 2007], [Ellis et al. 2007].

Being in a position to ask and answer these questions is very satisfying. Now, over 10 years since I joined the Visual Studio User Experience team, I can appreciate how much we have learned about how best to create a positive impact on the developer user experience. I look forward to seeing similar advances in the future.

## References


[Blackwell 2002] Blackwell, A. F. 2002. First Steps in Programming: A Rationale for Attention Investment Models. *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*:2.

[Clarke and Becker 2003] Clarke, S. and C. Becker. 2003. Using the cognitive dimensions framework to measure the usability of a class library. *Proceedings of the First Joint Conference of EASE & PPIG (PPIG 15)*:359–366.

[Clarke 2007] Clarke, S. 2007. What is an End-User Software Engineer? Paper presented at the End-User Software Engineering Dagstuhl Seminar, February 18–23, in Dagstuhl, Germany.

[Cwalina and Abrams 2005] Cwalina, K. and B. Abrams. 2005. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*. Upper Saddle River, NJ: Addison-Wesley Professional.

[Ellis et al. 2007] Ellis, B., J. Stylos, and B. Myers. 2007. The Factory Pattern in API Design: A Usability Evaluation. *Proceedings of the 29th international Conference on Software Engineering*:302–312

[Farooq and Zirkler 2010] Farooq, U. and D. Zirkler. 2010. API peer reviews: a method for evaluating usability of application programming interfaces. *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*:207–210.

[Fowler 2004] Fowler, M. 2004. Module Assembly. *IEEE Software* 21(2):65–67.

[Green and Petre 1996] Green, T.R.G. and M. Petre. 1996. Usability Analysis of Visual Programming Environments: A "Cognitive Dimensions" Framework. *Journal of Visual Languages and Computing* 7(2):131–174.

[Robillard 2009] Robillard, M. 2009. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software* 26(6):27–34.

[Stylos and Clarke 2007] Stylos, J. and S. Clarke. 2007. Usability Implications of Requiring Parameters in Objects' Constructors. *Proceedings of the 29th International Conference on Software Engineering*:529–539.