Measuring API Usability

Usability of software tools impacts developer efficiency

Steven Clarke

The Visual Studio usability group at Microsoft, of which I'm a part, is responsible for helping design and build the user experience for developers using Visual Studio and the .NET platform. Much of our efforts at improving the developer experience have focused on improving the usability of the Visual Studio development tool itself. For example, we run extensive empirical studies in usability labs focusing on particular features of Visual Studio (such as the debugger), gathering data to help feature teams improve their designs. In this way, we can understand how best to present complex information, such as the structure of application data, to support developers better when they are debugging code.

But while improving the design of Visual Studio is a necessary component of improving the developer experience, it is not sufficient. The development tool is only one tool that developers routinely use when building applications. The other major tools are the programming language they are coding in (C++, C#, VB, and so on) and any class libraries or APIs that they are coding against (MFC, .NET Frameworks, and the like). The usability of the languages and libraries that developers use have a significant impact on their ability to successfully complete a set of development tasks. In this article, I describe some of the techniques we use to design and evaluate the usability of the APIs that ship with .NET, and discuss ways in which you can use them for APIs you are designing.

Steven is a usability engineer at Microsoft, working on Visual C++, the .NET Frameworks, and the WinFX libraries. Prior to joining Microsoft, he was a developer at Motorola, building development tools for Smartcard applications. Steven can be contacted at stevencl@microsoft.com.

Usability Applied to APIs

Most of us know what usability means when applied to software with graphical user interfaces (GUIs)— developer tools, word processors, or e-mail clients. We expect these tools to let us perform a given set of tasks and to work the way we expect them to. If I'm using a visual debugger, for example, I expect to be able to examine the state of a variable at some point during the execution of my code. I expect to be able to do this by using some debugger tool window that shows a list of variables that are currently in scope and selecting the variable that I want to examine. Or, if I am checking my e-mail, I want to see the new, unread e-mail that has arrived since I last read my e-mail. I expect to be able to do this by opening the client and having the new e-mail displayed in a way that distinguishes it from the old e-mail that I have read. If any of the products that we use fail to work in the way that we expect them to, we generally label this as poor design or usability.

There can be many causes for poor product usability or design; for example, users may not be aware of the actions that they can take with a product. Likewise, they may not be able to predict the result of performing some action and hence might be reluctant or less inclined to perform that action for fear of the potentially negative consequences that might ensue. Or they misinterpret the actions that they can take and the results of those actions.

Psychologists use the term "affordances" to refer to the actions that a user can take on some object. However, just because an object "affords" some action, it doesn't mean that the user will realize that they can take that action— users have to be able to perceive the affordances exposed by some object. Many usability breakdowns that occur in GUIs are due to affordances not being perceived by users. For example, for some users, the Start button in Windows XP is not perceived to afford the action of shutting the computer down.

Likewise, APIs expose affordances. Every API has a set of actions that it can perform. Therefore, usability problems can exist in an API that are related to users not perceiving the affordances the API supports. For example, an API for performing file I/O might afford creating a new file, writing to a file, reading from a file, deleting a file, copying a file, and moving a file; see Example 1. However, developers browsing the public interface of this API might not perceive the complete set of affordances that this API offers. In particular, they might not realize that they can use this API to move a file from one location to another by copying the file, then deleting the original. Or they may not realize that the constructor creates a new file, if the path specified in the constructor does not already point to an existing file.

Designing Usable APIs

One of the best ways to design usable GUIs or APIs is to follow a user-centered-design approach. User-centered-design encompasses understanding both your users and the way that they work in the scenarios the product is designed to support. Application implementation details do not drive the design of the user interface and should not be represented in the interface design. Instead, the design should reflect the way that users expect to work.

Designing usable APIs is similar to designing usable GUIs. You must make sure that you understand the characteristics of users and how those characteristics impact the way they expect the API to work. Following a scenario-based design approach ensures that the API reflects the tasks that users want to perform, instead of reflecting the implementation details of the API.

We follow a scenario-based design approach for all of our APIs. For each API, we write a collection of code samples that show how to use the API in a set of given scenarios. The API design then proceeds in such a way that it supports developers in writing these code samples. If a particular design decision makes it more difficult for developers to write the code contained in one of the samples, that decision is carefully reviewed. Additionally, APIs are reviewed periodically by examining the code samples and determining whether or not developers would be comfortable writing the code in each of the samples.

Just as for GUI design, scenario-based design for an API must start from some characterization of users. If the API design starts out with invalid assumptions about how developers write code, then the API will be less likely to work the way that developers expect it to. Instead of exposing the set of types and methods that developers will be expecting, it exposes a different set of types and methods whose purpose may not be so clear and that may be difficult to use. We use a framework known as the "cognitive dimensions" to describe the characteristics of developers that have an impact on the usability of an API.

Cognitive Dimensions

The cognitive dimensions framework presents a set of 12 different dimensions (or factors) that individually and collectively have an impact on the way that developers work with an API and on the way that developers expect the API to work. These dimensions are the:

- 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/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 amount 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 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.

The framework is used to describe both what a given set of users expect from an API and what the API actually provides. For example, we can describe the abstraction level that users expect to be exposed by an API and can compare this with the abstractions actually exposed by the API. It's the comparison between what developers expect and what the API provides that is interesting when evaluating the usability of an API. If there is a good match on each of the dimensions, we can feel reasonably confident that the API will meet the needs of that particular type of developer. However, if there are significant differences between what the API exposes and what the developer expects, we can concentrate on improving the API in the particular areas where differences exist.

To do such a comparison, we normally run a usability study in our usability labs. In a usability study, we ask a group of developers to use the API to perform a set of tasks. The tasks that the developers work on in the usability study are designed in such a way that they expose those areas of the API in which we are most interested in gathering feedback. Typically, the tasks relate to the main scenarios that the API is designed for; but sometimes, we focus on a specific area of the API to get detailed feedback on that one area.

After the study, we spend time reviewing the data. We look for patterns of behavior across all of the participants, whether they are patterns that led to success or failure on tasks. If the pattern led to success, we want to make sure that the API team is aware of the particular aspect of the API design that was responsible so that the successful design can be reused when appropriate elsewhere. If the pattern led to failure, likewise, we want to make sure that we do not repeat the same mistakes elsewhere in the design. In both cases, we use the cognitive dimensions framework to analyze and describe the results of the study.

Using the Cognitive Dimensions Framework

While we tend to use the cognitive dimensions framework to analyze the results of a usability study, you don't necessarily need to go to such lengths to get usability feedback on your own API. You can use the framework to analyze a set of code samples before the API has even been implemented.

For example, consider a fictional API for building e-mail clients. Example 2 demonstrates how to use this API to create a new e-mail message and send it. Developers using this API need to create instances of *IPAddress*, *CEmailServer*, *CMailbox*, and *CEmailMessage*. Furthermore, the *CMailbox* and *CEmailMessage* are created by calling factory methods exposed by the *CEmailServer* and *CMailbox* classes, respectively. Developers using this API need to create and configure all of these objects in conjunction with one another.

To review this API in terms of the cognitive dimensions framework for this given scenario, we ask a set of questions for each dimension. For example, looking at the work-step unit, we might ask:

- Does the amount of code required for this scenario seem just about right, too much, or too little? Why?
- Does the amount of code required for each subtask in this scenario seem just about right, too much, or too little? Why?

For progressive evaluation, we might ask:

- How easy is it to stop in the middle of the scenario and check the progress of work so far?
- Is it possible to find out how much progress has been made? If not, why not?

Or, for role expressiveness dimensions:

- When reading code that uses the API, is it easy to tell what each section of code does? Why?
- Are there some parts that are particularly difficult to interpret? Which ones?

```
class FileIO
{
public FileIO(string path){...}
protected ~FileIO(){...}
public void Close() {...}
public void WriteLine(string text) {...}
public string ReadLine(){...}
public void DeleteFile(){...}
public FileIO CopyFile(string newPath) {...}
}
```

Example 1: *API for simple file I/O.*

- When using the API, is it easy to know what classes and methods of the API to use when writing code?

By answering these questions, it is clear that there are a number of issues that arise related to the work-step unit, progressive evaluation, and role expressiveness.

Considering the work-step unit dimension, it's clear that developers must complete significant chunks of work with this API when they want to implement simple steps such as create an e-mail message or open a mailbox. For example, in the just mentioned API, you can't write code to create an instance of a *CEmailMessage* without writing all of the surrounding code to connect to a server, retrieve a mailbox, and open the mailbox. When the chunks of work are large enough, you have to use mental resources to keep track of the work that has been done and the work that remains to be done.

It is difficult for developers using this API to stop at some point before completing the scenario to check their progress. In other words, the API does not support progressive evaluation. Because developers cannot create an instance of a *CEmailMessage* without connecting to a server and opening a mailbox, there is no way for them to isolate the code they need to write to configure an e-mail message. The lack of progressive evaluation exposed by this API means that runtime errors can't easily be tied to specific lines of code or objects and makes debugging code written against the API more difficult. It also makes learning how to use the API difficult for some users because it is not really possible to explore and learn about individual types on their own.

Lastly, the role expressiveness is poor. There are at least three lines of code that are difficult to understand. Let's look at the first one:

```
CEmailServer theServer =
        new CEmailServer (theAddress);
```

Does this create a new mail server or an instance of the class representing an existing mail server? It's actually the latter, but the fact that we are creating a new instance of a class to represent an object that already exists could be confusing. Creating an instance of the *CEmailServer* class in this way might indicate that we are also creating a new e-mail server. Something that might be more expressive would be to use a static method that creates an instance representing an existing mail server:

```
CEmailServer theServer =
      CEmailServer.GetServer(theAddress);
```

```
IPAddress hostIPAddress = IPAddress.Parse("130.23.43.234");
IPAddress theAddress = new IPAddress(hostIPAddress);
CEmailServer theServer = new CEmailServer (theAddress);

if (theServer.Connect ())
{
      CMailbox theMailbox = theServer.GetMailbox ("stevencl");
      theMailbox.Open("stevencl", true);
      CEmailMessage theMessage = theMailbox.NewMessage();
      theMessage.Build("message", " subject", "to@you.com");
      theMailbox.Send (theMsg, true);
      theMailbox.Close();
}
else
      Console.WriteLine ("Could not connect to the mail server");
```

Example 2: *Using an API to create/send a new e-mail message.*

The second and third lines of code that are difficult to understand have the same problem — they use Boolean values as method parameters:

```
theMailbox.Open("stevencl", true);
theMailbox.Send (theMsg, true);
```

In these code snippets, how are we supposed to know the effect of passing *true* to both methods? There's really no way other than looking up the documentation with the effect that someone reading this code, say in a code review, could easily make the wrong assumption about what the code will do. One easy fix, of course, is to create a well-named Boolean variable, assign it the value *true*, and pass it in to the method instead of the literal Boolean value. One drawback to this solution, though, is that someone reading this code then has to trace back through the code to find the last point at which the variable is assigned before calling the method, to know what value is passed in. In some cases, another solution might be to use an *enum* to represent the set of possible options. For example, *CMailbox.Open* might take an *AuthenticationOption enum* exposing *NTAuthentication and UsernamePassword* values. In that case, the call would change to:

```
theMailbox.Open("stevencl",
   AuthenticationOption.NTAuthentication);
```

Using the cognitive dimensions this way, problems related to progressive evaluation, work-step unit, and role expressiveness were uncovered (note that there are other issues with this API, some that would be uncovered by considering other dimensions and others that would be uncovered by considering other scenarios). Some of the problems are easy to fix without a deep understanding of the developers using the API, such as the role expressiveness issues. The progressive evaluation and work-step unit issues, however, require a better understanding of what users expect from an API before being able to design a solution that meets a user's needs. Knowing what your users expect from an API provides you with a direction in which to chart the design of the API.

What Will Users Want?

In most cases, it's not always easy to figure out the right fix for a potential API usability problem. You may be able to come up with a solution, but it's difficult to know whether or not the fix will work for the developers that the API is designed for. This is why understanding your users is so important.

We use personas to represent our understanding of who our users are. In particular, the Visual Studio Usability group has defined three main developer profiles or personae that describe the stereotypical behavior of three main developer groups:

- Opportunistic.
- Pragmatic.
- Systematic.

For each profile, we have defined what developers matching that profile would expect from an API, in terms of the cognitive dimensions framework. Thus, we now have a way to judge whether an API is usable by comparing the API evaluation with the developer profile requirements. What's more, we are able to display such comparisons in graph form, as in Figure 1.

The black dotted line in the graph represents the API being evaluated, in this case the fictional e-mail API. The spokes in the "wheel" represent each of the different dimensions (spokes are numbered corresponding to the list of dimensions presented earlier. For example, work-step unit, progressive evaluation, and role expressiveness are spoke numbers 4, 5, and 11, respectively). End points (the middle and outer edges) on each spoke represent the end points on a scale for each dimension. Thus, the point on each spoke where the black dotted line crosses it corresponds to the point on the scale for that dimension that the API has been valued at.

The blue line on the graph represents a particular developer profile. For example, it shows that this persona prefers APIs that have small work-step units (the inner edge on the work-step unit scale), support progressive evaluation at the line of code level (the inner edge on the progressive evaluation scale), and have rich role expressiveness (the outer edge on the role expressiveness scale). If you compare the points where the blue line crosses each spoke with the point where the black line crosses each spoke, you get an idea of how well the API matches the persona's ideal API. Hopefully, you can see that the e-mail API doesn't really hit the mark for this persona (there are big differences between the points where both lines cross on most, if not all, of the spokes in the graph).

Cognitive dimensions have really helped us get to the root cause of issues that we have observed in the usability labs. For example, in one study, we observed lots of developers spending a large amount of time in the help docs looking for code samples that would show them how to accomplish a given task. The first interpretation of this data was simply "Fix the help docs!" However, when we used the cognitive dimensions framework to describe the issues, it became clear that the reason the developers weren't successful

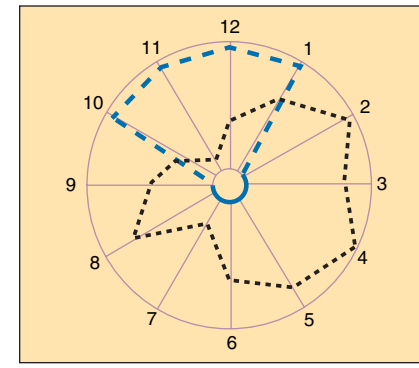

Figure 1: *Cognitive dimensions analysis comparing an API analysis with a developer profile.*

when they were searching through the help was because what they were looking for simply didn't exist. The API they were working with exposed a set of abstractions that were at the wrong level for these particular developers. They expected a particular type of abstraction to be exposed by the API but since it wasn't, they couldn't find anything about it in the help docs. As a result, the API team redesigned this API to expose abstractions more in line with what developers were expecting. When we retested the API, it worked much better.

At the very least, the cognitive dimensions framework provides developers with a common vocabulary with which to talk about and discuss API usability. The benefits of such a common vocabulary are twofold:

- First, the language shapes the distinctions that are considered important to attend to. By explicitly describing each of the dimensions, it is more likely that they will be attended to.
- Secondly, the language lets developers discuss issues using terminology that they can assume will be understood by others who are also familiar with the framework.

Conclusion

To ensure that you know how your customers expect your API to work, designing the API through scenario-based design is a starting point. You can also use the cognitive dimensions framework to describe how well the API meets your customers' requirements and to gather feedback from customers. You don't need expensive usability labs to be able to do good usability studies. All you really need are time, patience, willing participants, and a framework with which to understand the results of your analysis.

DDJ