Using the Cognitive Dimensions Framework to evaluate the usability of a class library

Steven Clarke and Curtis Becker
Microsoft Corporation
41/2755
1 Microsoft Way
Redmond
WA 98052
stevencl@microsoft.com *a-curtb@microsft.com*

Abstract

In this paper we describe our attempts at using the Cognitive Dimensions framework to evaluate the usability of an object oriented (OO) application programming interface (API). The Cognitive Dimensions framework was originally designed as a tool to evaluate and describe the usability of a programming language but to our knowledge, no reports of it being used to evaluate an API are publicly available. Since many popular OO languages come with an existing set of APIs in the form of OO class libraries, it is important to ensure that the APIs developers use are usable, just as it is to ensure that a programming language is usable also. Thus, in order to measure the usability of an API, we have adapted the Cognitive Dimensions framework so that it better measures those aspects of an API that we feel have an impact on its usability. This paper describes these modifications and the reasons we chose to make them. In addition, using the framework in an industrial setting has presented some interesting challenges, and so we describe those challenges here and our attempts to address them.

Keywords: POP-III.D. class libraries, POP-II.B. coding

Introduction

The Visual Studio usability group at Microsoft is responsible for helping build a great user experience for developers using Visual Studio and the .Net platform to build applications. 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 the 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 for example, we can understand how best to present complex information, such as the structure of application data, in order to support the developer better when they are debugging their code. However, 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 particular programming language they are coding in (C++, C#, VB etc.) and any class libraries or APIs that they are coding against (MFC, .Net Frameworks etc.). The usability of the languages and libraries that developers use will obviously have a significant impact on their ability to successfully complete a set of development tasks. We have recently shifted part of our focus to consider the class libraries and APIs that developers use. We want to ensure that we provide developers with the most usable tools we can, be they parts of the developer environment, such as the debugger or project system, or the APIs that developers code against.

We have made use of the Cognitive Dimensions framework (Green and Petre, 1996) in order to study the usability of class libraries and APIs. In this paper we describe modifications we have made to the framework in order to apply it to the design of class libraries. We also describe how we have presented the framework to the different teams across Microsoft responsible for implementing class libraries, such as the .Net framework. We conclude with a discussion on our experiences using the framework so far and challenges that we still have to overcome.

Applying the Cognitive Dimensions Framework to class libraries

In order to understand what it means for a class library to be usable, we looked at the Cognitive Dimensions framework and considered whether or not we would be able to use it to describe the usability of a class library. We had already used the framework when studying C#, making use of the cognitive dimensions questionnaire as a means for participants to self report issues that arose during a usability study, (Blackwell & Green,

2000), and had seen some success with using the framework as a vocabulary for discussing programming language usability. One of the benefits of having such a shared vocabulary is that it can be used to describe the results of specific usability studies in such a way that the results can be generalized across different scenarios. One API team can learn something from the study, even if it was another team's API that was being studied. We wanted to be able to have the same conversations about class library usability, enabling all class library developers throughout Microsoft to benefit from usability studies on a particular library by discussing the results in terms of the Cognitive Dimensions, as applied to class libraries.

Before we could do so, we wanted to make sure that the vocabulary of the Cognitive Dimensions framework would make sense in the context of discussions about class library usability. We were concerned that since the framework had been developed to discuss programming language usability, it may not be as applicable when discussing class library usability. Additionally, and just as importantly, we had to consider the audience we would be delivering this to. We had to make sure that the terms and concepts described in the framework would make sense to class library developers and designers at Microsoft. We also had to ensure that the results of studying a class library in terms of the framework would be actionable, and would lead to direct changes or improvements to the class library. We did not want to produce a framework that was useful for talking about the usability of a class library in general, but which was not as useful when talking about specific class libraries.

We were fortunate to have already been given somewhat of a head start. We had already run two studies of a class library in our usability labs before we started to seriously consider using the cognitive dimensions framework. The first study looked at how successful users would be in a given number of tasks using a class library that had already been designed and implemented. The results from the study were used to design the second version of the class library and a second usability study was conducted to measure the impact of the changes made to the library. The improvements between the 1st and the 2nd version of the class library were significant. As a result, the benefit of running

usability studies on a class library were demonstrated, at least among the majority of teams at Microsoft responsible for designing and developing class libraries.

The studies provided us with a source of data about class library usability. We looked at the Cognitive Dimensions framework to see if we could describe these results in terms of the framework.

New Dimensions

We wanted to know if the Cognitive Dimensions framework would be a useful framework within which to discuss class library usability. However, in trying to describe the results from a previous class library usability study, we uncovered the following issues:

1. Many of the results from the studies weren't easily described as being artifacts of one or more of the cognitive dimensions;
2. We felt that some of the names used for each of the dimensions might not translate so well into an industrial setting.

We found that many of the results from our usability studies were related to the number of classes that a developer has to work with and instantiate in order to accomplish a task. If this number is too high, it can be difficult for the developer to work out what classes they need and the order in which they need to use them. This becomes an issue in cases where the developer attempts to make a call to a method that takes a parameter of some type and creating that parameter involves creating and manipulating a succession of helper classes. The amount of code that needs to be written may be considered excessively high in such a scenario.

We initially considered describing this in terms of the diffuseness dimension. However, we felt that there was more going on than just suggesting that the code required for this particular scenario was too verbose. Instead of just focusing on ways in which to reduce the amount of code required for a scenario, (e.g., through providing smaller method and class names, providing default constructors for classes), we wanted to focus on the

relationship between the amount of code that the developer needs to write and the task that developers are attempting to accomplish. We observed that in many cases, developers had difficulties with creating and manipulating multiple classes to accomplish some task not because the amount of code they had to write was excessive, but rather it was unexpected. Instead of having to use multiple classes in combination with one another for a given task, developers expected instead to have a minimum number of classes that they could use to accomplish the same task.

Such an issue might also be explained in terms of the abstraction level dimension. However, in the context of discussing class library usability, we wanted to ensure that the abstraction level dimension was used to discuss the overall magnitude of abstractions exposed by the class library and the style of most of those abstractions. We did not want to overload the term with discussions about the number of abstractions used in a particular scenario.

We decided to create our own dimension, *Work-Step Unit*, to describe the amount of work that a developer needs to do to accomplish a particular step. We describe this dimension on a scale where one end describes a small, locally focused amount of work (i.e., one line of code written in the context of some local method) to the other end of the scale which requires the developer to create and manipulate multiple objects or classes in parallel in order to accomplish their task.

Related to this dimension, we introduced another which we call *Working Framework*. This refers to the amount of context that developers have to maintain about the code that is being written. If developers are working with a reasonably complex API, they may be required to keep mental track of the impact that the code they are currently writing will have throughout the rest of their application. Even if the API supports a small work-step unit (i.e., most tasks can be accomplished in code through writing a small number of lines of code in a local context) it might still be important to keep track of the impact of this code on other components of the overall application. For example, registering interest in a particular event fired by some class or system component may only require one or two

lines of code, but it is important that the developer realize the impact that registering for that event will have on the ability of the rest of the application to respond to that same event.

We also took the liberty of renaming some of the dimensions. We felt that some of the names given to the dimensions would not translate as well into the working environment at Microsoft. We had no data to suggest that this would be the case, other than a couple of random straw-man polls with colleagues in the hallways. Their reactions upon seeing the original names for the cognitive dimensions suggested to us that it might be worthwhile renaming some of the dimensions to make their purpose and applicability clearer. We also tried to cut down the number of dimensions to include just those that have a direct bearing on class library usability.

The new list of Cognitive Dimensions for analyzing class library usability is shown below.

1. **Abstraction Level:** what are the minimum and maximum levels of abstraction exposed by the API, and what are the minimum and maximum levels usable by a targeted developer.
2. **Learning Style:** what are the learning requirements posed by the API, and what are the learning styles available to a targeted developer.
3. **Working Framework:** what is the size of the conceptual chunk needed to work effectively
4. **Work-Step Unit:** how much of a programming task must/can be completed in a single step.
5. **Progressive Evaluation:** to what extent can partially completed code be executed to obtain feedback on code behavior?
6. **Premature Commitment:** to what extent does a developer have to make decisions before all the needed information is available?
7. **Penetrability:** how does the API facilitate exploration, analysis, and understanding of its components, and how does a targeted developer go about retrieving what is needed.
8. **API Elaboration:** to what extent must the API be adapted to meet the needs of a targeted developer?

9. **API Viscosity:** what are the barriers to change inherent in the API, and how much effort does a targeted developer need to expend to make a change.

10. **Consistency:** once part of an API is learned, how much of the rest of it can be inferred?

11. **Role Expressiveness:** how apparent is the relationship between each component and the program as a whole?

12. **Domain Correspondence:** how clearly do the API components map to the domain? Are there any special tricks?

As can be seen, most of the dimensions have been used without modification from the original list. Two of the main new dimensions we have introduced were described above. We have also introduced the *Learning Style* and *Penetrability* dimensions and renamed Abstraction Barrier to *Abstraction Level* and *API Elaboration*. Lastly, we renamed viscosity to *API Viscosity*.

This list is definitely not complete. However, we have been able to use these dimensions to describe the results of three different usability studies that we ran after creating the list.

Making use of the Cognitive Dimensions

Our approach to using the cognitive dimensions framework in running class library usability studies and reporting on them has been reasonably straightforward. We work with the team developing the class library to figure out the core scenarios their library should support, and then design an empirical study to investigate how easily developers can work on those scenarios using the class library. We create a set of development tasks for participants to work on. We videotape each participant while they work on the different tasks. We then analyse the data to look for interesting patterns of behaviour across participants and to look for breakdowns in the design of the class library. We go through each of the dimensions and describe the major findings from the study in terms of these dimensions.

So far, we have been able to use the dimensions in this way with a satisfactory level of success. Class library developers throughout Microsoft have been exposed to the list of cognitive dimensions and have seen how they impact class library usability by seeing

examples of each dimension in terms of specific results from individual usability studies. Thus the Cognitive Dimensions is not some abstract framework which is difficult to relate to real class library design. Instead, it is a useful tool that allows class library developers to reach a shared understanding of and talk about different aspects of class library usability.

Further challenges

We have introduced the vocabulary of Cognitive Dimensions to class library developers. Many developers around Microsoft are now actively investigating how usable their class libraries are and are talking about the results of studies in terms of the Cognitive Dimensions. This is a big step and we are pleased that we have been able to accomplish this. The next step however is to try to encourage the use of the Cognitive Dimensions framework as a design tool. We would like to see the influence of the framework in the original design of a class library at Microsoft, not just in the way that the class library is studied in the usability lab.

References

Green, T. R. G. & Petre, M. (1996) Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *J. Visual Languages and Computing*, 7, 131-174.

Blackwell, A.F. & Green, T.R.G. (2000). A Cognitive Dimensions questionnaire optimised for users. In A.F. Blackwell & E. Bilotta (Eds.) *Proceedings of the Twelth Annual Meeting of the Psychology of Programming Interest Group*, 137-152.