In G. Kadoda (Ed). Proc. PPIG 13 Pages 275-289

Evaluating a new programming language

Steven Clarke
Microsoft Corporation
1 Microsoft Way
Redmond, WA 98052 USA
+1 425 705 5978
stevencl@microsoft.com

Keywords: POP-III.B. *C#* POP-III.C. *all cognitive dimensions*
POP-V.B. *questionnaire*

Abstract

We describe our efforts using the Cognitive Dimensions framework to evaluate a new programming language. We used a questionnaire approach to gather framework data related to the new language and combined this with a traditional lab-based evaluation. With this approach, we were able to validate the findings from the cognitive dimensions questionnaire by correlating them with the findings from the lab-based study. From this, we were able to determine if it is feasible to perform such an evaluation with usability participants who have no prior experience of the language being evaluated. Our findings have implications for anyone intending to use the cognitive dimensions to evaluate new notations or programming languages.

Introduction

To evaluate the usability of a new programming language, called C#, developed at Microsoft Corp., we used the Cognitive Dimensions framework (Green and Petre, 1996). This paper describes our experiences using the framework to evaluate a new language for which no experienced users yet exist.

The paper starts by presenting a brief overview of the Cognitive Dimensions framework. Following this, the next section describes the study undertaken, followed by the analysis of the data gathered from the study. Lastly, a discussion of the results of the study and a description of the further work suggested by the study is presented.

Using the Cognitive Dimensions

Cognitive Dimensions is a framework for analysing the usability of a notation, or programming language. The framework presents thirteen dimensions upon which judgments are made regarding cognitive demands made by a particular notation or programming language. Due to space constraints, the dimensions are not described here. Interested readers are referred to Green and Petre, (1996) for more detail.

Having evaluated a number of different methods for using the cognitive dimensions (Clarke and Spencer, 2000; Cox, 2000; Modugno *et al*, 1994; Roast and Siddiqi, 1996) we settled on the questionnaire approach described in Blackwell and Green (2000). The questionnaire is neutral to the language being evaluated and so we did not need to do any additional work to be able to apply it in our situation.

However, one issue with using the questionnaire was whether or not we would be able to use this successfully with a set of participants who have no experience with the programming language being evaluated. Another issue was validity. How could we determine the validity of the results obtained from applying the cognitive dimensions in this way? In the next section, we describe the study we carried out, and how we attempted to deal with participant's lack of experience with the programming language and the issue of validity.

Methods and Procedure

The study offered an opportunity to answer a number of research questions. First, we wanted to determine how effectively we could use the cognitive dimensions questionnaire in cases where participants have no experience with the notation or programming language being evaluated. Secondly, we wanted to be able to assess and demonstrate the validity of this method of evaluation.

And thirdly, we wanted to learn about the kinds of problems that users might experience when programming with the new programming language, C#.

New Features of C#

Before describing the study in detail, it is worthwhile to describe those features of C# that we were most interested in studying in a usability study.

C# is a new object oriented language, based on C++. While the syntax of the language is similar to C++ there are a number of new features:

- Properties. These are standard get and set member variable functions that allow programmers to write code that looks like they are directly accessing class member variables. This is intended to make some code easier to read and write.
- Versioning. Introduction of a new member in a base class with the same name as a member in a derived class is not an error in C#, unlike other languages, such as C++. A class must explicitly state whether a method is intended to override an inherited method, or whether a method is a "new" method that simply hides a similarly named inherited method. This is intended to make it easier for programmers to adapt to changing requirements.
- No pointers. C# programmers do not manipulate memory directly. Unreferenced memory is garbage collected automatically, and all references to objects are by-reference, rather than by-value.

Subjects

Five professional programmers participated in the study. Four had at least two years of professional experience programming in C++. One participant was a novice, and had six months experience working as a professional developer, having recently graduated from High School.

Test Methods

In order to address both the issues of experience and validity, we decided to run a traditional laboratory based usability evaluation, and to administer the cognitive dimensions questionnaire after each lab study. Each participant was given a specification for an online bookstore application. Participants were given three hours to implement the specification as best they could and were free to tackle the specification in any style they liked. They were not asked to ensure they completed the specification in the time given, just to do as much as they could in the time given. Verbal protocol was recorded for each session, and sessions were recorded on videotape.

After the three hour session in the usability lab, participants were given an hour to complete the cognitive dimensions questionnaire. The questionnaire was administered online, and participants answered the questions by typing in their responses. Participants did not talk aloud while answering the questionnaire, but they were able to ask questions about the questionnaire, for example to clarify any terms used. The questionnaire did not explicitly mention any of the cognitive dimensions. Each set of questions corresponding to one of the cognitive dimensions was presented on separate pages. Participants were provided with their choice of two items of Microsoft software for their participation in the study.

We attempted to review the answers given by respondents to generate numerical data to compare the number of issues that were reported by each participant, numbers of unique issues, and numbers of similar issues. However, it was too difficult to agree upon a useful definition or granularity for an issue such that this data could be collected. Also, each respondent interpreted the questions differently, and worded their answers differently, so it was sometimes difficult to tell if the answers referred to the same issue.

Instead, we compared the verbal protocol collected for each participant to the answers given in the questionnaire. In comparing the two sources of data, we were interested in highlighting those areas where, for each appropriate dimension:

- Observation and questionnaire answers overlapped (i.e., participants reported on issues that had been observed and recorded in the lab);
- Observations made were not reported (i.e., particular issues were observed in the lab, but were not reported on by participants in the questionnaire);
- Observations were not made, but participants reported on them in the questionnaire (i.e., participants reported on some issue, but it had not been observed in the lab.)

For each dimension, we looked through the responses given by the participants, and reviewed the verbal protocol to see if the issue raised from the questionnaire had been recorded in the verbal protocol. We then looked through the verbal protocol to see if there were any issues recorded that had not been mentioned in the questionnaire. Some of the issues mentioned earlier, which made comparing participants responses difficult, also arose here. For example, it was difficult to be certain that an issue raised in the questionnaire was the same as one described in the verbal protocol. However, we only compared participants' responses with their own verbal protocol, rather than comparing responses between different participants. This helped reduce differences due to participants varying interpretation. However, the fact that participants gained in experience with the language, even after three hours, was taken into account when comparing the responses with the verbal protocol.

Results

In this section, we describe some of the issues that fell into each of the above three categories. For the sake of brevity, we do not describe issues for each of the thirteen dimensions. We present here only those we feel are of most interest.

Viscosity

While in the lab, participant 2 had difficulties using the Visual Studio editor to match the braces marking the start and end of a method he was implementing. The method took up more space than could be displayed in one screen. In the questionnaire, participant 2 explicitly mentioned this problem: "*I had trouble tracking whether the member I was adding was still inside the class declaration or whether it had slipped past the closing brace*"

In the questionnaire, participant 3 described how he couldn't see the whole class hierarchy for the system classes that he was working with, leading to difficulties finding information on one of the system classes he wanted to use. In the verbal protocol collected however, no mention of this issue was made.

Participant 4 was observed using the Intellisense feature (see figure 1), a feature of Visual Studio that shows the functions and variables that can be accessed by a particular object. This feature addresses the visibility of members of objects contained in code. However, no participant mentioned the effect this feature had with respect to visibility in their responses to the questionnaire, even though they were observed using it frequently.

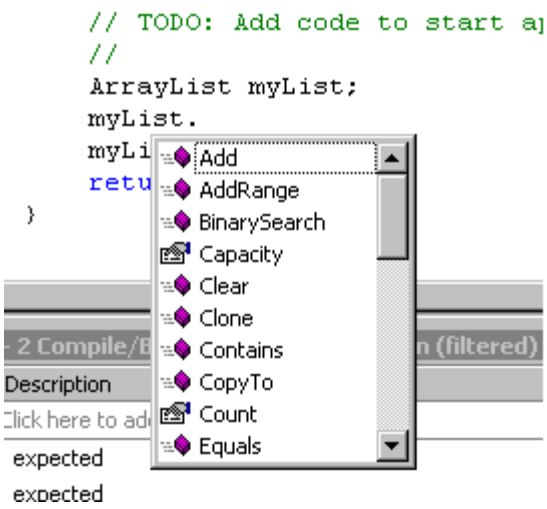

Figure 1 The Intellisense feature shows all the members of the ArrayList object, myList. Intellisense is invoked whenever the user references an object by typing its name followed by a dot, in the source editor

Diffuseness

When programming in C#, programmers must declare an access modifier for methods they declare in a class. Each method must have its own access modifier (i.e., methods must be declared as public or private). This is unlike C++, where an access modifier can be made to apply to a group of methods. In the usability study, participant 1 forgot to provide the access modifier while he was declaring his methods. This was recorded in the verbal protocol: "*Oh, these need to be declared as public. The reason I'm forgetting that is because in C++ you just declare everything as public or private. Its just habit not to put that in front of everything*". In the questionnaire, the participant explicitly mentioned this problem while answering questions related to the diffuseness dimension: "*Having to declare each member as public or private seemed a bit long winded.*"

Closeness of mapping

In C#, programmers do not manipulate pointers to memory. This has been identified as one of the major causes of error in languages such as C and C++. However, in order to be able to utilize the power that pointers provide in certain scenarios, some additions have been made to the language. One powerful feature of C and C++ is the pointer to function variable, which allows a variable to point to different functions. In C#, programmers can still manipulate pointers to functions through the delegates structure. Delegates are basically a class wrapped around a pointer to function variable, providing lots of safety mechanisms so that programmers do not make some of the same mistakes that used to be made with pointer to function variables.

The BookStore specification required that participants use the delegates feature to implement certain parts of the application. All participants were observed having difficulties grappling with the syntax required to code the delegate. In the questionnaire, participant 3 explicitly mentioned this while answering the questions related to the closeness of mapping dimension: "*The language constructs that gave me the most trouble were delegate and event. I have written "C" code that does exactly what I was trying to do in C#. I like the idea of "delegate", having a type-safe way to do function pointers, but the syntax was strange and it was hard to figure out how to use it in a simple case*". Clearly, participant 3 understood what the delegate feature was to be used for. Indeed, he was able to relate it to his experience writing C code that implemented the same functionality. However, he could not transfer his experience and understanding to be able to implement it correctly in his C# code.

Hidden dependencies

In C#, like most object oriented programming languages, programmers have available to them a large array of existing classes that they can use and specialize in their own code. Such class libraries form what is known as a class hierarchy. The hierarchy shows how all of the available classes relate to

each other. In some languages, the class hierarchy contains one object at the top of the hierarchy, from which every object in the hierarchy derives from, or is related to. In other languages, no such top-level object exists, and classes can be placed in the hierarchy without having any explicit relationship to any other class in the hierarchy.

In C#, all classes are derived from, or related to a class called Object. No class can exist totally in isolation from any other class. This means that all classes can be guaranteed to implement certain basic functionality. This functionality is placed in the Object class.

In C#, while classes must be derived from Object or from some other class that itself is derived from Object, classes that derive from Object do not have to specify this when they are declared. The relation with Object is implicit, if no other relationship to any other class is made explicit.

In the lab-based study, participants were given a set of existing classes that they were to use while implementing their own classes. While examining the classes, participant 4 noticed that one of the methods in the Book class was using the definition of the method in its parent class. The participant looked at the Book class definition to find out the name of the parent class that Book was derived from but no class was specified. The class declaration made it look as if the class existed in isolation from other classes. In actual fact, the class was derived from Object (as described above, if no explicit relationship is declared, the class is assumed to derive from Object). However, participant 4 was not aware of this and could not work out how the Book class could use the definition of a function contained in a parent class when it looked like the Book class was not derived from any other class: "*So it has a property Equals. It's overriding it from…I'm not sure what.*" However, when answering questions related to the hidden dependencies dimension, participant 4 did not mention this difficulty at all.

Related to the issue of overriding member functions, is the issue of how to specify that a function can be overridden. In C++, if the programmer wants a function to be able to be overridden, they declare the function as virtual. By default then, child classes can override the function in the parent class. However, this can cause maintenance difficulties, especially if new functions are added to the parent class with the same name as functions that have been defined in child classes. To get around these issues, in C# functions are not overridden by default, even if the parent class has declared the function as virtual. Instead, the child class has to declare the child version of the function using the override keyword, to explicitly state that this function is overriding the function in the parent class.

In the questionnaire, participant 1 mentioned the effect this had on his working style, while answering questions related to the hidden dependencies cognitive dimension: "*Also, the inheritance operator made it clear what was to be inherited from, but since the virtual keyword still exists, it took back referencing to see what could and could not be inherited from. Generally, I desire all methods to be inherited, and non inherited methods are the exception, not the rule, so some method of calling those types of dependencies out would be useful*". However, in the verbal protocol collected during the lab-based study, no mention of this problem was made.

In order to use the system classes mentioned earlier, programmers programming in C# have to use the using keyword. The using keyword makes certain classes available to the programmer. Since there are so many system classes, they are organized into structures called namespaces. Namespaces are like folders in a file structure. The namespace contains the definitions of classes, as well as other namespaces. When a programmer says that they are using a particular namespace in their program, with the syntax 'using myNameSpace;' they can use all of the classes that exist in the namespace myNameSpace. Thus there is a dependency between the class and the namespace it is contained within. In the lab based study, participants were observed having difficulties working out how to explicitly state the dependency between the class and the namespace. Participant 1 thought he had to say he was using the class, rather than the namespace that contained the class. To use the class Book, participant 1 typed 'using Book;' instead of 'using BaseClasses2;' (BaseClasses2 is the namespace the contains the definition of class Book). In the questionnaire, participant 1 mentioned this while answering questions related to the hidden dependencies dimension: "*The "using" keyword was helpful in showing the larger dependencies*".

Discussion

There are three questions that arise from this work. The first question concerns whether or not the questionnaire has any value. The second concerns how best to interpret the data returned from the questionnaire. Lastly, the third issue concerns how much coverage the questionnaire provides, over all of the usability issues that exist in the language. To be of value to usability engineers, or language designers, the questionnaire has to show that it can collect valid data regarding the usability of the language, in at least as efficient a manner as other techniques.

In this regard, one thing that the cognitive dimensions questionnaire has in its favour is its uniqueness. To the author's knowledge, there are few tools specifically designed to measure the usability of a programming language. Thus, the questionnaire is a valuable tool, given the lack of alternatives.

More importantly however, is the value of the data the tool gives us. Even if the questionnaire were the only tool available for performing evaluations of this nature, its value would not be considered so great if the data it returns was questionable or difficult to interpret. In the previous section however, it is clear that the questionnaire gave us valid and valuable data. The questionnaire highlighted issues that were observed and recorded in the lab. This demonstrates that the questionnaire can be used to collect data that can also be collected through observation in a lab setting. It also validates at least a subset of the data collected using the questionnaire. Further enhancing the value of the questionnaire is the data the questionnaire gave us that the verbal protocol did not. For example, in terms of visibility, the data returned from the questionnaire suggested that users would have difficulties seeing all of the system classes that are used when programming in C#. The verbal protocol collected did not reflect this issue. Balancing this however, is the data that the verbal protocol collected but that the questionnaire did not. For example, participants in their responses to the questionnaire did not mention the advantages of using the Intellisense feature, in terms of visibility. One reason for this could be that the questionnaire does not encourage positive responses with respect to each of the dimensions.

If we believe that the questionnaire has value, the next question that needs to be answered is what does it tell us? How do we interpret the results from the questionnaire?

It is clear that the data cannot be viewed as a list of usability problems. There are two main reasons for this. First, the dimensions do not exist in isolation from one another (Green and Petre, 1996). Rather, there are tradeoffs that have to be made between different dimensions. For example while the Intellisense feature (described earlier) helps increase visibility by showing all of the member functions and methods that an object or class supports, one participant reported in the questionnaire how it forced him into a particular order of working. In order for Intellisense to work, classes and objects must already be defined. There is no way Intellisense can show information for an object that does not exist. The participant wanted to start working with the top-level class in the specification. However, this class used all of the lower level classes. Without a definition of these classes, Intellisense could not be invoked to provide useful information. Thus the participant was forced to define the lower level classes before he could work on the top-level class. Therefore the feature was detrimental with respect to the premature commitment dimension. However, there is no way to resolve the tradeoff without being able to judge confidently the effect that the feature has on users abilities to complete certain tasks. This leads to the second point, concerning the neutrality of the questionnaire with respect to users tasks.

The questionnaire, by its very nature, is task neutral. None of the questions it contains are related to specific tasks. Thus, it is impossible to collect data regarding participants' abilities to use the notation to complete certain tasks. Without such data, it is impossible to confidently predict whether or not users will have problems with a certain feature when attempting to carry out a task. Without knowing the effect that an issue has on a user's ability to complete a task, we cannot evaluate how much of a problem the issue will cause. With the data from the questionnaire, the best we can do is to speculate about the impact on the tasks that users will attempt.

However, the data does identify areas where further usability testing or investigation could be useful. For example, in the above analysis, the tradeoff regarding the Intellisense feature and the visibility and premature commitment dimensions could be resolved by collecting data from two appropriately designed experiments. In one experiment, data related to the tasks the feature is designed to support

could be collected. In the other experiment, data related to the effect of influencing the order of development could be collected. Only when we have such data can we make judgments about how best to resolve the tradeoffs between the dimensions. The data from the questionnaire alone does not allow us to answer such questions.

The last question remaining for discussion relates to issues of coverage and the format of responses requested from respondents. How many of the usability issues that relate to the language being evaluated are uncovered by the questionnaire? The answer to this question depends on a number of factors, including the number of respondents, how experienced respondents are with the language in question and the format of the responses given.

In the study reported here, it is difficult to accurately estimate how comprehensively the questionnaire exposed the usability issues with respect to the C# language. One way to estimate coverage would be by attempting to compare the number of unique issues raised by respondents with the number of similar issues raised. Once respondents start to regularly report issues that have already been reported by previous participants, it is clear that reasonably good coverage has been obtained. In order to be able to estimate similarity however, we really need a solid definition of an issue, and what it means for one issue reported by one participant, to be similar to another issue reported by another participant. For example, can a similar issue be reported for different dimensions, between different participants? What level of detail should respondents be asked to describe? Responses must be at the same level of detail in order to make reasonable comparisons.

The amount of experience respondents have, not only with the language being evaluated, but also any related or similar languages, is also important. In this study, all of the participants were completely inexperienced with C# but, apart from one participant, had at least two years of experience with C++, a closely related language. Thus, most of the issues raised by the questionnaire concerned many of the issues that would be encountered by a newcomer to both the language and the development tool. For example, issues related to new and strange syntax and issues related to using the different features of the development tool such as the help system. It is likely that if participants were more experienced with the language, a different set of issues might have been raised. However, given the participants experience in C++, some of the issues raised were related to expectations for similarities with C++. For example, participants expected that the syntax for declaring a class to be derived from another class would be similar to that used in C++. Likewise, given a different set of participants with different experiences with other programming languages, it is likely that a different set of issues would have been raised.

Conclusions

We believe we have demonstrated that the cognitive dimensions questionnaire is a useful and valuable tool to use in performing a usability evaluation of a new programming language. We have described our usage of the cognitive dimensions questionnaire in a usability evaluation of a new programming language called C#. The questionnaire helped us to identify potential areas where further usability work would be most beneficial.

By carrying out a traditional laboratory based evaluation, and comparing the verbal protocol collected with the participants answers to the questionnaire, we were able to attempt to validate the questionnaire. Such a comparison, while not without difficulties, demonstrated to us that the answers given by respondents to the questionnaire do indeed describe real usability issues worthy of investigation, rather than having been made up by respondents simply to fill in space on an empty questionnaire. We were also encouraged to find that some issues described in the questionnaire had not been recorded in the form of participants' verbal protocol. We argue that this suggests the questionnaire is a useful tool for collecting some kinds of data that other kinds of usability tools may not be likely to collect.

However, we conclude that while the questionnaire is indeed a useful tool, careful interpretation of the results obtained from using it is required. The responses to the questions do not describe a list of usability problems, since the questions are neutral with respect to the tasks the users might be expected to be able to carry out with the language. Without any indication of the effect the issue has on the users ability to carry out certain tasks, we have no way of judging whether or not the issue is a usability problem, nor how severe it may be. Also, since there are trade-offs between each of the cognitive dimensions, which form the basis of the questions in the questionnaire, care must be taken

in interpreting the responses to ensure that relevant tradeoffs are identified, before any attempt is made to identify usability problems.

Further Work

Much work still remains to be done. Questions related to the coverage of usability issues offered by the questionnaire need to be answered. We need to know how many participants should be asked to complete the questionnaire and what amount of experience the participants should have with the language and related languages, for the data collected from the questionnaire to be most effective.

We need a better format for participants to provide their responses in, so that more detailed and comprehensive analysis of the answers from the questionnaire can be obtained. This entails defining, for the purposes of this questionnaire, what a usability issue is, and how best to describe it. Once such a format is described, it could be used to help perform more detailed validations of the issues raised by the questionnaire.

Ackowledgements

The author would like to thank his colleagues for their insightful comments and feedback on earlier drafts of this paper.

References

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

Clarke, S. and Spencer, R. Usability testing object oriented class libraries, *Industry Day & Adjunct Proceedings, HCI 2000*, Sunderland, UK, 2000

Cox, K. Cognitive Dimensions of Use Cases – feedback from a student questionnaire, In Blackwell, A.F., & Bilotta, E., (Eds.) *Proceedings of the Twelfth Annual Meeting of the Psychology of Programming Interest Group*, 2000, 99 - 122

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

Modugno, F., Green, T.R.G., and Myers, B.A., Visual Programming in a Visual Domain: A Case Study of Cognitive Dimensions, *Proceedings of HCI '94*, Glasgow, UK, 1994

Roast, C.R., and Siddiqi, J.I. The Formal Examination of Cognitive Dimensions. In Blandford, A. and Thimbleby, H. (Eds.) *Industry Day & Adjunct Proceedings HCI '96*, London, UK, 1996