Class Libraries: A Challenge for Programming Usability Research

Kerry Rodden and Alan Blackwell
University of Cambridge Computer Laboratory
15 JJ Thomson Avenue, Cambridge CB3 0FD, UK
{Kerry.Rodden, Alan.Blackwell}@cl.cam.ac.uk

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

Abstract

In previous collaboration with the Visual Studio usability team at Microsoft, we have learned that the Microsoft Foundation Classes are considered central to the usability of their products. There is little research in psychology of programming that is directly relevant to the design and evaluation of class libraries, despite the fact that they clearly occupy a central place among the cognitive challenges faced by professional programmers. Research into software reuse has considered some of the human factors in deploying class libraries. But the MFC library, despite being (probably) the most widely reused code in the world at present, has rather different problems from those addressed in reuse research. In this paper we analyse the nature of those problems, identify promising research avenues, and propose a challenge for future research in evaluating and improving the usability of class libraries.

Introduction

A central tenet of psychology of programming is that programmers are users too. Not in the sense that there are (end) users who happen to do programming, but that professional programmers are the users of programming environments and thus, like all users, deserve tools designed with attention to usability. It has long been observed that only a tiny fraction of the effort devoted to programming language research studies the behaviour of the programmer – more than 99% (judging by publication rates) studies the behaviour of the machine. This is despite the fact that programming languages are often described as communication channels between person and machine (in which case we have been foolish in studying only one end of the channel), or that the software crisis is caused by a shortage of programmers (in which case we have been foolish in not studying why so many people can't write programs).

Of course, none of this is news in the field of psychology of programming, founded for these very reasons. But it has been disappointing, considering that psychology of programming research has been conducted for some years, that commercial developers of programming languages have not made more use of our findings to date. There is some good news, however. Microsoft is the largest supplier of programming tools today, and there is a team within the company devoted to evaluating and improving the usability of the Visual Studio programming products. Members of the team have attended Empirical Studies of Programmers and Psychology of Programming meetings, and have applied research results, as well as contributing to the research literature.

This paper arises from a visit to the Microsoft campus in September 2000, hosted by Steven Clarke and involving psychology of programming researchers Alan Blackwell, Chris Roast and Thomas Green. The main purpose of the visit was to discuss the use of Cognitive Dimensions (Green and Petre, 1996) as an evaluation tool – Steven had just completed a study (Clarke, 2001) of the new C# language using the CDs questionnaire (Blackwell and Green, 2000). CDs was not originally designed as an evaluation tool, but a new kind of design tool that would guide designers' decisions rather than reacting to them. Of course this implies an organisation where design is led by usability analysts, rather than a "throw it over the wall" model of usability evaluation where new products are specified solely from market research data, implemented, then subjected to usability labs in order to find and fix usability bugs.

The Visual Studio usability team, in response to our perspective on using CDs earlier in the design cycle, offered a challenge relating to their next project. Much of the day to day effort for Visual Studio users arises not from the usability problems of the user interface, or even the cognitive

challenges of complex language syntax and execution models, but in understanding and applying the Microsoft Foundation Class (MFC) libraries – the huge set of interfaces to Windows operating system functions. The team were about to start an evaluation project for the next release of MFC. Our group of visitors recognised the importance of the problem, spent a while thinking about previous research, and ultimately were unable to offer much help, either from CDs or from other research results. Hence this paper, passing on the challenge to the wider research community.

Class library usability versus software reuse

One area of research that is clearly relevant to the question is that of software reuse (Mili, Mili, and Mili, 1995). It is very well recognised, in the software reuse research community, that the main impediments to reuse are human factors. However the most important human factors in that context are a) that writing reusable code takes far more effort than writing it for a single project (and programmers are seldom motivated to invest a lot of effort that will benefit someone else's project) and b) that it is often easier to write a function again than to develop understanding and trust of code written by a colleague.

Neither of these factors apply in the case of MFC (or similar libraries such as the Java API). System libraries are coded by professional specialists, rather than as a by-product of other projects, and they are used by programmers who have no choice in using them – the MFC libraries are the only practical way to access operating system functions under Windows, for example. This means that the usability issues related to these libraries are focused on more specific cognitive questions rather than managerial and economic ones. As a result they are more tractable to traditional HCI research methods, and should be a promising opportunity for research; in the following sections we offer some initial points of reference for that research.

Evaluation methods

The usability of a class library will depend on three things, all of which should be subject to evaluation: its structure and representation (such as class and method names), the quality of its documentation, and the development environment in which it is used. Some class libraries are tied to a particular development environment (such as MFC, tied to Visual Studio), but others are not (such as the Java API).

Rasmussen, Pejtersen, and Goodstein (1994, Chapter 8) distinguish between analytical and empirical evaluation, and recommend using a combination of methods, at different stages in the design process. McGrath (1995) also advocates the use of multiple research strategies, noting that "credible empirical knowledge requires consistency or convergence of evidence across studies based on different methods", and discussing the trade-offs that must be made between generalisability, control, and realism when selecting a strategy. In this section we consider how different evaluation methods might be applied to class libraries. .

Analytical:

Creators of class libraries have already developed a number of guidelines and rules of thumb for their design (and redesign, or refactoring), via cumulative experience and feedback from programmers, and these should provide a good starting point for heuristic evaluation. Korson and McGregor (1992), for example, list ten desirable attributes of a class library (complete, consistent, easy-to-learn, easy-to-use, efficient, extendable, integrable, intuitive, robust, supported), and 23 criteria on which these attributes can be tested.

We have already noted how the Cognitive Dimensions framework has been used to evaluate a new programming language (Clarke, 2001), and it could also be applied to class libraries in particular. For example, the naming of classes and methods in a library and could be tested for closeness of mapping, consistency, diffuseness, error-proneness, and role-expressiveness. Level of abstraction is another CD of obvious relevance to class libraries in general.

In addition, there are many metrics of software quality, e.g. (Chidamber and Kemerer, 1994), which are easy to calculate and do seem to have some relationship to the number of faults present (Basili, Briand, and Melo, 1996) but it is not yet clear how they might relate to the usability of the components from a programmer's viewpoint.

Empirical:

Laboratory-based user experiments are perhaps the most popular type of empirical study, and could be carried out at any stage in the development of a class library. The **efficiency** of participants' interactions can be measured by considering the time taken to perform a set task, and the **effectiveness** of their solutions to the task may also be assessed, in terms of quality or accuracy. Experiment participants are also usually asked to fill in questionnaires to indicate their **satisfaction** with various aspects of the system, or their preference for different versions. In this way, different systems (or different versions of the same system) can be rigorously compared.

Frøkjær, Hertzum, and Hornbæk (2000) found that effectiveness, efficiency, and satisfaction are not necessarily related to each other, meaning that all three should be considered in any experiment. They also noted that efficiency is a far better indicator of usability for routine tasks than complex tasks, where a fast completion time may simply mean that the solution is ineffective. When evaluating a class library, the task set in an experiment is likely to involve programming, which is complex, and therefore it is crucial that the findings are not based on solution time alone.

In contrast to experiments, field studies are carried out in a natural setting, such as within an organisation; this may take the form of simple observation, as in ethnography, or something more obtrusive like manipulating an element of the system in use and recording what effect it has. The conventional alpha and beta testing process would probably be classified as a field study, but it occurs at a stage when the software is almost ready for release, and the developers are usually more concerned with finding bugs than with overall design issues. In the case of a development environment, it would be interesting to be able to deploy slightly different versions at this stage, instrumented to create log files, which would allow the versions to be compared. Such logging would also facilitate the compilation of statistics about the library, such as which components are used most often (Prieto-Díaz, 1991).

Finally, empirical methods also include surveys and questionnaires; ideally these would be carried out on a large scale, with a sample of people carefully selected to be representative of the target group of programmers.

Class libraries and information retrieval

In a usable library, it should be easy for the programmer to locate a class or method to suit a particular need. The software reuse community, assuming that programmers would soon have huge repositories of reusable components at their disposal, have explored this issue by adapting techniques from information retrieval research, an area which we examine in some depth in this section.

A usable class library should also be easy to understand, so that the programmer can grasp what a component actually does, decide whether it meets her needs, and incorporate it into her applications. In terms of information foraging theory (Pirolli and Card, 1999), this is known as *sensemaking*. Existing research into program comprehension tends to concentrate on source code, but with class libraries it is usually only the method signatures and documentation that are available to be read. Gibbs and his colleagues (1990) have noted the importance of the representation, or "packaging" of classes; typically it is necessary to rely on appropriate naming and good documentation. Fischer, Henninger, and Redmiles (1991) describe a system with complementary facilities for locating components and explaining them to the user (by way of example code fragments).

Querying and browsing

Information retrieval systems are used to automatically index collections of text documents: the individual words are extracted from each document, and a list of every unique term in the collection is created, where each term has pointers to all of the documents that contain it. The user issues a query by entering a few terms that characterise her requirement, and the system matches these against its list, returning a set of documents containing the user's terms. The system may weight the terms according to their frequency (within the individual documents or the collection as a whole), and this allows the results of the query to be ranked in order of their estimated relevance. The most widely visible application of these systems is in World Wide Web search engines, which index the full text of billions of documents.

With software components, however, the terms present in source code are not as content-descriptive as those in a typical text document, and it is difficult to automatically extract anything meaningful from them. This makes it necessary for the information retrieval system to rely on associated text, such as the library's documentation, enabling the user to search for components by issuing queries (Maarek, Berry, and Kaiser, 1991).

Another form of searching is directed browsing (Marchionini, 1995), where the user attempts to satisfy her requirement by simply looking through the library in whichever form it is presented to her. To support directed browsing, a class library should be well structured and named. With this strategy the user can recognise a relevant component when she sees it, rather than having to explicitly describe its characteristics. It may be difficult to create a good enough representation of a component (or a related group of components) in a small amount of screen space.

Mili and his colleagues (1999) make the distinction between querying and browsing in collections of software components, stating that "there is ample evidence to the effect that browsing is the most predominant pattern of library usage, if only because software reuse is consistent with bottom-up software design". The user may prefer directed browsing to querying if her requirement is vague, or difficult to express in words, especially if she is already somewhat familiar with the classes. In the latter case, the development environment can prompt interactively to help her remember method names, parameters, and so on, meaning that she does not have to rely on her own memory, or break her current train of thought to look up the class in a manual.

Annotation and organisation of components

The author of a class library's documentation will concentrate on clarity of exposition, not on whether she is using the right terms to facilitate automatic indexing and retrieval. Both querying and browsing can be assisted by the provision of more specialised cataloguing or indexing, via manual annotation of the components. Classes in Eiffel (Meyer, 1990), for example, may contain a special "indexing" clause, which allows the programmer to write annotations directly in the code. Manual annotations may take many forms, such as facet analysis (Prieto-Díaz, 1991), keywords from a controlled vocabulary, or attribute-value pairs. Creating high quality annotations is likely to be time-consuming and difficult, however, and programmers may need to be offered suitable incentives to invest sufficient effort in it.

In addition, the annotations will always reflect the subjective judgement of the annotator, who cannot anticipate all of the potential requirements for which a component may be useful. Furnas and his colleagues (1987) identified what they called "the vocabulary problem" in the context of command naming: they found that, in five different application domains, it was unlikely that two people would spontaneously use the same term to describe a given concept. For example, a programmer may not be aware of specialised terminology used to describe a particular type of data structure or algorithm.

Class libraries are often given some form of hierarchical structure (like the packages in the Java API), grouping related items together as in the classification of non-fiction books according to subject matter. Again, this is usually highly subjective (Atkinson, 1997). Information foraging theory suggests that such structuring can support directed browsing, as long as the chosen representatives at each level of the hierarchy offer a good *information scent*; this is defined as "the (imperfect)

perception of the value, cost, or access path of information sources obtained from proximal cues, such as bibliographic citations, WWW links, or icons representing the sources" (Pirolli and Card, 1999).

Frakes and Pole (1994) carried out an experiment that compared automatic full-text indexing of Unix command documentation to three manual forms of cataloguing (hierarchical structuring, facet analysis, and attribute-value). Participants were asked to locate a Unix command, given a description of its function. There were no significant differences between the four indexing methods in terms of effectiveness or satisfaction, but the participants were significantly more efficient with the system based on hierarchical structuring than any of the other three. The library was very small, however, with only 120 items.

Inferring user needs

A query is an explicit description of the user's current requirement, but when she is browsing through a class library, it may be possible to regard her selections as implicit indications of the sort of component she is looking for. This is the approach taken by Drummond, Ionescu, and Holte (2000), who describe a system that makes inferences based on this information, suggesting components to the user based on the similarity of their structure and naming to those that she has already shown some interest in.

So far we have assumed that the programmer will realise that she needs to go and look through the library in order to find a component to perform a particular function. It may not occur to her, however, that an appropriate component already exists, and she may write new code without knowing that she could saved herself some effort by reusing code from the library. The system described by Ye and Fischer (2002) processes whatever the user is currently writing in the program editor, and again compares structure and naming (using method signatures) to suggest suitable methods from the library. It also processes any text that the programmer places inside Java's special documentation comments, and uses that to make suggestions based on conventional text retrieval.

Collaborative filtering techniques can also be employed in this area (Chalmers, 2000): if a library is used by many programmers, it is likely that at some point in the past, someone else will have followed a similar browsing path to the current user, when searching for code to carry out the same function. Again, this can be used to make suggestions, based on the previous paths. One advantage of this method is that it can be used regardless of whether the components have been annotated.

Evaluation

Information retrieval researchers have also recognised the need to evaluate different aspects of a system, using different methods (Dunlop, 2000). Evaluations tend to centre around the concept of relevance (Schamber, 1994), which is usually taken to mean relatedness to a particular topic, although a document's real utility will also depend on many other factors, including its quality and its novelty. For software components we could add many more, such as performance and adaptability. Evaluation measures like recall and precision are based on counting how many of the items retrieved in response to a query are actually relevant to it, because the user may gather useful information from a number of documents. When searching in a class library, however, the user is normally only looking for a single component, making these evaluation measures fairly meaningless for a single search.

Conclusions

The design of class libraries brings serious challenges for research into human issues in programming. Much of the relevant research has been conducted under the sponsorship of major research initiatives into software reuse (the largest of which was funded by the US Department of Defense). This paper, in contrast, has taken its lead from a direct challenge made by the usability group at Microsoft, where the Microsoft Foundation Classes are central to the usability demands of the Visual Studio products. We believe that such initiatives, although rare, should be strongly welcomed as bringing priorities and guidance to academic research. In this paper we have set out some of the ground established in

previous research, especially in information retrieval, but leave far more potential for future research, in a broad challenge to this community.

Acknowledgements

This research has been funded by the Engineering and Physical Sciences Research Council under EPSRC grant GR/M16924 “New paradigms for visual interaction”.

References

Atkinson, S. (1997) Cognitive Deficiencies in Software Library Design. In *Proceedings of the 4th Asia-Pacific Software Engineering and International Computer Science Conference (APSEC ‘97 / ICSC ‘97)*.

Basili, V.R., Briand, L.C., and Melo, W.L. (1996) A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering*, **22**(10): 751-761.

Blackwell, A.F., and Green, T.R.G. (2000) A Cognitive Dimensions Questionnaire Optimised for Users. In *Proceedings of the 12th Workshop of the Psychology of Programming Interest Group*.

Chalmers, M. (2000) When Cookies Aren’t Enough: Tracking and Enriching Web Activity with Recer. In R. Rogers, editor, *Preferred Placement: Knowledge Politics on the Web*. Maastricht: Jan van Eyck Akademie Editions, pages 99-102.

Chidamber, S.R., and Kemerer, C.F. (1994) A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, **20**(6): 476-493.

Clarke, S. (2001) Evaluating a new programming language. In *Proceedings of the 13th Workshop of the Psychology of Programming Interest Group*.

Drummond, C.G., Ionescu, D., and Holte, R.C. (2000) A Learning Agent that Assists the Browsing of Software Libraries. *IEEE Transactions on Software Engineering*, **26**(12): 1179-1196.

Dunlop, M (2000) Reflections on Mira: interactive evaluation in information retrieval. *Journal of the American Society for Information Science*, **51**(14):1269-1274.

Fischer, G., Henninger, S., and Redmiles, D. (1991) Intertwining Query Construction and Relevance Evaluation. In *Proceedings of ACM CHI’91*.

Frakes, W.B., and Pole, T.P. (1994) An Empirical Study of Representation Methods for Reusable Software Components. *IEEE Transactions on Software Engineering*, **20**(8): 617-630.

Frøkjær, E., Hertzum, M., and Hornbæk, K. (2000) Measuring usability: Are effectiveness, efficiency, and satisfaction really correlated? In *Proceedings of ACM CHI 2000*.

Furnas, G. W., Landauer, T. K., Gomez, L. M., and Dumais, S. T. (1987) The vocabulary problem in human–system communication. *Communications of the ACM*, **30**(11):964-971.

Gibbs, S., Tsichritzis, D., Casais, E., Nierstrasz, O., and Pintado, X. (1990) Class Management for Software Communities. *Communications of the ACM*, **33**(9): 90-103.

Green, T. R. G., and Petre, M. (1996) Usability analysis of visual programming environments: a ‘cognitive dimensions’ framework. *Journal of Visual Languages and Computing*, **7**(2):131-174.

Korson, T., and McGregor, J.D. (1992) Technical criteria for the specification and evaluation of object-oriented libraries. *Software Engineering Journal*, **7**(2): 85-94.

Maarek, Y., and Berry, D., and Kaiser, G. (1991) An Information Retrieval Approach For Automatically Constructing Software Libraries. *IEEE Transactions on Software Engineering*, **17**(8): 800-813.

Marchionini, G. (1995) *Information Seeking in Electronic Environments*. Cambridge: Cambridge University Press.

McGrath, J. E. (1995) Methodology matters: Doing research in the behavioural and social sciences. In R. M. Baecker, J. Grudin, W. A. S. Buxton, and S. Greenberg, editors, *Readings in Human–Computer Interaction: Toward the Year 2000*, pages 152-169. San Francisco: Morgan Kaufmann.

Meyer, B. (1990) Lessons from the design of the Eiffel libraries. *Communications of the ACM*, **33**(9):68-88.

Mili, A., Yacoub, S., Addy, E., and Mili, H. (1999) Toward an Engineering Discipline of Software Reuse. *IEEE Software*, **16**(5): 22-31.

Mili, H., and Mili, F., and Mili, A. (1995) Reusing Software: Issues and Research Directions. *IEEE Transactions on Software Engineering*, **21**(6): 528-561.

Pirolli, P., and Card, S. K. (1999) Information Foraging. *Psychological Review* **106**(4): 643-675.

Prieto-Díaz, R. (1991) Implementing Faceted Classification for Software Reuse. *Communications of the ACM*, **34**(5): 89-97.

Rasmussen, J., and Pejtersen, A.M., and Goodstein, L.P. (1994) *Cognitive Systems Engineering*. New York: Wiley.

Schamber. L. (1994) Relevance and information behavior. *Annual Review of Information Science and Technology*, **29**:3-48.

Ye, Y., and Fischer, G. (2002) Information Delivery in Support of Learning Reusable Software Components on Demand. In *Proceedings of ACM Intelligent User Interfaces 2002*.