An Approach for Categorizing End User Programmers to Guide Software Engineering Research

Christopher Scaffidi
Institute for Software Research Intl.
School of Computer Science
Carnegie Mellon University
+1-412-268-3564
www.cs.cmu.edu/~cscaffid
cscaffid+isri@cs.cmu.edu

Mary Shaw
Sloan Software Industry Center &
School of Computer Science
Carnegie Mellon University
+1-412-268-2589
www.cs.cmu.edu/~shaw
mary.shaw@cs.cmu.edu

Brad Myers
Human-Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
+412-268-5150
www-2.cs.cmu.edu/~bam/
bam@cs.cmu.edu

ABSTRACT
Over 64 million Americans used computers at work in 1997, and we estimate this number will grow to 90 million in 2012, including over 55 million spreadsheet and database users and 13 million self-reported programmers. Existing characterizations of this end user population based on software usage provide minimal guidance on how to help end user programmers practice better software engineering. We describe an enhanced method of characterizing the end user population, based on categorizing end users according to the ways they represent abstractions. Since the use of abstraction can facilitate or impede achieving key software engineering goals (such as improving reusability and maintainability), this categorization promises an improved ability to highlight niches of end users with special software engineering capabilities or struggles. We have incorporated this approach into an in-progress survey of end user programming practices.

Categories and Subject Descriptors
D.2.6 [**Software Engineering**]: Programming Environments – *interactive environments, graphical environments, integrated environment;* D.2.11 [**Software Engineering**]: Software Architectures – *data abstraction;* K.8.1 [**Personal Computing**]: Application Packages – *database processing, spreadsheets*.

General Terms
Design

Keywords
end user software engineering, end user programming, abstraction

1. INTRODUCTION
As reported in 1995 [6], and widely disseminated in 2000 [7], Boehm et. al. estimated that end user programmers would number 55 million in 2005, compared to fewer than 3 million professional programmers.

First Workshop on End-User Software Engineering (WEUSE I). May 21 2005, Saint Louis, Missouri, USA.

We examined the context and method that generated this "55 million" estimate and discovered that it actually constitutes an estimate of Americans using computers at work—rather than end user programmers, per se [25]. Here, we seek to distinguish end user programmers from non-programmers in a way that goes beyond just a single number and helps guide the design of tools to support end user software engineering.

Specifically, a simple binary division of "end user programmers" from "end user non-programmers" provides inadequate insight into end user behavior to guide future research and tool development. Instead, we argue that end users exhibit a variety of practices ranging from programming-like to non-programming-like, and we believe that we can fruitfully characterize this distribution on the basis of how end users represent *abstractions*. (While we argue for this approach on the basis of its relevance to software engineering research, Blackwell has made similar arguments from the standpoint of studying the cognitive aspects of programming [2].)

In Section 2, we describe how previous research has attempted to categorize end users based on software usage, and we highlight this method's inadequacies. In Section 3, we detail a categorization of end users based on how they represent abstractions, and we describe an in-progress survey that incorporates this abstraction-oriented approach.

2. PROBLEM BACKGROUND
2.1 The End User Population's Size
Boehm estimated that end users in American workplaces would number 55 million in 2005 [6], but in fact the end user population already exceeded 64 million in 1997 and continues to grow [25]. This realization prompted us in a previous report [25] to extend Boehm's "55 million" estimate with fresh data and a richer model accounting for rising computer usage rates among workers. Using survey results and projections from the Bureau of Labor Statistics (BLS), we estimated that over 90 million Americans will use a computer at work in 2012 (the year for which BLS published occupational projections), including over 55 million spreadsheet and database users and 13 million self-reported programmers, compared to fewer than 3 million professional programmers [25]. Thus, the potential pool of end user programmers will significantly exceed the population of professional programmers for the foreseeable future.

2.2 End Users' Diverse Software Usage

How many end users actually program? BLS software usage data from 2001 [25] offer a coarse-grained answer shown in Table 1.

End users exhibit a diversity of software usage practices. Although only 15% of end users reported that they "do programming" at work in 2001, over 60% of end users used spreadsheets or databases at work. Further, the BLS data do not explicitly address many other end user programming environments, such as "educational simulation builders, web authoring systems, multimedia authoring systems, e-mail filtering rules, CAD systems" [24] and other scripting environments.

Moreover, within the context of a given software tool such as spreadsheet editors, end users exhibit a range of usage patterns. For example, Fisher and Rothermel's survey of 4498 spreadsheets on the web found that only 44% contained formulas [12]. Hall's study of 106 spreadsheets created by well-educated Australian workers revealed that 47% used "if" functions, while only "21% involved links with a database" [15]. To date, researchers appear to have studied spreadsheet usage more than any other end user programming environment; however, we anticipate that studying end users' practices in other environments (such as web page authoring) would also reveal a comparable variety of activities ranging from programming-like to non-programming-like.

In short, the burgeoning end user population demonstrates a diversity of software usage practices, and software usage data like these constitute a coarse-grained characterization of the extent to which end users engage in programming.

Table 1. Software application usage by US workers in 2001

Question: Do You…	Thousands of Users	Percent of Computer Users
Use a computer at your main job?	**72,277**	**<<100%>>**
• Connect to the Internet or use email?	51,895	71.8%
• Do word processing or desktop publishing?	48,426	67.0
• Use spreadsheets or databases?	45,029	62.3
• Use a calendar or do scheduling on the computer?	38,235	52.9
• Do graphics and design?	20,816	28.8
• Do programming?	10,986	15.2

2.3 Past Software-Focused Categorizations

Unfortunately, a coarse-grained categorization based on software usage is inadequate for guiding programming tool designers: it tells *what* tools people use but not *why*, nor *how* to improve tools. First, as discussed below, it glosses over niches of end users with special needs or capabilities. Second, it fails to highlight concerns spanning multiple types of programming environments.

These limitations are apparent, for example, in Nardi's taxonomy of programming environments [20]:

- Textual languages, including spreadsheet formulas
- Programming by example (PBE) systems, exemplified by the Eager extensions to HyperCard
- Automatic programming systems, such as WorldBuilder
- Form-input tools, like FrameMaker's style designer, where users fill in a form to specify the style
- Visual programming languages, such as LabView

First, purely tool-based categorizations like Nardi's mask potentially interesting sub-populations. For instance, both FileMaker and FrontPage mainly rely on visual design (and fall into the last bullet above) to support creation of forms by end users. However, FileMaker (unlike FrontPage) allows end users to define a data structure and associate multiple forms with that data structure. Thus, FileMaker gives extra capabilities to users (who may say, "I chose to delete a data field, and FileMaker conveniently removed it from all forms"); on the other hand, FileMaker also presents extra challenges ("I accidentally deleted a data field, and FileMaker removed it from *all* my forms!") So FileMaker users may constitute a niche with special capabilities and challenges. Coarse tool-based categorizations like Nardi's fail to reveal much of the variation in power among tools within the same category.

Second, purely tool-focused categorization fails to emphasize issues affecting many tools. For instance, though research has documented the prevalence of spreadsheet bugs [21], it is not clear whether the same types of end user bugs abound in textual, PBE, automatic, visual, and form-based environments. This lack of research may hinder generalizing reliability research from one end user programming venue (like spreadsheets) to others.

Indeed, most software engineering concerns, including reliability, apply to many programming environments. These goals include fostering constructability, safety, maintainability, efficiency, cost-effectiveness, dependability, security, and ubiquity. Learning how these cross-cutting issues impact multiple categories of end user programmers may guide research that will benefit more than one category of end user programmer at a time.

3. PROPOSED APPROACH

We believe that studying how end users represent *abstractions* will uncover interesting niches of end user programmers and will highlight key end user software engineering challenges and opportunities. In this section, we define "abstraction" and discuss three common ways to represent abstractions (variables, functions, and data structures). After discussing the interplay between abstraction and key software engineering concerns (such as reliability and maintainability), we explain how our focus on abstraction guided our survey of end user programming practices.

3.1 Abstraction's Definition and Purpose

One formal view of "abstraction" comes from lambda calculus. "Abstracting a composed value *v* from some simple value *a* means 'stripping off' the *a* property from *v*, creating a generalized object—a function to be applied later. Technically, the result of such an abstraction is replacing each occurrence of *a* in *v* by a variable *x*, yielding a function of a single parameter *x*" [1]. For example, the expressions "(80.0/100)" and "(60.0/100)" are represented by the abstraction "(*x*/100)" by "stripping off" 80.0 and 60.0, respectively, and replacing them with variable *x*.

Applied to programming practice, "abstraction" acquires a pragmatic flavor. In this context, "a good abstraction is one that emphasizes details that are significant to the reader or user and suppresses details that are, at least for the moment, immaterial or diversionary" [26]. For example, consider writing an algorithm to

convert percentages to decimal fractions. The algorithm implementer mainly focuses on dividing by 100; in contrast, the algorithm users mainly focus on specifying percentages to convert. Representing the abstract algorithm as a function cleanly separates implementer concerns and user concerns.

Abstractions hold value (in part) because they facilitate focusing on the general aspects of a problem and reusing the solution on many instances of that problem. For example, an accountant might define a `calcInterest` function to calculate total interest on an arbitrary loan and then apply this function to specific loans. In short, representing an abstract generalization may facilitate reuse across problem instances.

3.2 Abstraction as a Focus of Past Research

Researchers have developed tools that facilitate representing abstractions as variables, functions, and data structures.

3.2.1 Variables

Variables constitute the simplest programming representation of abstraction. They separate value generation (when some variable V is set equal to some expression E) from value usage (when V's value is retrieved for use in some later computation C_0).

Unfortunately, a coder may skip defining V and embed E directly inside C_0 and other computations C_1, C_2, and C_N, resulting in less maintainable code. For example, if he uses a 6% interest rate to compute the total owed on a $200 loan and the interest accrued, he may skip defining temporary variables, instead coding:

```
print("Total=");      print(200*exp(1+6/100));
print(", Interest="); print(200*exp(1+6/100)-200);
```

Note that the failure to define temporary variables led to wasteful replication of expressions, resulting in less efficient, maintainable, and reusable code. Hence, researchers have provided tools that a professional programmer can use (after code is written as in the example above) to automatically extract expression E from each computation C_i, replacing each usage with a variable V initialized once from E and reused in each C_i [14]. Supporting abstraction with variables is particularly valuable, since variables participate in other representations, such as functions and data structures.

3.2.2 Functions

Functions represent algorithmic abstractions. They existed since the invention of macros and assemblers in the 1950's [26] (and, of course, in mathematics since Leibniz coined the term in 1694). More sophisticated types of functions now exist, including spreadsheet macros, JavaScript event handlers, and stored procedures. Functions encompass what Blackwell refers to as "abstraction over time" [3], where a user records behavior for playback; however, since functions accept parametric variables, they separate behavior concerns from data concerns, in addition to separating behavior concerns from time of execution.

Many research prototypes of end user programming environments provide a means for end users to represent algorithmic abstractions (see Table 2). Conversely, in the "real world," spreadsheet tools provide little or no support for defining and reusing functions, yet these tools constitute the most widely used type of end user programming environment [16].

Though researchers have extended spreadsheets to ease definition of formulas [16], the disparity between functional abstraction research and practice raises a number of questions: Do end users often define functions in tools that support functional abstraction? If not, is it because of tool deficiencies, learning barriers, or simply because representing new algorithmic abstractions holds little value for end users? We will return to such questions below.

3.2.3 Data structures

Finally, many abstractions involve composing pieces of data into a structured whole. Various end user programming environments support representation of such abstractions (see Table 2). Data structures offer a fairly simple concrete representation of what Blackwell terms "abstraction over a class of entities" [3], though in the software engineering literature, structured data research dovetailed into more advanced innovations: abstract data types, generic types, and inheritance (see [26] for a survey). It is unclear

Table 2. A sampling of end user programming environments and their support for representing new abstractions

Environment	Domain	Support for functions	Support for data structures
AutoHAN [3]	Home automation	Channel Cubes can map to scripts that call functions on appliances.	Aggregate Cubes can represent a collection of other Media Cubes.
BOOMS [1]	Music editing	Functions record series of music edits.	Structures contain notes and phrases.
Forms/3 [8]	Spreadsheet editing	Forms simultaneously represent a function and an activation record.	Types are structured collections of cells and graphical objects.
Gamut [17]	Game design	Behaviors are learned from positive and negative examples.	Decks of cards serve as graphical containers with properties.
Janus [11]	Floor plan design	Critic rules encode algorithms for deciding if a floor plan is "good."	Instances of classes may possess attributes and sub-objects.
KidSim [27]	Simulation design	Graphical rewrite rules describe agent behavior.	Agents may possess properties and are cloned for new instances.
Lapis [18]	Structured text editing	Scripts automate a series of edits.	Text patterns can contain sub-structure.
Pursuit [19]	File management	Scripts automate a series of manipulations.	Filter sets contain files and folders.
QUICK [9]	UI design	Actions may be associated with objects (that are then cloned).	Objects may have attributes and be cloned and/or aggregated.

whether end users utilize any of these more advanced representations of abstraction, nor how researchers might enhance existing tools to provide better support in this area.

3.3 Abstraction and Software Engineering

The use of abstraction can significantly affect key software qualities such as maintainability and reliability.

On one hand, abstraction can increase the quality of software. For example, high notation viscosity (the difficulty of making local changes) can damage software maintainability, but it is known that "viscosity can be reduced by increasing the number of abstractions" [13]. Of course, simply adding more abstraction does not automatically reduce viscosity. Abstractions must be selected prudently in order to encapsulate features that are likely to change in the future; this prevents local changes from cascading into other sections of the application. Though this maintainability-enhancing design principle first appeared in the software engineering literature over thirty years ago [23], it seems likely that it has not yet significantly impacted actual end user programming practice.

Likewise, researchers realized long ago that comprehensive testing requires modular code for several reasons. First, modular structures tend to exhibit much lower complexity, thereby reducing the number of tests required to achieve adequate confidence in code correctness. Second, if a system is built by combining smaller abstraction "building blocks," then each abstraction's module may ideally be tested independently of the others, further simplifying the testing task. Finally, the opportunity to reuse modules may save coding time, which the programmer may then invest in other activities, such as testing. For all these reasons, abstraction-centric, modularized code has the potential to exhibit high correctness and reliability [10] [22].

On the other hand, "increasing abstractions tends to create hidden dependencies" because "quite often abstractions themselves bring problems of visibility" [13]. In other words, abstractions can hinder changing the system without introducing bugs. Thus, used incorrectly, abstraction can degrade maintainability and reliability. Tool designers cannot simply provide support for representing abstractions and assume this will alone improve maintainability and reliability; instead, tools must also provide guidance to help programmers effectively create and comprehend abstractions. Achieving this requires understanding whether, when, and how end user programmers create and understand abstractions.

Abstraction can benefit or harm a wide variety of other software quality attributes, each of which involves many categories of programmer and programming environment. Thus, studying end user abstraction representation promises insight into the software engineering challenges and opportunities facing end users today.

3.4 Abstraction as the Focus of Our Survey

Based on these considerations, we tailored our data collection to emphasize abstraction representation by end users. We created a survey that first asks users about their software usage and then about usage of features related to the representation of abstraction.

Our survey was fielded in *Information Week* magazine beginning in February 2005 (using a questionnaire posted on their web site), and we will have results by May 2005. We will follow this with an updated survey on a targeted, scientific sample. Based on discussions with researchers, we identified the following popular end user programming tools in the business context:

- Spreadsheets
- Word processors and presentation tools
- Web page editing tools
- Web server scripting languages
- Databases
- Reporting tools / business intelligence

For each type of software, our survey asks about features that end users might utilize for representing abstractions. Different programming environments represent abstractions differently, and we have worded our questions accordingly. For example, to test for function-like representations of algorithmic abstractions, we ask spreadsheet users about recording macros, as well as creating or editing macros in the macro editor; for databases, we ask about creating stored procedures.

The survey also contains several questions related to programming practices. For example, we ask spreadsheet users whether they test their spreadsheets. We ask all respondents several questions about documentation habits, how they use the web during programming, and their knowledge of programming terminology. We also ask about background information for use as independent variables.

We expect to launch a web-based survey of 2500 marketing professionals in 2005. We selected this population because preliminary discussions with marketing professionals suggested that they perform a wide variety of programming activities, ranging from manipulating numerical data to publishing web pages. In a sense, marketing professionals may represent an "upper-bound" on the amount and diversity of end user programming in the workplace.

3.5 Building on the Survey Results

Our surveys will likely show that each abstraction representation is used less often in some programming environments than in others. For example, we may discover that end users frequently represent functional abstractions in web pages (using JavaScript functions) but only rarely in spreadsheet environments (through macros) and databases (through stored procedures). Relatively low usage rates raise an important question: Do users rarely utilize a given abstraction representation in certain environments because it is not useful in those contexts, or would they like to use the representation but fail due to inadequate tool support? Although our surveys will not answer this question directly, they will highlight areas where the question applies.

A related question concerns how well end users understand abstraction and the extent to which they want to represent new abstractions. This issue influences what type of assistance the environment must provide. For example, using the terminology of Bloom's taxonomy in the cognitive domain [5], suppose an end user currently manipulates abstractions at the *Knowledge* level of understanding (perhaps he has memorized that a script needs to be wrapped with Tcl keywords, as in Lapis [18]). In such a case, it would not be reasonable for the tool to require *Synthesis* of multiple scripts in order to achieve useful work, since synthesis involves a much higher level of understanding within Bloom's

taxonomy. This mismatch between user and system requirements exemplifies the pitfall of excessive "abstraction-hunger" [13].

Answering these questions will require interviews and observational studies of end users at work. Combining the results from these future studies with our survey data will provide guidance for how to improve tools to better support end users' programming goals.

4. ACKNOWLEDGEMENTS

We thank Andrew Ko for comments on drafts. This work has been funded in part by the EUSES Consortium via the National Science Foundation (ITR-0325273), by the National Science Foundation under Grant CCF-0438929, by the Sloan Software Industry Center at Carnegie Mellon, and by the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

5. REFERENCES

[1] Balaban, M., Barzilay, E., and Elhadad, M. Abstraction as a Means for End User Computing in Creative Applications. *IEEE Transactions on Systems, Man and Cybernetics, Part A, 32*, 6 (Nov. 2002), 640-653.

[2] Blackwell, A. First Steps in Programming: A Rationale for Attention Investment Models. In *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, 2002, 2-10.

[3] Blackwell, A., and Hague, R. AutoHAN: An Architecture for Programming the Home. In *Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments*, 2001, 150-157.

[5] Bloom, B., Mesia, B., and Krathwohl, D. *Taxonomy of Educational Objectives*. David McKay Publishers, New York, NY, 1964.

[6] Boehm, B., et al. Cost Models for Future Software Life Cycle Processes: COCOMO 2.0. *Annals of Software Engineering Special Volume on Software Process and Product Measurement*, J.C. Baltzer AG Science Publishers, Amsterdam, The Netherlands, 1995.

[7] Boehm, B., et al. *Software Cost Estimation with COCOMO II*. Prentice-Hall, 2000.

[8] Burnett, M., et al. Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm. *Journal of Functional Programming, 11*, 2 (Mar. 2001), 155-206.

[9] Douglas, S., Doerry, E., and Novick, D. Quick: A User-Interface Design Kit for Non-Programmers. In *Proceedings of the 3rd Annual ACM SIGGRAPH Symposium on User Interface Software and Technology*, 1990, 47-56.

[10] Edwards, N. The Effect of Certain Modular Design Principles on Testability. In *Proceedings of the International Conference on Reliable Software*, 1975, 401-410.

[11] Fischer, G., and Girgensohn, A. End User Modifiability in Design Environments. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1990, 183-192.

[12] Fisher II, M., and Rothermel, G. The EUSES Spreadsheet Corpus: A Shared Resource for Supporting Experimentation with Spreadsheet Dependability Mechanisms. Technical Report 04-12-03, University of Nebraska--Lincoln, Lincoln, NE, Dec. 2004.

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

[14] Griswold, W., and Notkin, D. Automated Assistance for Program Restructuring. *ACM Transactions on Software Engineering Methodology, 2*, 3 (July 1993), 228-269.

[15] Hall, J.. A Risk and Control-Oriented Study of the Practices of Spreadsheet Application Developers. In *Proceedings of the 29th Hawaii International Conference on System Sciences*, 1996, 364-373.

[16] Jones, S., Blackwell, A., and Burnett, M. A User-Centred Approach to Functions in Excel. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*, 2003, 165-176.

[17] McDaniel, R., and Myers, B. Getting More Out of Programming-By-Demonstration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1999, 442-449.

[18] Miller, R., and Myers, B. LAPIS: Smart Editing with Text Structure. In *CHI '02 Extended Abstracts on Human Factors in Computing Systems*, 2002, 496-497.

[19] Modugno, F., and Myers, B. Pursuit: Graphically Representing Programs in a Demonstrational Visual Shell. In *Proceedings of the CHI '94 Conference Companion on Human Factors in Computing Systems*, 1994, 455-456.

[20] Nardi, B. *A Small Matter of Programming*, MIT Press, Cambridge, MA, 1993.

[21] Panko, R. What we know about spreadsheet errors. *Journal of End User Computing, 10*, 2 (Spring 1998), 15-21.

[22] Parnas, D. The Influence of Software Structure on Reliability. In *Proceedings of the International Conference on Reliable Software*, 1975, 358-362.

[23] Parnas, D. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM, 15, 12* (Dec. 1972), 1053-1058.

[24] Ruthruff, J., et al. Debugging and Finding Faults: End User Software Visualizations for Fault Localization. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, 2003, 123-132.

[25] Scaffidi, C., Shaw, M., and Myers, B. The "55M End User Programmers" Estimate Revisited. Technical Report CMU-ISRI-05-100, Carnegie Mellon University, Pittsburgh, PA, 2005.

[26] Shaw, M. Abstraction Techniques in Modern Programming Languages. *IEEE Software, 1*, 4 (Oct.1984), 10-26.

[27] Smith, D., Cypher, A., and Spohrer, J. KidSim: Programming Agents without a Programming Language. *Communications of ACM, 37*, 7 (July 1994), 54-67.