Reuse in the world of end-user programmers

Christopher Scaffidi, Mary Shaw
{ cscaffid, mary.shaw } @ cs.cmu.edu
Carnegie Mellon University

Half a century ago, deliverymen brought milk to the doorsteps of customers, who would rinse and return empty glass bottles later that week. Ultimately, the bottles would be sterilized and reused to deliver more milk. But reuse also happened outside the milk company. Children could glue beads and popsicle sticks onto a bottle for an arts-and-crafts project. Adults could drill a hole the size of a wire near the bottle's bottom in order to turn the bottle into a lamp base.

Glass milk bottles seem quaint now, but they illustrate the amazing range of human ingenuity when it comes to reuse. And reuse has never completely gone out of style: it simply takes different forms in different contexts and periods of history. Like sanitized milk bottles, many products ranging from Kitchen Aid mixers to Apple iPods are often refurbished and resold. Like raw materials for an arts-and-crafts project, movies and music are copied and mashed up into YouTube videos. And as when creating a base for a lamp, text is extracted from research papers, extended, and published in a longer form.[1]

Likewise, reuse is a central aspect of end-user programming. End users not only reuse code created by professional programmers, but they also reuse code created by their peers. This code includes spreadsheets, JavaScript and web macros. For example, empirical studies have shown that CoScripter users often reuse one another's web macros, which have proven to be valuable for capturing, communicating, and automating the steps required to manipulate web sites [5][14]. Users sometimes execute existing macros without modification, but other times they modify another person's macro before executing it. They also sometimes learn from one another's macros before writing a new macro of their own. In addition, they sometimes combine existing macros via copy-and-paste into a new macro.

Such reuse differs quite a bit from the idealistic vision of assembling fresh, pristine applications from sleek, finely-tuned components that call one another through tidy interfaces. Granted, to a certain extent, this vision has become a reality, as professional programmers now routinely create systems from database components, Java virtual machines, or SMTP servers. End-user programmers also reuse professionally-created components, such as the Excel spreadsheet environment, the Microsoft JavaScript interpreter, and the IBM CoScripter platform.

But reuse of code created by other end users is messy, since such code typically lacks clean interfaces, so reusing such code typically involves actually digging into its details and understanding it. Moreover, end users usually have not been trained in designing for reuse, and end users have no little or no time to design for reuse. Indeed, they may not even think to plan for reuse in the first place. Instead, their focus is getting their job done and moving on with life. This creates a tension: end users tend not to produce particularly reusable code, and other end users typically lack the time to invest in studying many different pieces of code in hopes of finding something reusable. Not surprisingly, a great deal of end-user code goes un-reused.

This tension puts a premium on expediency: end users need effective approaches for quickly identifying reusable code created by other users, understanding that code, and adapting the code or learning from it in order to create new code. A wide variety of empirical studies have shed light on how it might be possible to help users with the first of these activities, identifying reusable end-user code. These empirical studies show that the reusability of code can be inferred on the basis of "low-ceremony" evidence. This evidence is

[1] Portions of this chapter previously were presented at the End User Programming for the Web Workshop, in conjunction with the Conference on Human Factors in Computing Systems (CHI 2009), April 2009.

information that is often informal, possibly unreliable, but that can be quickly gathered, interpreted and synthesized without the investment of substantial effort or skill by code producers or consumers.

We begin this chapter by examining the empirical considerations that motivate reuse of end-user code in the first place. We discuss different ways in which end users reuse one another's code, and we contrast this with the ways in which they reuse code written by professional programmers. We follow this by describing how the reusability of end-user programmers' code can be inferred from low-ceremony evidence. Finally, we close by considering future research directions aimed at helping end users to identify, understand, and adapt reusable code.

Reuse: vision versus reality

When some software engineering researchers hear the word "reuse", they think of creating a program by assembling together meticulously designed components. McIlroy may have been the first (and most eloquent) person to espouse this vision [26]:

> *The most important characteristic of a software components industry is that it will offer families of routines for any given job… In other words, the purchaser of a component from a family will choose one tailored to his exact needs… He will expect families of routines to be constructed on rational principles so that families fit together as building blocks. In short, he should be able safely to regard components as black boxes.*

Such a vision might seem ideally suited to the needs of end-user programmers. What is not to like in the idea of letting end users assemble programs from black boxes? Let us deconstruct this vision and examine the reasons why its implicit assumptions are unsatisfied in the context of end-user programming.

Assumption: The "industry" can understand end users' work well enough to provide "routines" meeting the "exact needs" for "any given job".

Analyses of federal data show that end users have a remarkable range of different occupations [23]. They include managers in accounting companies and marketing firms, engineers in many different fields, teachers, scientists, health care workers, insurance adjusters, salesmen, and administrative assistants. In smaller numbers, they also include workers in many extremely specialized industries such as data processing equipment repairers, tool and die makers, water and sewage treatment plant operators, early childhood teacher's assistants, and printing press operators.

Researchers have documented a wide range of different ways that workers in these diverse industries write programs to help automate parts of their work. Examples include office workers creating web macros to automate interactions with intranet sites [14], advertising consultants creating web macros to compile sales information [13], managers creating spreadsheets to manage inventory [8], system administrators creating scripts to analyze server logs [1], and teachers creating programs to compute students' grades [29]. Within each industry and each occupation and each kind of programming is still more variation. For example, different office workers use different web sites, and even those within the same organization use their intranet sites in different ways.

Not only is the diversity of end users' contexts is simply staggering, but the sheer numbers of end users is overwhelming. Overall, by 2012, there are likely to be 90 million computer users in American workplaces alone, including 13 to 25 million end-user programmers (depending on the definition of "programming") [23]. In contrast, the federal government anticipates that professional programmers in America will only number 3 million, nearly an order of magnitude smaller than the number of end-user programmers.

Given the huge difference between the sizes of these two populations, as well as the incredible diversity of end users' jobs, it is inconceivable that the industry of professional programmers could understand every kind of end user's work at a sufficient level of detail to provide routines that meet the "exact needs" of "any

given [programming] job". McIlroy's hypothetical catalog cannot be complete if it is to meet the "exact needs" of end-user programmers. There simply is a huge range of domain-specific detail that only the end users themselves understand.

Thus, must necessarily be a significant functional gap between the generic components provided by an "industry" of professional programmers and the domain-specific requirements of end users. For example, IBM provides the general-purpose CoScripter platform, but the end users are the ones who implement web macros that manipulate specific web sites in a way that meets the particular requirements of their work.

In short, the "industry" of professional programmers can only meet the general needs of end users, not their "exact needs", and end users must fill the resulting functional gap by creating code of their own.

Assumption: The industry sells components to a "purchaser".

The vision presented by McIlroy implicitly presents a producer-consumer relationship, wherein the component consumer acts as a relatively passive purchaser. This is a reasonable way of viewing the role of a programmer who simply stitches together existing components that fairly closely meet an application's functional requirements.

However, this relationship is an incomplete description of situations where there is a large functional gap between the components and the finished program. In the case of end-user programming, the contributions of the end users sometimes require sizable expenditures of effort. In addition, the users' code sometimes embodies significant intellectual capital, such as when a spreadsheet implements an insurance company's proprietary pricing model. As a result, the contributions of the end users become valuable in their own right.

This raises the potential and indeed desirability of reusing end users' programs, to the extent that end users have similar needs that are unmet by the generic components offered by the component industry. For example, office workers sometimes reuse one another's web macros [14], scientists sometimes reuse one another's simulation and other analysis code [24], and system administrators sometimes reuse one another's scripts [1].

As a result, some end-user programmers play a dual role as consumers of professionally-produced code as well as producers of code that other end users might reuse.

Assumption: People can assemble programs from pieces that "fit together as building blocks" like "black boxes".

Even if end-user programmers could complete their work simply by stitching together professionally-produced components, it is still a somewhat vain hope that the pieces would always "fit together" seamlessly. Researchers have documented many different ways in which professionally-built components fail to mesh well, even when the components were explicitly designed for reuse [9]—even McIlroy's vision does not include compatibility between components of different families!

But in the context of end-user programming, black box reuse is further impeded by the fact that end users' code is rarely packaged up as a component. A component provides an interface specification that defines its properties, including the component's supported operations, functionality, and post-conditions guaranteed to be satisfied after operations are called [25]. Accordingly, McIlroy's vision phrases component reuse in terms of calling "routines" implemented as components' operations, with clearly defined functionality, precision, robustness, and time-space performance [26]. But the most popular end-user programming platforms offer no way to specify callable operations. For example, Excel spreadsheets do not expose any externally visible operations, nor do CoScripter macros (except for simply running a macro in its entirety). Not only do Excel and CoScripter lack any mechanism for exposing operations, but they provide no way to specify the properties of these hypothetical operations (such as post-conditions or time-space performance).

If end-user programmers want to package code for black box reuse, then they must rely on general-purpose languages and create components. However, such languages are well-known to be difficult for end-user programmers to learn, and it takes years for a people to develop the skills necessary to design robust component interfaces [27]. Few end users have overcome these hurdles: the former chief systems architect at BEA has estimated that for every programmer who knows how to use a Java-like programming language, there are nine more who can "build really complex spreadsheets, do Visio, use Access" [2]. In other words, for every programmer (professional or end user) who can use a language that supports creating components, there are nine more who can use create code that does not support creating components.

In such a context, it is unreasonable to expect that most code created by end users will be callable in a "black box" manner. Instead, end users will need alternate approaches for reusing one another's code.

Forms of reuse in the end-user programming context

Hoadley et al have identified three forms of code reuse by end-user programmers: code invocation, code cloning, and template use [11].

Table 1. Reuse in the world of end-user programmers rarely takes the form of simply stitching together "black box" components into a finished application

Provider of reusable code →	*Professional programmers*	*End-user programmers*
Form of reuse		
Code invocation / black box reuse	Common	Rare
Code cloning / white box reuse	Common	Common
Template reuse / conceptual reuse	Common	Common

Code invocation / black box reuse

Empirical studies confirm that it is fairly rare for end user code to be invoked through a callable interface. For example, one study by Segal revealed "plenty of evidence of the research scientists reusing publicly available code and components [produced by professional programmers]… But the only local reuse episodes we saw were two scientists who had a long history of working together." [24] Many scientists have a high level of programming skill, and they often use general-purpose programming languages (such as Fortran and C) that provide the syntactic constructs necessary to define components with precisely defined interfaces. Yet black box reuse of end-user code does not seem to thrive even in this propitious context, so we were unsurprised that we could not find any other empirical studies reporting situations where end users wrote programs that called one another's code.

Code cloning / white box reuse

When programming, end users typically create programs that are specialized to the specific task at hand. When later faced with a similar (but not identical) task, people can sometimes reuse an existing program as a starting point, though doing so will typically require making some edits. For example, one study of CoScripter macro use found that "in many cases, a user initially created a script with a hard-coded value and then went back and generalized the script to reference the Personal Database [a parameter]" [5]. In another study of end users who created kiosk software for museum displays, every single interviewee reported sometimes making a copy of an existing program that he or she wanted to reuse, then editing the program to make it match new requirements [6].

Hoadley et al refer to this form of reuse as "code cloning" because it so often involves making a copy of an existing program. However, as mentioned in the CoScripter macro study, end-user programmers sometimes edit the original program directly in order to make it general enough to support more than one situation. Therefore, this general category of reuse might be better identified with the more widely accepted phrase "white box reuse".

One inhibitor to white box reuse is that the end user must be able to find and understand code before he can reuse it. Consequently, end users are more likely to reuse their own code than that of other people [5][19][29].

This inhibitor can become a significant impediment to white box reuse of code written by professional programmers. For example, we are not aware of any empirical studies documenting situations where end-user programmers viewed the source code for professionally-produced components (despite the fact that many such components are now open source). The explanation for this might be that professional programmers commonly write code with languages, APIs, algorithms, and data structures that are unfamiliar or even unintelligible to most end users. In addition, there is often no easy way to find the source code for a professional's code. On the other hand, the professionals who created CoScripter have posted tutorial macros on the wiki, and end users have been able to read these macros, edited them, and executed them [5]. The macros were written in a form that the users could understand and customize.

Template reuse / conceptual reuse

End-user programmers often refer to existing programs as a source of information about how to create new programs. For instance, in one study of reuse by end-user programmers who created web applications, "interviewees commonly described using other people's code as a model for something they wanted to learn" [19]. The examples are sometimes provided by other end users and sometimes by professional programmers (often to facilitate writing code that calls their components or APIs). In many cases, the examples are fully-functional, so the programmer can try out the examples and better understand how they work [28].

Hoadley et al refer to this form of programming as "template reuse" because in the 1980's and early 1990's, researchers commonly used the word "template" to describe a mental plan for how to accomplish a programming task. Another appropriate name for this form of reuse might be "conceptual reuse", as in such cases, concepts are reused rather than actual code.

Identifying reusable end-user code

Given the value of reusing end users' code and the multiple ways in which that code can be reused, it might seem as though reuse of end-user code would be commonplace. Yet for every reusable piece of end-user code, many are never reused by anyone [10][21][24]. Actually identifying the reusable pieces of code within the mass of other code can be like looking for a needle in a haystack.

Conventional software engineering provides methods for assessing or ensuring the quality of code. These methods include formal verification, code generation by a trusted automatic generator, systematic testing, and empirical follow-up evaluation of how well the software works in practice. We have used the term "high-ceremony evidence" to describe the information produced by these methods [22], since applying them requires producers or consumers of code to exert high levels of skill and effort, in exchange for strong guarantees about code quality.

But end-user programmers (and some professional programmers) often lack the skill, time, and interest to apply these methods. What they need instead are methods based on "low-ceremony" evidence: information that may be informal, imprecise, and unreliable, but that can nevertheless be gathered, interpreted, and synthesized with a minimal amount of effort and skill in order to generate *confidence* (not a guarantee) that code is reusable.

Low-ceremony evidence would be particularly appropriate for end-user programmers because they rarely need the resulting program to perform perfectly. For example, in one study, teachers reported that their gradebook spreadsheets were "not life-and-death matters" [29], and in another study, web developers "did not see their efforts as 'high stakes' and held a correspondingly casual view of quality" [19]. For these people, the strong quality guarantees of high-ceremony methods are probably not worthwhile enough to

justify the requisite effort. But low-ceremony evidence might suffice, if it is possible to make reasonably accurate assessments of code's reusability based on whatever low-ceremony evidence is available about that code at a particular moment in time.

In the remainder of this chapter, we review empirical evidence showing that the reusability of end-user programmers' code can indeed be inferred from certain kinds of low-ceremony evidence. These studies allow us to develop a catalog of low-ceremony evidence that is known to relate to reuse of end-user programmers' code. We categorize the evidence based on its source: evidence based on the code itself, evidence based on the code's authorship, and evidence based on prior uses of the code. For example, code is more likely to be reusable if it contains variables rather than hard-coded values, if it was authored by somebody who has been assigned to create reusable code, and if it was previously used by other people who rated it highly. Our catalog of kinds of evidence can be extended in the future if additional studies empirically document new sources of low-ceremony evidence that are indicative of code reusability.

As shown in Table 2, we draw on 11 empirical studies of end-user programmers:

- Retrospective analyses of a web macro repository [5][21]
- Interviews of software kiosk designers [6]
- A report about running a Matlab code repository [10]
- Observations of college students in a classroom setting [12]
- An ethnography of spreadsheet programmers [17]
- Observations of children using programmable toys [18]
- Interviews [19] and a survey of web developers [30]
- Interviews of consultants and scientists [24]
- Interviews of K-12 teachers [29]

Where relevant, we supplement these with one simulation of end-user programmer behavior [4], as well as empirical work related to professional programmers [3][7][16][20]. In the sections below, we underline citations of work related to professional programmers in order to make it clear when a statement is only supported by research on professionals.

Table 2. Many studies mention evidence that is based on the code itself. This evidence contains information about mass appeal, flexibility, understandability, and functional size. A few studies mention evidence based on authorship or prior uses.

	Studies of end-user programmers												Studies of professional programmers			
Evidence based on																
Information about	[4]	[5]	[6]	[10]	[12]	[17]	[18]	[19]	[21]	[24]	[29]	[30]	[3]	[7]	[16]	[20]
Code itself																
Mass appeal					x				x			x	x			x
Flexibility		x			x		x	x	x		x	x	x			
Understandability			x		x	x		x	x	x	x		x			x
Functional size	x								x				x	x	x	
Code's authorship						x			x							
Code's prior uses				x												

After reviewing the three sources of low-ceremony quality evidence (the code itself, the code's authorship, and the code's prior uses), we summarize a recent study that combined evidence from the first two sources to accurately predict whether web macro scripts would be reused [21]. Our results open several research

opportunities aimed at further exploring the range and practical usefulness of low-ceremony evidence for identifying reusable end-user code.

Source #1: Evidence based on the code itself

If programmers could always create programs simply by stitching together some components chosen from a catalog, then the code implementing those components would not matter. More precisely, suppose that programming was as simple as selecting components based on their interfaces (which would include all the specifications called for by McIlroy, including functionality, precision, robustness, and time-space performance [26]), then writing a program that called components' operations exposed through their respective interfaces. The point of an interface is that it abstracts away the implementation. So the interface alone should provide sufficient information to formally prove that the resulting program would work as intended. Gathering any additional information about the code itself would be superfluous in terms of proving the correctness of the program. The number of lines of code, the number of comments, the actual programming keywords used, the coupling and cohesion of the implementing classes—none of it would matter. Even if the code was an unintelligible wad of spaghetti or a thick ball of mud—it would not matter.

Yet as argued earlier, end-user programmers rarely use one another's code through black box methods. Instead, they more typically rely on white box and conceptual reuse, both of which involve actually understanding the code.

White box and conceptual reuse are also important among professional programmers. It has been argued, and widely confirmed through experience, that professional programmers' code is more reusable if it has certain traits [3]. In particular, the code must be *relevant* to the requirements of multiple programming tasks, it must be *flexible* enough to meet those varying requirements, it must be *understandable* to the people who would reuse it, and it must be *functionally large* enough to justify reuse rather than coding from scratch.

These traits also apparently contribute to the reusability of end-user programmers' code, since every study of end-users cited in Table 2 produced findings of the form, "Code was hard to reuse unless it had X," where X was a piece of low-ceremony evidence related to one of these four code-based traits. For example, unless code contained comments, teachers had difficulty understanding and reusing it [29]. In this example, the evidence is the presence of comments in the code, and the trait is understandability. Thus, the evidence was an indicator of a trait, and thus an indicator (but not a guarantee) of reusability.

Information about mass appeal / functional relevance

The presence of keywords or other tokens in a certain piece of code appeared to be evidence of whether the code was relevant to many peoples' needs. For instance, web macros that operated on web sites with certain tokens in the URL (such as "google") were more likely to be reused by people other than the macro author [21]. Perhaps one reason why keywords were so predictive of reuse is that repositories and programming environments usually provide a search interface where users can type keywords to locate code [12], [15]. Thus, the presence of certain keywords can be evidence of mass appeal, suggesting a higher potential for reuse.

But when programmers seek reusable code, they are looking for more than certain keywords. Keywords are just a signal of what the programmer is really looking for: code that provides functionality required in the context of the programmer's work [7][12][20]. Typically, only a small amount of code is functionally relevant to many contexts, so a simple functional categorization of code can be evidence of its reusability. For example, 78% of mashup programmers in one survey created mapping mashups [30]. All other kinds of mashups were created by far fewer people. Thus, just knowing that a mashup component was related to mapping (rather than photos, news, trivia, or the study's other categories) suggested mass appeal.

Information about flexibility and composability

Reusable code must not only perform a relevant function, but it must do it in a flexible way so that it can be applied in new usage contexts. Flexibility can be evidenced by use of variables rather than hardcoded values. In a study of children, parameter-tweaking served as an easy way to "change the appearance, behaviour, or effect of an element [component]", often in preparation for composition of components into new programs [18]. Web macro scripts were more likely to be reused if they contained variables [5][21].

Flexibility can be limited when code has non-local effects that could affect the behavior of other code. Such effects reduce reusability because the programmer must carefully coordinate different pieces of code to work together [12][29]. For example, web page scripts were less reusable if they happened to "mess up the whole page" [19], rather than simply affected one widget on the page. In general, non-local effects are evidenced by the presence of operations in the code that write to non-local data structures (such as the web page's document object model).

Finally, flexibility can be limited when the code has dependencies on other code or data sources. If that other code or data become unavailable, then the dependent code becomes unusable [30]. Dependencies are evidenced by external references. For example, users were generally unable to reuse web macros that contained operations which read data from intranet sites (i.e.: sites that cannot be accessed unless the user was located on a certain local network) [5].

Information about understandability

Understanding code is an essential part of evaluating it, planning any modifications, and combining it with other code [12]. Moreover, understanding existing code can be valuable even if the programmer chooses not to directly incorporate it into a new project, since people often learn from existing code and use it as an example when writing code from scratch [19][20]. This highlights the value of existing code not only for verbatim blackbox or near-verbatim whitebox reuse, but also for indirect conceptual reuse.

Many studies of end-user programmers have noted that understandability is greatly facilitated by the presence of comments, documentation, and other secondary notation. Scientists often struggled to reuse code unless it was carefully documented [24], teachers' "comprehension was also slow and tedious because of the lack of documentation" [29], office workers often had to ask for help in order to reuse spreadsheets that lacked adequate labeling and comments [17], and web macros were much more likely to be reused if they contained comments [21]. End-user programmers typically skipped putting comments into code unless they intended for it to be reused [6]. In short, the presence of comments and other notations can be strong evidence of understandability and, indirectly, of reusability.

Information about functional size

When asked about whether and why they reuse code, professional programmers made "explicit in their verbalisation the trade-off between design and reuse cost" [7], preferring to reuse code only if the effort of doing so was much lower than the effort of implementing similar functionality from scratch. In general, larger components give a larger "payoff" than smaller components, with the caveat that larger components can be more specialized and therefore have less mass appeal [3]. Empirically, components that are reused tend to be larger than components that are not reused [16].

Simulations suggest that end-user programmers probably evaluate costs in a similar manner when deciding whether or not to reuse existing code [4], though we are not aware of any surveys or interviews which show that end-user programmers evaluate these costs consciously. Nonetheless, there is empirical evidence that functional size does affect reuse of end-user programmers' code. Specifically, web macros that were reused tended to have more lines of code than web macros that were not reused [21].

Source #2: Evidence based on code's authorship

In some organizations, certain end-user programmers have been tasked with cultivating a repository of reusable spreadsheets [17]. Thus, the identity of a spreadsheet's author might be evidence about the spreadsheet's reusability.

Even when an author's identity is unknown, certain evidence about the author can be useful for inferring code's reusability. For example, CoScripter web macros were more likely to be reused if they were uploaded by authors located at internet addresses belonging to IBM (which developed the CoScripter platform) [21]. In addition, web macros were more likely to be reused if they were created by authors who previously created heavily-reused macros.

Source #3: Evidence based on code's prior uses

Once someone has tried to reuse code, recording that person's experiences can capture information about the code's reusability. Repositories of end-user code typically record this information as reviews, recommendations, and ratings [10][15]. In the Matlab repository, capturing and displaying these forms of reusability evidence has helped users to find high-quality reusable code [10].

Identifying reusable end-user code based on low-ceremony evidence

As a first step toward finding effective models for combining low-ceremony evidence into predictions of reusability, we have designed and evaluated a machine learning model that predicts reuse of CoScripter web macros [21].

Before applying this model, it must be trained using information about macros (low-ceremony evidence) and about whether those macros were ever reused. In particular, while evaluating this model, we used low-ceremony evidence based on the code itself and authorship. For example, we used the number of comments in each web macro.

As described earlier in this chapter, reuse takes several forms in the end-user programming context, so we tested whether low-ceremony evidence would suffice for predicting different forms of reuse. In particular, we tested whether each macro would be executed by its author more than 1 day after its creation, whether it would be executed by users other than its author, whether it would be edited by users other than its author, and whether it would be copied by users other than its author. Black box reuse is detected by the first and second measures of reuse, while white box reuse is detected by the third and fourth measures. Admittedly, these measures are not perfect—particularly the second measure, which probably also captures information about conceptual reuse because users probably execute existing macros while learning from them in preparation for creating new macros. However, the measures do provide a good starting point for testing whether low-ceremony evidence provides enough information to identify reusable code.

Our machine learning training algorithm produces a set of simple arithmetic constraints that we call "predictors". For example, one predictor might be a constraint predicting that a macro would be executed by users other than its author if it contains more than 3 comments, and another might be that it would be executed by users other than its author if it referenced no more than 1 intranet site. Ideally, the predictors inferred for each measure of reuse would be true for every reused macro and false for every un-reused macro.

After the training process produces a set of predictors, a second algorithm uses this set to predict if each macro will be reused. The algorithm counts the number of predictors matched by the macro and predicts that it will be reused if it matches at least a certain number of predictors. The usual machine learning ten-fold validation approach was used (where nine-tenths of the macros were used for training and one-tenth for testing, and then the process was repeated ten times so that each macro had a chance to be in the testing set).

The model predicted reuse quite accurately, even using just the low-ceremony evidence collected directly from code and from information about the code's author. Specifically, the model predicted with 70-80% recall (at 40% false positive rate) whether other end-user programmers would reuse a given macro. The most useful predictors related to mass appeal, functional size, flexibility, and authorship.

These results show that low-ceremony evidence can be combined in a simple manner to yield accurate predictions of web macro reuse. While there may be other equally-accurate methods of combining evidence, our model has the advantage of being relatively simple, which might make it possible to automatically generate explanations of why the model generated certain predictions. Moreover, the model is defined in such a way that it does not require that the programs under consideration must be web macros. Thus, we are optimistic that it will be possible to apply the model to other kinds of end-user code.

Conclusion and future directions

Professional programmers cannot anticipate and provide components for every domain-specific need. To close this functional gap, end users create code that is valuable in its own right. Other end users often can benefit from reusing this code, but reusing it is not as simple as plucking components from a catalog and stitching them together. Reuse is more typically a multi-stage, white box process in which users search for useful code, attempt to understand it, and make needed modifications through cloning and/or editing. Users also need to understand code in order to learn reusable concepts from it. Empirical studies show that code with certain traits tends to be more reusable. At least in the web macro domain, it is actually possible to accurately predict whether code will be reused, based on information that may be informal, imprecise, and unreliable, but that can nevertheless be gathered, interpreted, and synthesized with a minimal amount of effort and skill.

These results represent one step toward providing end users with more effective approaches for quickly identifying reusable code created by other users, understanding that code, and adapting the code or learning from it in order to create new code. A great deal of additional work will be needed before end users obtain anything resembling the benefits originally promised by the stereotypical vision of component-based reuse.

First, we have found relatively few studies showing that code reuse is related to the code's authorship or prior uses. This was somewhat surprising, since evidence about prior uses has been incorporated into many repositories in the form of rating, review, and reputation features. Thus, one direction for future work is to perform more studies aimed at empirically identifying situations where this and other low-ceremony evidence helps to guide end-user programmers to highly reusable code. Further empirical studies might also help to extend our catalog by identifying new sources of low-ceremony evidence, beyond the code itself, authorship, and prior uses.

Second, it will be desirable to empirically confirm the generalizability of our machine learning model. This will require amassing logs of code reuse in some domain other than web macros (such as spreadsheets), collecting low-ceremony evidence for that kind of code, and testing the model on the data. At present, except for the CoScripter system, we are unaware of any end-user programming repository with enough history and users to support such an experiment. Ideally, just as we have drawn on research from studies performed by many teams, the machine learning model would be confirmed on different kinds of code by different research teams.

Third, our results create the opportunity to collect and exploit low-ceremony evidence in new system features aimed at supporting reuse. For example, code that matches many reuse predictors could be ranked higher in search results, potentially making it easier to discover reusable code. Having already shown that low-ceremony evidence is predictive of reusability, implementing features like these would show that it is possible to put these predictions to good practical use in a real system

Fourth, although these studies have shown the importance of understandability in promoting white box and conceptual reuse, there has been virtually no work aimed at helping end users to produce code that other people will be able to understand. For example, while virtually all of the studies emphasized the importance

of code comments in promoting understandability, most studies also showed that end-user programmers rarely take the time to embed comments in their code. End users lack the time to make significant up-front investments in understandability, yet this hampers reusability by their peers. New approaches are needed to break this deadlock.

Finally, a similar deadlock exists in the problem of designing code that other end-user programmers can easily adapt. Among professional programmers, it is widely accepted that good design promotes flexibility, chunks of manageable and useful functional size, and ultimately mass appeal. Yet end-user programmers often lack the time and skills to invest up-front in design. The resulting code can not only be hard to understand but also hard to adapt. End-user programmers need effective techniques and tools to support the creation of well-designed code, as well as the adaptation of poorly-designed code. Perhaps this might involve providing approaches that help users to create code that can more easily be reused in a black box fashion. In other cases, it will be necessary to develop techniques and tools for analyzing, refactoring, combining, and debugging existing code.

End-user code is not a simple thing, like a milk bottle, and helping end users to effectively reuse one another's code will require more than a vision of simply snapping components together like building blocks.

ACKNOWLEDGEMENTS

We thank the members of the EUSES Consortium for constructive discussions. This work was supported by the EUSES Consortium via NSF ITR-0325273, and by NSF grants CCF-0438929 and CCF-0613823. Opinions, findings, and recommendations are the authors' and not necessarily those of the sponsors.

REFERENCES

[1] R. Barrett, E. Kandogan, P. Maglio, and E. Haber. Field Studies of Computer System Administrators: Analysis of System Management Tools and Practices. *2004 ACM Conference on Computer Supported Cooperative Work*, 2004, 388-395.

[2] BEAs Bosworth: The World Needs Simpler Java, *eWeek Magazine*, February 19, 2004.

[3] T. Biggerstaff and C. Richter. Reusability Framework, Assessment, and Directions, *IEEE Software (4)*, No. 2, 41-49.

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

[5] C. Bogart, M. Burnett, A. Cypher, and C. Scaffidi. End-User Programming in the Wild: A Field Study of CoScripter Scripts, *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2008, 39-46.

[6] J. Brandt, P. Guo, J. Lewenstein, and S. Klemmer. Opportunistic Programming: How Rapid Ideation and Prototyping Occur in Practice, *4th Intl. Workshop on End-User Software Engineering*, 2008, 1-5.

[7] J. Burkhardt and F. Détienne. An Empirical Study of Software Reuse by Experts in Object-Oriented Design, *5th International Conference on Human-Computer Interaction*, 1995, 38-138.

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

[9] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch or Why It's Hard to Build Systems out of Existing Parts, *17th International Conference on Software Engineering*, 1995, 179-185.

[10] N. Gulley. Improving the Quality of Contributed Software and the MATLAB File Exchange, *2nd Workshop on End User Software Engineering*, 2006, 8-9.

[11] C. Hoadley, M. Linn, L. Mann, M. Clancy. When, Why and How do Novice Programmers Reuse Code, *6th Workshop on Empirical Studies of Programmers*, 1996, 109-129.

[12] A. Ko, B. Myers, and H. Aung. Six Learning Barriers in End-User Programming Systems. *2004 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2004, 199-206.

[13] A. Koesnandar, et al.. Using Assertions to Help End-User Programmers Create Dependable Web Macros, *16th International Symposium on Foundations of Software Engineering*, 2008, 124-134

[14] G. Leshed, E. Haber, T. Matthews, and T. Lau. CoScripter: Automating & Sharing How-To Knowledge in the Enterprise, *26th SIGCHI Conference on Human Factors in Computing Systems*, 2008, 1719-1728.

[15] G. Little, T. Lau, A. Cypher, J. Lin, E. Haber, and E. Kandogan. Koala: Capture, Share, Automate, Personalize Business Processes on the Web, *25th SIGCHI Conf. Human Factors in Computing Systems*, 2007, 943-946.
[16] P. Mohagheghi, R. Conradi, O. Killi, and H. Schwarz. An Empirical Study of Software Reuse vs. Defect-Density and Stability, *26th International Conference on Software Engineering*, 2004, 282-291.
[17] B. Nardi. *A Small Matter of Programming*, MIT Press, 1993.
[18] M. Petre and A. Blackwell. Children as Unwitting End-User Programmers, *2007 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2007, 239-242.
[19] M. Rosson, J. Ballin, and H. Nash. Everyday Programming: Challenges and Opportunities for Informal Web Development, *2004 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2004, 123-130.
[20] M. Rosson and J. Carroll. The Reuse of Uses in Smalltalk Programming, *Transactions on Computer-Human Interaction (3)*, No. 3, 1996, 219-253.
[21] C. Scaffidi, C. Bogart, M. Burnett, A. Cypher, B. Myers, and M. Shaw. Predicting Reuse of End-User Web Macro Scripts. Submitted to *2009 International Conference on Visual Languages and Human Centric Computing*.
[22] C. Scaffidi, M. Shaw. Toward a Calculus of Confidence, *1st Intl. Workshop on Economics of Software and Computation*, 2007.
[23] C. Scaffidi, M. Shaw, and B. Myers. Estimating the Numbers of End Users and End User Programmers, *2005 IEEE Symposium on Visual Languages and Human-Centric*, 2005, 207-214
[24] J. Segal. *Professional End User Developers and Software Development Knowledge*, Tech. Rpt. 2004/25, Dept. of Computing, Faculty of Mathematics and Computing, The Open University, Milton Keynes, United Kingdom, Oct 2004.
[25] M. Shaw, R. DeLine, D. Klein, and T. Ross. Abstractions for Software Architecture and Tools to Support Them, *IEEE Transactions on Software Engineering (21)*, No. 4, 1995, 314-335.
[26] *Software Engineering: Report on a conference sponsored by the NATO Science Committee* (P. Naur, B. Randell, eds.), Garmisch, Germany, October 1968.
[27] A. Spalter. Problems with Using Components in Educational Software, *2002 SIGGRAPH Conference Abstracts and Applications*, 2002, 25-29.
[28] R. Walpole, M. Burnett. Supporting Reuse of Evolving Visual Code, *1997 Symposium on Visual Languages*, 1997, 68-75.
[29] S. Wiedenbeck. Facilitators and Inhibitors of End-User Development by Teachers in a School Environment, *2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2005, 215-222.
[30] N. Zang, M. Beth Rosson, and V. Nasser. Mashups: Who? What? Why?. *26th SIGCHI Conference on Human Factors in Computing Systems - Work-in-Progress Posters*, 2008, 3171-3176.