***Chapter 21***

***Reuse in the world of end-user programmers***

***Authors: Christopher Scaffidi, Mary Shaw***

***Abstract:*** End-user programmers often reuse one another’s code when creating new programs, but this reuse is rarely as clean or simple as the blackbox reuse that professional programmers aspire to achieve. In this chapter, we explore the motivations that drive reuse of end-user code, the challenges that impede reuse of end-user code, and several approaches that facilitate reuse of end-user code. We give particular emphasis to the problem of helping end-users to identify high-quality reusable code amid the millions of scripts, macros, spreadsheets, and other programs produced by essentially anonymous users out on the web. Based on a review of empirical studies, we find that reusable code is often characterized by certain traits, suggesting that it might be possible to help end-user programmers to find reusable code by automatically gathering, analyzing, and synthesizing information about whether code has these desirable traits. We close the chapter with a discussion of future research opportunities aimed at helping end users to identify, understand, and adapt reusable code.

***Introduction***

Reuse is a central aspect of end-user programming. For example, empirical studies have shown that users commonly reuse one another’s CoScripter macros, which automate browser interactions with web sites (Chapter 5). Users sometimes execute existing macros without modification, but other times they modify another person’s macro before executing it. In addition, they sometimes combine existing macros via
copy-and-paste into a new macro. They also sometimes learn from one another’s macros before writing a new macro of their own, demonstrating conceptual if not literal code reuse. All these forms of reuse can help users to create new programs more quickly and correctly than they could by writing code totally from scratch.

Macros are not the only kind of code that end-user programmers reuse. Code includes any human-readable instructions that a computer can execute or translate to an executable form. For instance, people reuse spreadsheets [19], JavaScript and HTML [22], and Matlab programs [11].

But reuse of code created by other end users is often messy. For one thing, such code lacks formally-defined 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 design for reuse in the first place. Not all end-user code is easy to reuse.

This stands in contrast to the practices of professional programmers, who often can assemble applications by combining existing components. For instance, professional programmers now routinely create systems with database components or SMTP servers, or even smaller components like ActiveX controls. “Thousands of such controls have been developed, sold, and used. They are small, easy to work with, and an effective example of binary reuse” [16]. Despite the fact that programmers still encounter some problems when combining components [9], it should still be recognized that to a remarkable extent, professional programming has moved toward a vision where programmers are “able safely to regard components as black boxes” [29].
Yet reuse of user-created code lags behind. In this chapter, we explore several reasons why reuse of end-user code is desirable yet not as simple as snapping together black boxes. 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 focusing on the crucial problem of helping end-users to identify reusable code, which is a prerequisite to doing any reuse at all. 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 engineers 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 [29]:

“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.”
Even today, among some professional programmers, there is a view that programming can be as simple as hooking together black boxes [25]:

***[blockquote]***

“Components represent the ‘transistor’ of the software industry, or perhaps more accurately, the ‘integrated circuit.’ They provide a known quantity, a building block of established functionality and quality that can be used to assemble applications in a way more akin to assembly of a television than traditional development of an application.”

Achieving this idyllic vision would allow programmers to focus on selecting the right black boxes and hooking them together in a sensible way, rather than digging into the minutia of each box’s internal implementation. And in many circumstances, professional programming really is this simple. For example, accessing legacy COM objects in the more modern .NET programming platform is “surprisingly easy,” typically involving only a few lines of code [16]. Constructing the graphical layout of user interfaces in a C# or Java IDE is usually as simple as dragging and dropping icons [16][20]. Other easy-to-reuse components include database components, web services, XML parsers, and SMTP servers.

Combining black boxes to produce applications is such an effective approach for software development that it forms the basis for the “software factory” concept [10]. In this approach, a team is responsible for creating a “factory” that comprises highly-reusable components, recipes for combining those components, and tailored tools for
executing the recipes under programmers’ control. The factory then “can be used to rapidly and cheaply produce an open-ended set of unique variants of an archetypical product” [10]. In other words, the factory semi-automates the production of a product family. Certainly, not all organizations produce product families (so the software factory is not a “silver bullet”), but if an organization needs to create many similar applications in one domain, then the software factory is a valid approach to consider.

Such a vision might also 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.

***[H2]*** 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 [26]. 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
[15], advertising consultants creating web macros to compile sales information [14], managers creating spreadsheets to manage inventory [8], system administrators creating scripts to analyze server logs [1], and teachers creating programs to compute students’ grades [32]. 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 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 55 million end-user programmers (depending on the definition of “programming”) [26]. 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 are not enough professional programmers available to produce a software factory for every single group of end-user programmers. There simply is a huge range of domain-specific detail that only the end users themselves understand.
Thus, there 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”. Consequently, if users want to have the needed code, then they must fill the resulting functional gap by creating code of their own.

***[H2]*** 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 or uses a software factory. 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 many professional and end-user programmers, producing a final program often requires sizable expenditures of effort. In addition, end users’ code often embodies significant domain-specific intellectual capital, such as when a spreadsheet automates the computations in 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 [15], scientists sometimes reuse one another’s simulation and other analysis code [27], 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 “black boxes”**.

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 [28]. Accordingly, McIlroy’s vision phrases component reuse in terms of calling components’ operations, with clearly defined functionality, precision, robustness, and time-space performance [29].

But end users’ code is rarely packaged up as a component with a well-defined interface. In fact, the most popular end-user programming platforms do not even offer any way to specify interfaces. 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).
So if end-user programmers want to package code for reuse via interfaces, then they must rely on general-purpose languages. 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 [30]. 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 only know how to use typical end-user programming environments like Microsoft Excel, Visio, and 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 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 through interfaces. Instead, end users will need alternate approaches for reusing one another’s code.

***[H1]***Forms of reuse in the end-user programming context

Hoadley et al. have identified several forms of code reuse by end-user programmers: code invocation, code cloning, and template use [12] (Table 1). The producer of the reused code could be a professional or an end-user programmer.
Table 1. ***Caption*** Reuse in the world of end-user programmers rarely takes the form of simply stitching together “black box” components into a finished application

<table>
<thead>
<tr>
<th>Provider of reusable code</th>
<th>Professional programmers</th>
<th>End-user programmers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Form of reuse</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code invocation / black box reuse</td>
<td>Common</td>
<td>Rare</td>
</tr>
<tr>
<td>Code cloning / white box reuse</td>
<td>Common</td>
<td>Common</td>
</tr>
<tr>
<td>Template reuse / conceptual reuse</td>
<td>Common</td>
<td>Common</td>
</tr>
</tbody>
</table>

***[H2]*** Code invocation / black box reuse

Since Hoadley developed this categorization in 1996, empirical studies have shown that all of these forms of reuse are common, except for black box reuse of end-user code. Specifically, 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” [27]. 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][22][32].

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.

***[H2]*** 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” [22]. 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 [31].

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.
Low-ceremony evidence of 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 [11][24][27]. Actually identifying the reusable pieces of code within the mass of other code can be like looking for a needle in a haystack.

The same could also be said for reuse of professional programmers’ code, but conventional software engineering has well-established 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 [25], 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. Many professional programmers can call upon these techniques to look for the reusable code in the haystack.

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, analyzed, and synthesized with a minimal amount of effort and skill in order to generate confidence (not a guarantee) that code is reusable. Individual pieces of low-ceremony evidence are of low reliability, yet good decisions can be made on the basis of an accumulation of such evidence when that evidence provides generally consistent results.
For example, if a certain spreadsheet has been downloaded by dozens of well-respected co-workers, this fact would be low-ceremony evidence of reusability. It is informal because no formal process, notation, logic, or other formalized structure is prescribed for the gathering or use of the data underlying this fact. This evidence is imprecise, since different people might have downloaded the code for different reasons (for example, either to read the code or to use the code). This evidence is unreliable, since having numerous code downloads does not always imply reusability (for example, they might have all downloaded the code and found that it was full of bugs). Yet download counts are widely used and widely helpful, and they are claimed to play a crucial role in helping users to find reusable web macros [11] and Matlab code [17]. Such evidence can be easily gathered from server logs. It can be analyzed in many easy ways, whether by segmenting downloads by date or by groups of users (which can prompt insights such as realizing that certain code has fallen out of disfavor, or that a certain spreadsheet is only popular among accountants). Finally, such evidence can be synthesized with other evidence—for example, if the code has been downloaded many times, and all of those users continue to use the code on an ongoing basis, and if some of those co-workers are well-regarded as being technology savvy, then all of these forms of evidence can in the aggregate produce high confidence in the code’s reusability.

Low-ceremony evidence does not provide the strong guarantees of high-ceremony evidence. Rather, it provides indicators to guide decisions. This kind of 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” [32], and in another
study, web developers “did not see their efforts as ‘high stakes’ and held a correspondingly casual view of quality” [22]. For these people, the strong quality guarantees of high-ceremony methods probably do not provide enough value 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][24]
- Interviews of software kiosk designers [6]
- A report about running a Matlab code repository [11]
- Observations of college students in a classroom setting [13]
• An ethnography of spreadsheet programmers [19]
• Observations of children using programmable toys [21]
• Interviews [22] and a survey of web developers [33]
• Interviews of consultants and scientists [27]
• Interviews of K-12 teachers [32]

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][18][23]. 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.

<table>
<thead>
<tr>
<th>Evidence based on Code's authorship</th>
<th>Studies of end-user programmers</th>
<th>Studies of professional programmers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information about</td>
<td>[4] [5] [6] [11] [13] [19] [21] [22] [24] [27] [32] [33]</td>
<td>[3] [7] [18] [23]</td>
</tr>
<tr>
<td>Code itself</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mass appeal</td>
<td>x x x x x x x x x</td>
<td></td>
</tr>
<tr>
<td>Flexibility</td>
<td>x x x x x x x x</td>
<td></td>
</tr>
<tr>
<td>Understandability</td>
<td>x x x x x x x x</td>
<td>x x</td>
</tr>
<tr>
<td>Functional size</td>
<td>x x x x x x x</td>
<td>x x x</td>
</tr>
<tr>
<td>Code’s authorship</td>
<td>x x</td>
<td></td>
</tr>
<tr>
<td>Code’s prior uses</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>
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 [24]. 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.

***[H2]*** 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 and writing a program that called components’ operations through their interfaces. The point of an interface is that it abstracts away the implementation. So if the code supporting an interface has been formally validated, that 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 [32]. 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.
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” in “google.com”) were more likely to be reused by people other than the macro author [24]. If somebody is already familiar with a certain website, and has created some scripts which access that site, then he might be interested in other scripts that also target that site. Put another way, if there was a large supply of scripts containing a certain keyword, then there also was a large demand for scripts containing that keyword.

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][13][23]. 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 [33]. 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.

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 [21]. Web macro scripts were more likely to be reused if they contained variables [5][24].

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 [13][32]. For example, web developers considered scripts to be less reusable if they happened to “mess up the whole page” [22], 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 [33]. Dependencies are evidenced by external references. As an 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]. For instance, one web macro accesses the site “intranet.etc.com.co”, which can only be accessed by people who work at La Empresa de Telecomunicaciones de Bogotá (who owns the etb.com.co domain). Except for employees of this company, the script is entirely unreusable.
***[H3]*** Information about understandability

Understanding code is an essential part of evaluating it, planning any modifications, and combining it with other code [13]. 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 [22][23]. 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 [27], teachers’ “comprehension was also slow and tedious because of the lack of documentation” [32], office workers often had to ask for help in order to reuse spreadsheets that lacked adequate labeling and comments [19], and web macros were much more likely to be reused if they contained comments [24]. 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.

***[H3]*** 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 [18].

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 [24].

***[H2]*** 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 [19]. 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) [24]. In addition, web macros were more likely to be reused if they were created by authors who previously created heavily-reused macros.
***[H2]*** 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 [11][17]. In the Matlab repository, capturing and displaying these forms of reusability evidence has helped users to find high-quality reusable code [11].

***[H2]*** Untapped evidence

It is interesting that empirical studies have documented so few ways in which reuse of end-user code is affected by evidence based on authorship and prior uses (sources #2 and #3, above). In our view, this is somewhat surprising because analogous sources of evidence play a vital role in everyday life outside of code reuse. For example, when shopping for a new vacuum cleaner, it is not uncommon for people to consider information about the different vacuum cleaners’ manufacturers (analogous to authorship) as well as their previous experiences with vacuum cleaners (analogous to prior uses). Indeed, there are many forms of evidence analogous to authorship and prior uses that people rely on in everyday life:

- reviews of products by Underwriters Laboratories or by Consumer Reports and similar publications (reporting on prior uses of the products in a laboratory setting)
• third-party ratings and reviews of products by users at amazon.com or other online stores (reporting on prior uses of the products by other people)
• recommendations by co-workers or friends (again reporting on prior uses by other people)
• branding or seller reputation, including brand management through advertising (presenting evidence about the product producer)
• “best X” reports, such as Money Magazine’s “Best Graduate Programs” (often reporting based on surveys of people about their perceptions of product producers or products)

Outside of ordinary life, a great deal of this evidence is available to help professional programmers select components for reuse. For example, Dr Dobb’s Journal often provides reviews of products, programmersparadise.com programmers can rate components available for sale at, co-workers often offer opinions (sometimes rather zealously) about components, component producers manage their brands carefully through web sites and sometimes even television commercials, and sites for programmers like slashdot.org sometimes run polls about components, including the “favorite filesystem” poll.

Because of the importance of this information in everyday life and in the work of professional programmers, we anticipate that evidence about authorship and prior uses also plays an important role in guiding reuse of end-user code. Future research might be
able to uncover the nature of this evidence and develop approaches for using it to facilitate reuse.

**Predicting code reuse 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 [24]. This model classifies macros into two categories—“likely to be reused” or “unlikely to be reused”—based on features corresponding to low-ceremony evidence. These features include information about the macro’s source text (such as the number of comments and number of variables) as well as information about the macro’s authorship (such as the number of macros previously created by the macro’s author). Additional features can be added to the model in the future.

This model is based on the notion of evaluating how well scripts satisfy arithmetic rules that we call “predictors”. The first step is to train the model using an algorithm that selects rules (using an information theoretic criterion) which distinguish between macros that are reused and macros that are not reused. For example, one predictor might be that $\text{number\_of\_comments} \geq 3$, another might be that $\text{number\_of\_variables} \leq 2$, and a third might be $\text{number\_previously\_authored} \geq 1$.

After the model has been “loaded” with a set of predictors during training, it can be used to predict if some other macro will be reused. Specifically, a macro is classified as “likely to be reused” if it matches at least a certain number of predictors. Continuing the example above, requiring at least 1 predictor match would predict that a script will be
reused only if $\text{number\_of\_comments} \geq 3$ or $\text{number\_of\_variables} \leq 2$ or $\text{number\_previously\_authored} \geq 1$.

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 four different forms of reuse. In particular, we tested based on 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 of reuse are not perfect. However, the measures do provide a good starting point for testing whether low-ceremony evidence provides enough information to identify reusable code.

The model predicted reuse quite accurately, 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, analyzed, 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, our results create the opportunity to collect and exploit low-ceremony evidence in new system features aimed at supporting reuse. Several of the systems presented in this book would provide excellent test beds for this exploration. For example, the CoScripter repository (Chapter 5) could use the model presented in this...
chapter as a ranking function in search results, to be able to sort scripts by their reuse potential. Another possible application would be to integrate reusability scores with information from other sources, such as models of social networks; for example, if reuse is localized within a community, or among people with shared interests, then the CoScripter repository could rank code more highly if it appears to be highly reusable and if it was created by somebody with similar interests to the user running the search.

Systems might use evidence to rank code differently, depending on the user who is looking for code to reuse. For example, the designers of the ITL framework note that “Advanced users may be comfortable with complex structures, such as conditionals and iteration, but these may confuse novice users” (Chapter 11). Consequently, it might be useful for the repository to differentiate between advanced users and novices, presenting different search results to each. Novice users might be identified as those who have never uploaded code that had conditionals and iteration; advanced users then would be those who have uploaded such code at least once. When presenting search results to a novice user, the repository could filter out (or down-rank) code that contained conditionals and iteration, but it would use no such filter for advanced users. In this case, the evidence of reusability (the presence of conditionals and iteration) would be used to characterize users and code, with the goal of only presenting hard-to-understand code to users who have previously presented evidence that they are capable of understanding such code. Of course, any such innovation would need thorough testing to determine whether it is effective at linking users with useful code.

As one final idea for how to apply low-ceremony evidence in the context of end-user programming for the web, consider the notion of “trusted” and “untrusted” widgets.
created by users for users in the Mash Maker system (Chapter 9). Trusted widgets are allowed to execute certain operations in the user’s browser that untrusted widgets are not allowed to execute (such as to read cookies transmitted between the browser and server). To become trusted, a widget must receive an imprimatur provided by the Mash Maker system administrators at Intel. This creates a bottleneck – no widget can become trusted until after Intel’s employees have a chance to look at it. One possible improvement would be to let users decide for themselves whether they want to trust a widget that another user created, based on low-ceremony evidence. For example, after downloading an untrusted widget, the user could click on a button in the Mash Maker client to view evidence about trustworthiness. The client might indicate, for instance, “This widget does not call any API functions that could send data outside your browser. It was authored by a user who was confirmed to have the email address someuser@company.com. It has previously been trusted by 9 people whose code you have previously trusted, including your-own-sysadmin@your-own-company.com.” Depending on whether the user is persuaded by this evidence, he could decide to trust the widget or not.

Second, 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.
Third, 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.

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, and helping end users to effectively reuse one another’s code will require more than simply snapping together building blocks.

***[H1]***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.

***[H1]***References


[29] Software Engineering: Report on a conference sponsored by the NATO Science Committee (P. Naur, B. Randell, eds.), Garmisch, Germany, October 1968.


