Proceedings of the ACM RecSys 2010 Workshop on User-Centric Evaluation of Recommender Systems and Their Interfaces (UCERSTI), Barcelona, Spain, Sep 30, 2010
Published by CEUR-WS.org, ISSN 1613-0073, online ceur-ws.org/Vol-612/paper7.pdf

Recommendations for End-User Development

Will Haines
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025
1 650-859-6153
haines@ai.sri.com

Melinda Gervasio
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025
1 650-859-4411
gervasio@ai.sri.com

Aaron Spaulding
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025
1 650-859-3911
spaulding@ai.sri.com

Bart Peintner
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025
1 650-859-3209
peintner@ai.sri.com

ABSTRACT

End-user development (EUD), the practice of users creating, modifying, or extending programs for personal use, is a valuable but often challenging task for nonprogrammers. From the beginning, EUD systems have shown that recommendations can improve the user experience. However, these usability improvements are limited by a reliance on handcrafted rules and heuristics to generate reasonable and useful suggestions. When the number of possible recommendations is large or the available context is too limited for traditional reasoning techniques, recommender technologies present a promising solution. In this paper, we provide an overview of the state of the art in end-user development, focusing on the different kinds of recommendations made to users. We identify four classes of suggestion that could most directly benefit from existing recommendation techniques. Along the way we explore straightforward applications of recommender algorithms as well as a few difficult but high-value recommendation problems in EUD. We discuss the ways that EUD systems have been evaluated in the past and suggest the modifications necessary to evaluate recommenders within the EUD context. We highlight EUD research as one area that can facilitate the transition of recommender system evaluation from algorithmic performance evaluation to a more user-centered approach. We conclude by restating our findings as a new set of research challenges for the recommender systems community.

Categories and Subject Descriptors

H.1.2 [**Models and Principles**]: User/Machine Systems – *end-user development, recommender systems.*

General Terms

Algorithms, Design, Experimentation, Human Factors.

Keywords

end-user development, recommender systems.

1. INTRODUCTION: THE CASE FOR END-USER DEVELOPMENT

Computing devices are ubiquitous in today's professional environments and are increasingly invading our homes and mobile lives. Unfortunately, a deep understanding of these systems, and the ability to modify them, remains confined to the realm of the specialist. While the human-computer interaction community has made great strides in improving software usability, it has devoted far less attention to making systems customizable by end-users [22]. We argue here that existing recommender techniques can make a meaningful contribution toward increasing the usability, and therefore the acceptance and proliferation, of customizable software.

The current state of the art in user-centered software design is to engage users in an iterative design process and to test systems with users to refine the interaction design. Customization, if available, is built into the system at design time as a bounded number of user-selectable options. However, as workflows and processes change over time, even customized software applications need updates. Currently, the burden of these updates falls on the shoulders of professional developers, but the pool of users needing customizations is expected to grow much faster than the supply of professional software engineers [3].

One promising approach is end-user development (EUD), the practice of users creating, modifying, or extending programs for personal use [22,18]. This approach has two main benefits. One, it puts systems design in the hands of the domain experts who are most familiar with what needs should be met. Two, it scales with both a rapid increase in users and the increasing rate of change of many business processes. Unfortunately, EUD faces one major challenge—most end users do not have the specialized knowledge currently required to perform even basic development tasks [25].

As such, EUD research mainly focuses on approaches for lowering the barrier of entry to software development. Such approaches cover a wide spectrum, from enhancing the macros and spreadsheets that millions use every day to sophisticated algorithms that create programs by example without ever exposing the user to textual code [22]. While the technology behind these approaches may vary a great deal, there are some crosscutting techniques that seem to improve usability across the spectrum of EUD systems. In this paper, we will discuss one particular mechanism for improving user performance—system-generated recommendations.

As early as 1991, EUD systems like EAGER were using simple proactive suggestions as a component of their user interaction [6]. When the system detects that the user is performing an iterative task, it suggests a sequence of actions for completing the iteration automatically. In this case, the recommendation algorithm is straightforward, owing to the fact that there is only one recommendation type. If the system can complete the iteration it makes the recommendation, otherwise it does not.

Nearly twenty years later, advances in EUD systems have greatly expanded the opportunities for offering recommendations; however, most approaches continue to rely on constrained forms of recommendation that use handcrafted rules to make limited types of suggestions. In this paper, we explore the different classes of recommendations made in EUD systems, highlighting the areas where current approaches fall short and how recommender technologies can help fill the gap. Concurrently, we emphasize the ways in which these systems have been evaluated to date and discuss the ways in which such approaches will need

Proceedings of the ACM RecSys 2010 Workshop on User-Centric Evaluation of Recommender Systems and Their Interfaces (UCERSTI),
Barcelona, Spain, Sep 30, 2010
Published by CEUR-WS.org, ISSN 1613-0073, online ceur-ws.org/Vol-612/paper7.pdf

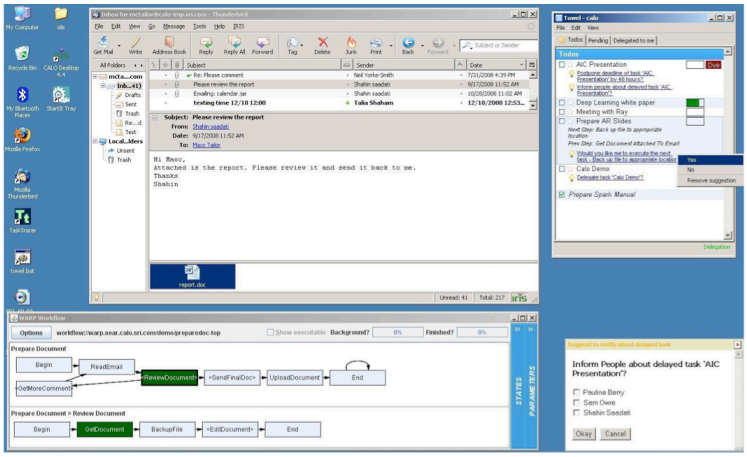

Figure 1. WARP identifies that the user is delayed in creating a document and offers to help [33].

modification to successfully evaluate integrated recommenders. We believe that by evaluating recommender systems in a user-focused context like EUD, researchers can facilitate the transition of recommender system evaluation from a focus on algorithm performance evaluation to a more user-centered approach.

2. INTEGRATING AUTOMATION INTO THE USER'S WORKFLOW

A core tenet of user-centered design is to create systems that mesh with users' existing workflows and environments [3]. EAGER is an early example of a tool that incorporates system suggestions to help "fit [end-user development] into the user's existing workflows" [18]. The focus on workflow integration is very important; it is likely that users will ignore an EUD system whose barrier to entry is too high. One approach to workflow integration, suggested by Lieberman, is to make "the cognitive load of switching from using to adapting … as low as possible" [22]. One way EUD systems have achieved this low barrier is by offering to automate portions of the user's workflow, essentially bypassing an explicit programming process and attempting to directly accomplish the user's task instead.

2.1 Current Approaches

One of the earliest approaches to EUD was programming by demonstration (PBD), also known as programming by example [5,21]. Many PBD systems rely on users explicitly demonstrating the process to be automated. However, some systems instead rely on implicit examples, continuously observing the user's actions to find repetitions over which they can learn a looping program to complete the user's task. Examples include EAGER, Dynamic Macro, and APE [6,24,28]. By recommending automation directly within the user's workflow, these systems achieve EUD transparently, without the user's awareness of having programmed the system. However, they are limited to automating repetitive tasks within the space of the looping programs they can generate.

A more general approach to automation within the user's workflow relies on activity recognition to observe what the user is doing and infer what that user is trying to accomplish. In combination with some mechanism for determining appropriate assistance, the system can use activity recognition to assist with the completion of a task. For example, Lumière uses Bayesian user models to offer context-dependent assistance [13]. While Lumière can offer assistance on a wider variety of tasks than the PBD systems focused on repetitive tasks, it is limited to assisting the tasks encoded by the developers, unlike PBD systems, which acquire looping programs on the fly.

Some systems combine aspects of both approaches. WARP (Figure 1), like Lumière, utilizes probabilistic models for activity recognition on a wide variety of tasks [33]. However, its meta-level assistance patterns are designed to work over a knowledge base of procedures rather than a developer-defined set of tasks. Because of this, the system not only can offer to automate more complex tasks but, through EUD, can continue to extend its knowledge base to handle a wider variety of tasks.

Task Assistant is another system that makes recommendations over an extensible knowledge base [27]. It allows users to explicitly define a workflow that groups of users collaboratively execute. It promotes automation by allowing the user to manually attach automated procedures, often produced by EUD, that support individual tasks in the workflow. Task Assistant uses these manual attachments to inform its suggestions for attaching other automated procedures to future tasks.

2.2 Recommender Systems Opportunities

EUD systems that make automation recommendations within a user's workflow provide a gentle transition from the user's core workflow into the world of programming, as users are generally not even aware that programs are being created or selected for execution. While current systems already provide useful EUD assistance, the use of recommender technology raises the possibility of further improvement.

2.2.1 Recommending shared procedures

EUD systems such as WARP and Task Assistant, which utilize a potentially unbounded set of automated procedures, offer the greatest opportunity for the application of recommender technology. Consider a shared procedure repository for the members of an organization. As the library of assistive procedures grows, handcrafting the rules or patterns for recommendation becomes more difficult. The problem is exacerbated when there are multiple criteria for evaluating procedure quality or applicability within a given context. As new procedure sources (e.g., web services, new EUD systems) continue to proliferate, the problem of finding and automating procedures and associating them to user tasks and workflows will only increase in frequency and importance.

Proceedings of the ACM RecSys 2010 Workshop on User-Centric Evaluation of Recommender Systems and Their Interfaces (UCERSTI),
Barcelona, Spain, Sep 30, 2010
Published by CEUR-WS.org, ISSN 1613-0073, online ceur-ws.org/Vol-612/paper7.pdf

For example, suppose the system determines that a user wants to schedule a meeting with some coworkers. The procedure repository may contain dozens of meeting scheduling procedures—some idiosyncratic to a given organization, some specific to certain types of meetings, and some buggy or outdated. Even if the procedure repository was constrained only to meeting scheduling, it would be difficult to craft a set of rules that covered all situations. Using recommender technology, one could imagine leveraging information about what other users have found useful (or not) to make better decisions about what procedures to recommend. The challenge is in incorporating sufficient context from the user's current activity into the recommendation algorithms.

2.2.2 Improving activity recognition

Instead of applying recommender technology only after determining what the user is doing, one could also imagine a system like WARP applying a recommender within the activity recognition algorithm itself. In this case, the algorithm might be able to narrow its search space to the activities that similar people have automated or been assisted with in the past. Such an enhancement would be particularly valuable in the case where the number of identifiable activities is very high.

2.3 Evaluation

To evaluate a recommender's ability to improve an EUD system's level of integration into a user's workflow, system designers must take a more user-centered approach than is traditional in recommender system evaluation. Instead of focusing on algorithmic performance of independent predictions involving a single, primitive task, EUD evaluation must situate itself in the context of a user performing a task, often comprising multiple subtasks. For Dynamic Macro and APE, this meant considering task performance time and user acceptance as part of the success criteria for the application [24,28]. Other systems may be able perform post-hoc analysis on user logs and avoid addressing the user's workflow directly, but regardless of methodology, the type and complexity of the user's task to automate will have some effect on the evaluation results [9].

The tasks supported by EUD systems are generally more complex than the simple viewing or purchasing decisions assisted by a traditional recommender system, so task-oriented evaluation can become particularly tricky. When testing even a simple task, it is can be difficult to tease apart user interface and algorithmic concerns. When the task becomes more complex, a simple design mistake can limit the user's ability to complete a task, rendering analysis of a recommender's efficacy difficult.

In such cases, it is desirable to control for user interface variation using a two-step process. First, a series of short qualitative studies can quickly identify high-priority user interface problems that can confound later study results. We find that think-aloud and heuristic evaluation protocols are well suited for this purpose [31,26]. After resolving the issues identified qualitatively, one can perform a controlled experiment that compares the user interface with a naïve recommendation implementation against the same user interface backed by more sophisticated algorithms.

A final consideration for evaluating increased automation and activity recognition is that recommendations may occur infrequently in comparison to the entire duration of the user's workflow. In this situation, longitudinal study protocols are appropriate. In the PBD space, Dynamic Macros and LAPDOG both worked with logs collected over an extended period of time [24,8]. For activity recognition, such evaluations do not yet exist in the literature, but one possible approach is a hybrid diary/log study that captures both logging information about when a task is recognized appropriately and diary entries capturing instances where the user expected a recommendation and received none.

3. HELPING THE USER MAKE THE RIGHT DECISIONS

The cognitive burden on end-user developers can be further reduced, and their task performance improved, by providing recommendations that help them make better programming decisions. This approach can be of particularly high impact in EUD systems that require a mixed-initiative interaction with the user—with appropriate recommendations, the user and system can move through their dialog more quickly and with fewer errors. By using the recommendations to sort lists of programming decisions by likelihood of correctness, EUD systems can help users make the right decisions in an unobtrusive, easy-to-override fashion.

3.1 Current Approaches

The Integrated Task Learning (ITL) system provides a procedure editor that takes advantage of recommendations to produce sensible defaults [11]. For example, to edit the data flow in a procedure, a user clicks on the argument to edit, and the procedure editor provides a list of suggested changes (Figure 2). The suggestions in this case are based on reasoning about the procedure using scope and type information. This bounds the recommendation space to suggestions that would not result in an invalid procedure; however, the system currently makes no attempt to rank the suggestions in any other way.

Another task in EUD where sensible default behavior is important is the generalization performed by PBD systems [5,21]. Because users need to provide the demonstrations from which PBD systems learn, a primary challenge is to find the correct generalization with as few examples as possible. One way to do this is to involve the user in the generalization process by presenting candidate generalizations and letting the user pick the correct one. Past systems have explored several different methods for selecting the candidates to present to the user. For example, SMARTedit performs conservative generalization within a well-defined version space and then uses a probabilistic weighting scheme to rank alternative generalizations [38].

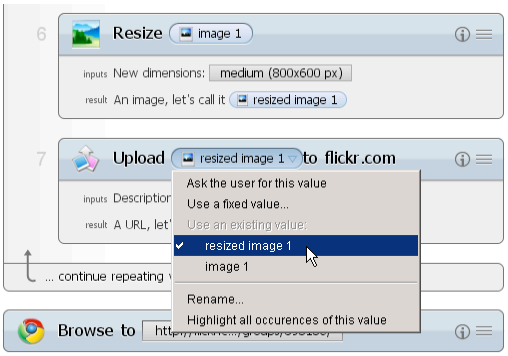

Figure 2. Clicking on 'resized image 1' in ITL provides suggestions under the heading 'Use an existing value:'.

Proceedings of the ACM RecSys 2010 Workshop on User-Centric Evaluation of Recommender Systems and Their Interfaces (UCERSTI), Barcelona, Spain, Sep 30, 2010
Published by CEUR-WS.org, ISSN 1613-0073, online ceur-ws.org/Vol-612/paper7.pdf

CHINLE uses a similar scheme not just to rank the candidates but also to color-code its interface [9]. LAPDOG uses heuristics to guide the inferencing it performs to explain connections between parameters and filter out unlikely generalizations [15].

3.2 Recommender Systems Opportunities

For this class of problem, recommender systems would likely sit behind the scenes, transparently organizing the plethora of options that systems currently present to the user. Potentially, this input could even cause systems to suppress options that are highly unlikely to be to be useful.

3.2.1 Suggesting preferred defaults

For a system like ITL, the primary limitation for reasoning is that for large procedures, the number of valid suggestions can be large, and if the user's desired edit does not appear high in the list of alternatives, that user's task performance can suffer. As with procedure recommendation, in large repositories, collaborative or social recommender techniques could also be used to improve recommended edits. However, a key difference is that the possible edits are information sparse compared to procedures. Algorithms would need to utilize the recommendation context even more to make appropriate suggestions, i.e. "other users *in this situation* tended to make change A" rather than "other users *generally like you* tended to make change A." Leveraging this context presents a challenge that recommender systems are only beginning to explore [1].

3.2.2 Suggesting more likely generalizations

The approach of presenting users with candidate generalizations and letting them choose the correct generalization is appealing because it transforms the difficult problem (for the system) of divining user intent into the relatively easier problem (for the user) of recognizing the correct generalization. However, with very few demonstrations, the number of alternative generalizations can become prohibitively large and determining the best ones may be difficult. PBD systems can thus benefit from mechanisms to organize the space of options and provide guidance toward reasonable generalizations.

Collaborative or social recommendations may help determine what and how to generalize. For example, in SMARTedit, a recommender system could speed up learning by suggesting a higher level of generalization based on how other similar actions were generalized in past procedures. And in LAPDOG, the preferences over different explanations could be derived from the explanations selected to explain similar actions in procedures developed by colleagues within an organization.

3.3 Evaluation

When determining the extent to which recommendations help users to make appropriate decisions, there are two phases in which a rigorous evaluation can be helpful: design time and implementation time. Post-implementation evaluations are common in the EUD literature but often fail to provide insight into the utility of the system in the field. Design evaluations are less common, although they are discussed for some systems, including ITL and CoScripter [11,20]. Such evaluations can define the types of recommendations that are beneficial to the user and to establish a set of ideal recommendations against which to evaluate actual algorithm performance in context.

For ITL, early application of think-aloud design evaluation methods proved useful to determine that recommended defaults would make a big difference in user performance [11]. Interestingly, it was possible to discover this insight using a wizard-of-oz protocol and thus the study required no actual recommender implementation. The lesson here is that it is possible to evaluate important EUD user interactions without having to settle on a recommender algorithm upfront.

In the space of recommending likely generalizations, evaluations have focused primarily on measuring whether the selected PBD algorithm can find the correct generalization—for example, in terms of whether it includes the correct generalization or if makes a correct predication using the generalization and ranks it highly [9,30,5]. In LAPDOG, there is an assumption that users will be able to recognize the correct generalization reasonably easily. However, many programming constructs, particularly when they include variables, are not easy for end users to comprehend. SMARTedit and CHINLE circumvent the generalization issue by presenting concrete predictions about the next step to execute in the context of a specific task. In this approach, users never have the sense of creating an actual program. Another interesting approach is *sloppy programming*, which represents programs in a pseudo-natural language that is interpretable by both humans and computers [22]. In general, PBD approaches that require the presentation of learned programs to users, whether for verification or for selection, need to identify the barriers to understanding such programs. For such cases, design-time survey methods may help to determine how users conceptualize the space of generalizations, leading system designers to more insight about how to create UI affordances to help users navigate the space of recommended generalizations.

4. HANDLING ERRORS

Much as is the case with professional programmers, end-user programmers spend a large proportion of their time debugging [29,14]. Consequently, many EUD systems provide mechanisms to help users identify and correct errors within their programs. Traditional debugging environments provide tools that allow developers to explore their code and track down errors; however, this exploration is left largely up to the user. A growing body of research suggests that for many users, this exploration process is hypothesis driven [10]. Unfortunately, when users do not have adequate knowledge of exactly how their programs execute, they can have difficultly formulating correct debugging hypotheses.

4.1 Current Approaches

Ko's Whyline provides a novel approach to supporting hypothesis-driven debugging by suggesting "Why?" questions for the user to explore when debugging (Figure 3) [17,15]. Its current implementation is more focused on professional developers than "end-user" developers, but it has been successfully implemented in novice programming environments and could be extended to pure-EUD systems [15].

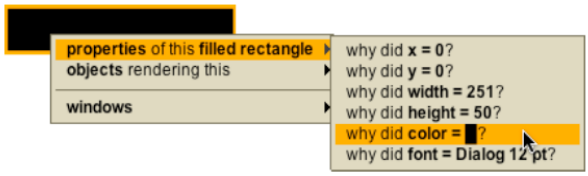

Figure 3. Whyline generates a set of possible questions about the black rectangle [17].

Proceedings of the ACM RecSys 2010 Workshop on User-Centric Evaluation of Recommender Systems and Their Interfaces (UCERSTI),
Barcelona, Spain, Sep 30, 2010
Published by CEUR-WS.org, ISSN 1613-0073, online ceur-ws.org/Vol-612/paper7.pdf

By recording and analyzing program execution and relating it back to the source code, Whyline is able to generate questions that focus on why a particular code segment behaved in a certain way rather than simply on why the program produced certain output [17]. By automatically identifying code related to failures, Whyline bounds the space of appropriate debugging hypotheses and helps programmers avoid guesses about irrelevant code segments.

Whyline generates its questions and answers automatically using heuristics and program slicing [16]. Questions are limited by two heuristics. The first heuristic relates to how users debug: hypotheses must reference observable failures. The second heuristic limits the number of possible entities to questions about code that the user wrote or directly referenced, with the expectation that the user will not ask questions about totally unfamiliar code. Once Whyline generates a set of questions, it extracts answers using program slicing to generate a "causal chain" of the code executed [16]. It then presents the questions to the user, fetching answers on demand.

Another approach to error handling is to take the act of debugging out of the user's hands entirely by automatically verifying the program for correctness before something goes wrong. For some classes of programming problems, EUD systems can detect errors and provide suggested fixes without forcing the user to search for problems. The ITL procedure editor takes this approach when possible (Figure 4) [11].

When the user performs an edit that causes an error, such as unbinding a variable, the editor detects the error via static analysis, marking the offending action. When the user clicks on the error icon, the system suggests solutions for that error. The solutions are parameterized to fit the given situation, but there are only a limited number of solutions at the present time [11].

4.2 Recommender Systems Opportunities

Whyline uses highly dynamic techniques to generate its hypothesis recommendations; however, it is still limited by its output and familiarity heuristics. Both exist to limit the number of questions and reduce time necessary to perform program slicing; however, as with all heuristics, they may not be correct all the time. By reducing the question space using recommendations rather than static heuristics, Whyline could find more bugs without sacrificing the processing time required to slice the program for every possible output primitive.

4.2.1 Suggesting potential problems based on similar programs

Users do not always test their programs on every possible execution path, so the observed output heuristic captures only a subset of the bugs that actually exist. A recommender that works over a community pool of debugged programs could suggest potential problem areas outside of the observed output based on problem areas in similar programs. This approach requires algorithms to be able to determine what features indicate programmatic similarity, a more difficult task than many traditional recommender applications. Nevertheless, programs are information-rich—the core problem is identifying what information is most relevant to determining similarity.

4.2.2 Suggesting examples

It is likely that users will be less successful debugging code with which they are unfamiliar. The familiarity heuristic weeds out

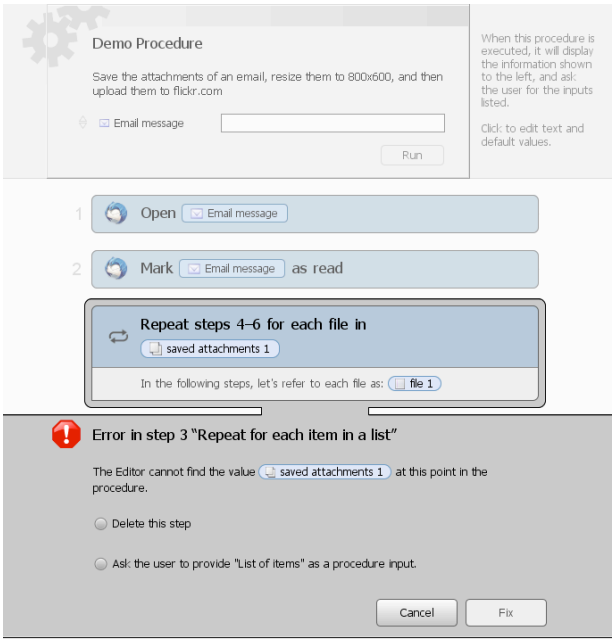

Figure 4. ITL alerts the user that 'saved attachments 1' is unbound and suggests two ways to fix the problem.

libraries that the user cannot modify without source code access, but it does not actually address the root cause of the unfamiliarity that leads to coding problems. A recommender of relevant code examples for problem application programming interfaces (APIs) could go a long way toward helping users debug calls into unfamiliar source. In fact, Brandt suggests that "opportunistic programming," based on web searches of code examples and tutorials, constitutes a primary programming approach for novice programmers [4]. His work on supporting this sort of programming focuses on how to improve user code browsing experiences, but recommendations do not make up the core of the approach. It may well be fruitful to see how system-driven recommendations interact with user-driven browsing to support opportunistic programmers.

4.2.3 Suggesting solutions to programming problems

Once errors are identified, recommending a subsequent fix is likely to be a highly relevant challenge for the recommender community. Here, current systems tend to pre-engineer relatively simple solutions to straightforward errors. If one could instead capture the ingenuity of an entire EUD system user community, it may be possible to propose solutions for more complex classes of problems. By creating a data set out of the problems and solutions faced by previous end-user developers, recommenders could suggest, and with some extension possibly apply, novel solutions to programming problems initially unanticipated by the creators of EUD systems. Such an advance would make EUD systems much more robust to unexpected programming challenges.

4.3 Evaluation

Whyline's evaluation consisted of a task-focused controlled experiment of debugging approaches [17]. Such a protocol also lends itself to the evaluation of error-handling recommenders because it provides a solid task context while still allowing the

Proceedings of the ACM RecSys 2010 Workshop on User-Centric Evaluation of Recommender Systems and Their Interfaces (UCERSTI),
Barcelona, Spain, Sep 30, 2010
Published by CEUR-WS.org, ISSN 1613-0073, online ceur-ws.org/Vol-612/paper7.pdf

experimenter to isolate algorithmic performance from user-interface design choices. For Whyline, a traditional breakpoint debugging system served as the control condition, and user task completion defined the success criteria. Thus, the experiment tested hypotheses about how the interaction design paradigm would affect user task performance

To test the recommender opportunities above, one would instead control for user interface deviation and vary the recommendations while still measuring task completion to evaluate the success of each condition. To manage user interface variation, we again suggest following the two-step approach advocated in section 2.3, first isolating and solving confounding user interface problems qualitatively, and then comparing task performance on interface without recommendations against an interface with recommendations.

Other possible conditions to test include recommendation domains of varying size or composition. Likewise, testing tunable parameters like recommendation confidence or recommendation diversity could yield insights into exactly what sorts of error-handling recommendations provide the most value to end user developers.

5. SUPPORTING UNPLANNED SHARING

End-user programmers, by definition, do not set out to create programs for others; however, unplanned sharing is a frequent side effect of EUD [23]. Even if end users do not plan on sharing their programs, they can often benefit from using someone else's code. While many early EUD systems lacked a community and, by extension, lacked sharing, several systems now leverage this powerful capability.

5.1 Current Approaches

Two systems that benefit from sharing are ITL, which provides a demonstration-based EUD environment for collaborative military command and control, and Task Assistant, which allows users to explicitly encode best-practice workflows in a sharable way [7,27]. An even more widely used application is the CoScripter web automation system. Its relatively wide adoption provides a case study in how sharing can affect a community of end-user programmers [20]. While both ITL and CoScripter provide end users with a rich repository of programs to share, neither currently provides sharing support past simple browsing and search.

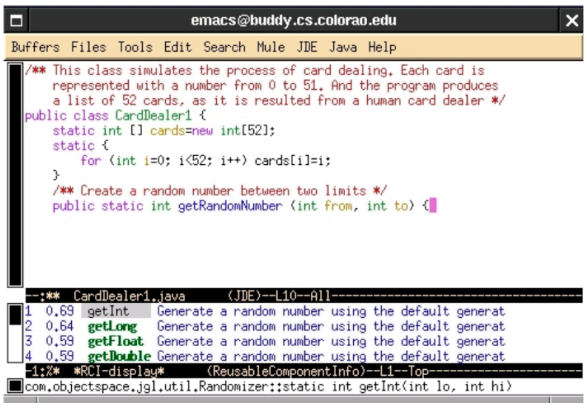

Figure 5. CodeBroker suggests that the user use the predefined getInt() method rather than the user's newly created getRandomNumber() method [32].

This sort of aid does not go far enough because users generally do not have sufficient familiarity with large codebases to know when code that solves their problems already exists [32]. Ye et al. suggest that "information push" (proactive recommendation) is one mechanism to assist users in such situations.

The CodeBroker system is one example of proactive development support. It recommends API calls to the user based on programming context (Figure 5) [32]. While it is not focused on "end-user" developers, it is straightforward to imagine using similar mechanisms for a code repository like the one used by CoScripter. CodeBroker generates its recommendations by gathering context from code comments. Then it applies Latent Semantic Analysis (LSA) to determine what components in the code base are conceptually similar to the concepts described by comments [19]. It further reduces the suggestion space by applying signature matching to only suggest API calls that are close type matches to the current calling context.

In the case that the above technique does not produce satisfactory results, the programmer can create a "discourse model" to manually identify suggestions that were not relevant [32]. Such irrelevant suggestions are not displayed in future queries. The system expands this concept to also include "user models" that manually identify code with which the user is already very familiar. Following the intuition that users do not want to get suggestions for familiar code, user models also serve to reduce the suggestion space for a given user. It is important to note, however, that in CodeBroker users manually create both the discourse and user models and must know a specific syntax for doing so.

5.2 Recommender Systems Opportunities

Community has not always been a central focus in EUD, but as recent systems push the boundaries toward more and more collaboration, community-based recommender systems seem to be a good fit. More users typically means more possible recommendations, and recommender algorithms can help EUD systems to cope with this increased scale.

5.2.1 Suggesting reusable code elements

LSA has been integrated previously into a collaborative filtering algorithm, so one could imagine implementing similar recommendation techniques within CodeBroker [12]. The system's major usability problem for end-user developers is likely to be its manually specified user and discourse models. Users are not apt to learn a brand-new programming syntax just to remove non-salient recommendations. If users can instead provide positive/negative feedback regarding the system recommendations, recommender technology could be used to acquire user profiles for making desirable recommendations based on individual user feedback as well as collective group feedback. In the latter case, by driving such a system with simple ratings rather than a complex programming syntax, users become more likely to actually provide the feedback necessary for the system to tailor its behavior to meet their preferences.

5.2.2 Assisting novices

Higher hurdles remain for EUD systems that are actually focused on novice programmers. Not only do these programmers not know what API calls exist, but they are also likely to misuse such calls. As discussed in Section 4.2.2, recommendations that also include code examples can be useful to teach novice opportunistic programmers how to reuse code.

Proceedings of the ACM RecSys 2010 Workshop on User-Centric Evaluation of Recommender Systems and Their Interfaces (UCERSTI), Barcelona, Spain, Sep 30, 2010
Published by CEUR-WS.org, ISSN 1613-0073, online ceur-ws.org/Vol-612/paper7.pdf

5.2.3 Suggesting best practices

Suggesting code reuse is only half of the battle. End-user developers are unlikely to aim to create reusable code when their core goal is to program for personal use [18]. This is unfortunate, because by creating more reusable code, users can not only facilitate reuse by others but also make more robust, flexible programs for themselves [2]. It is difficult to get users to code to appropriate levels of abstraction, and current EUD systems have to attempt to build such abstractions into the development environment [18]. One possibility is to recommend code structures that reflect the code abstractions created by the most experienced programmers in an EUD community. Recommender technology could facilitate suggesting appropriate code abstractions to novice programmers working on programs similar to those already solved by experts.

The workflows in Task Assistant explicitly encode best practices, and therefore are ripe for sharing. The difficulty for recommender systems in this case is the possibly endless copies of very similar workflows. For example, in a large company or community, subgroups may have many similar, but distinct, workflows for hiring a new employee. The best workflow for a given user depends on factors such as the user's usage history, which other users have used particular versions, and the relationship between the current user and the other users. A hybrid of collaborative filtering and more personalized recommendation techniques promises to improve such recommendations.

5.3 Evaluation

Sharing is a particularly difficult user need to evaluate because a realistic evaluation must be situated within a community of users. CoScripter was able to overcome this difficultly by selecting users to interview based on indicators gathered from broad-based logging of user activities [20]. Unfortunately, this approach is difficult to replicate without a critical mass of users such as the one CoScripter enjoys. Without such a user pool, the CodeBroker evaluation protocol falls back on extrapolating code sharing efficacy from precision and recall of the method calls suggested [32]. While these measures are valuable, they cannot tell the whole story of how a community-based sharing system will operate in practice.

In a situation where a large user pool is not available, we suggest an approach that combines a longitudinal protocol with user satisfaction surveys at regular intervals. In this case, users have a longer time in which to benefit from sharing, and the evaluation design explicitly addresses the user's perceived benefit from this sharing.

6. SUMMARY AND CONCLUSIONS

As a mechanism to improve usability, system-created recommendations have been a part of end-user development systems for a very long time. Unfortunately, few systems today take advantage of recommender technologies to cope with the increased scope of recommendation within EUD. Instead, handcrafted rules limit the types of suggestions that today's systems can make. As such, usability and adoption suffer.

We identify four main classes of recommendation that could be improved by using recommender technology:

- Inserting automation into the user's workflow
- Helping the user make the right decisions
- Handling errors
- Supporting unplanned sharing

Each of these classes of recommendation could make appearances in a variety of EUD systems, and as a whole they address user needs across the entirety of an end-user development workflow, from conceiving of a needed customization, to programming it, to sharing it with others.

In the space of inserting automation, recommenders could improve suggestions over shared repositories and perhaps directly improve activity recognition algorithms. Here, the major research challenges involve incorporating the user's operating environment into the recommendation algorithm to make the recommendations appropriately context-aware.

For improving user decision making, recommenders can help to organize the decision space, providing sensible defaults for a number of programming decisions, such as performing dataflow changes in an editor or choosing a best generalization in a PBD system. Challenges here include making recommendations about information-sparse decisions like simple edits. Again, context awareness could help to solve this problem.

When handling errors, opportunities for recommender systems are numerous, including suggesting potential problems, code examples, and possible fixes. While suggesting code examples is a fairly straightforward recommendation application, identifying potential problems and fixes exposes recommender algorithms to a nontraditional set of features to reason over. Learning exactly what features allow recommenders to identify similar programs is an interesting challenge we pose to the community.

Finally, supporting unplanned sharing allows recommender systems to apply their ability to leverage communities to the world of end-user development. Systems could help users to identify reusable components, especially aiding novices. Further, automatic identification of best practices could improve performance even for advanced end-user developers.

In all of these cases, end-user development provides a natural environment in which to evaluate recommendations in the context of real user workflows. Many evaluations in the end-user development space already combine user-centered evaluation methods with algorithm performance evaluation; user-centric evaluation of recommenders in this space simply requires evaluators to extend existing protocol designs to isolate the features unique to recommenders.

In conclusion, EUD systems have just begun to scratch the surface of how recommendations can improve user experience. By increasing communication between the EUD and recommender communities and creating recommendations that meet real user needs, we can move development out of the realm of the specialist and into the real world.

7. ACKNOWLEDGEMENTS

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. FA8750-07-D-0185/0004. Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA) or the Air Force Research Laboratory (AFRL).

Proceedings of the ACM RecSys 2010 Workshop on User-Centric Evaluation of Recommender Systems and Their Interfaces (UCERSTI), Barcelona, Spain, Sep 30, 2010
Published by CEUR-WS.org, ISSN 1613-0073, online ceur-ws.org/Vol-612/paper7.pdf

8. REFERENCES

[1] Adomavicius, G., Sankaranarayanan, R., Sen, S. and Tuzhilin, A. 2005. Incorporating contextual information in recommender systems using a multidimensional approach. *ACM Trans. on Information Systems*. 23, 1 (2005), 103–145.

[2] Basili, V.R., Briand, L.C. and Melo, W.L. 1996. How reuse influences productivity in object-oriented systems. *Communications of the ACM*. 39, 10 (1996), 116.

[3] Boehm, B.W., Madachy, R. and Steece, B. 2000. *Software Cost Estimation with Cocomo II*. Prentice Hall, NJ.

[4] Brandt, J., Guo, P., Lewenstein, J., Dontcheva, M. and Klemmer, S. 2009. Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. *Proc. 27th Conference on Human Factors in Computing Systems* (2009), 1589–1598.

[5] Cypher, A. and Halbert, D.C. 1993. *Watch What I Do: Programming by Demonstration*. MIT press.

[6] Cypher, A. 1991. EAGER: Programming repetitive tasks by example. *Proc. 9th Conference on Human Factors in Computing Systems* (New Orleans, LA, 1991), 33-39.

[7] Garvey, T., Gervasio, M., Lee, T., Myers, K., Angiolillo, C., Gaston, M., Knittel, J. and Kolojejchick, J. 2009. Learning by demonstration to support military planning and decision making. *Proc. 21st Conference on Applications of Artificial Intelligence* (2009).

[8] Gervasio, M., Lee, T.J. and Eker, S. Learning email procedures for the desktop. *Proc. AAAI 2008 Workshop on Enhanced Messaging*.

[9] Gervasio, M. and Murdock, J. 2009. What were you thinking? Filling in missing dataflow through inference in learning from demonstration. *Proc. 14th Conference on Intelligent User Interfaces* (2009).

[10] Gilmore, D.J. 1991. Models of debugging. *Acta Psychologica*. 78, 1-3 (1991), 151–172.

[11] Haines, W., Gervasio, M., Blythe, J., Lerman, K. and Spaulding, A. 2010. A world wider than the web: End user programming across multiple domains. *No Code Required*. Morgan Kaufmann. 213-231.

[12] Hofmann, T. 2003. Collaborative filtering via Gaussian probabilistic latent semantic analysis. *Proc. 26th Conference on Research and Development in Informaion Retrieval* (2003), 266.

[13] Horvitz, E., Breese, J., Heckerman, D., Hovel, D. and Rommelse, K. 1998. The Lumière project: Bayesian user modeling for inferring the goals and needs of software users. *Proc. of the 14th Conference on Uncertainty in Artificial Intelligence* (1998), 256–265.

[14] Ko, A.J., DeLine, R. and Venolia, G. 2007. Information needs in collocated software development teams. *Proc. 29th Conference on Software Engineering* (2007), 344–353.

[15] Ko, A.J. and Myers, B.A. 2004. Designing the whyline: A debugging interface for asking questions about program behavior. *Proc. 22nd Conference on Human Factors in Computing Systems* (2004), 158.

[16] Ko, A.J. and Myers, B.A. 2008. Debugging reinvented. *Proc. 30th Conference on Software Engineering* (2008), 301–310.

[17] Ko, A.J. and Myers, B.A. 2009. Finding causes of program output with the Java Whyline. *Proc. 27th Conference on Human Factors in Computing Systems* (2009), 1569–1578.

[18] Ko, A.J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwing, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., Rosson, M.B., Rothermel, G., Shaw, M. and Wiedenbeck, S. 2010. The state of the art in end-user software engineering. *ACM Computing Surveys*. (2010).

[19] Landauer, T.K. and Dumais, S.T. 1997. A solution to Plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge. *Psychological Review*. 104, 2 (1997), 211–240.

[20] Leshed, G., Haber, E.M., Matthews, T. and Lau, T. 2008. CoScripter: Automating & sharing how-to knowledge in the enterprise. *Proc. 26th Conference on Human Factors in Computing Systems*. (2008).

[21] Lieberman, H. 2001. *Your Wish is My Command: Programming by Example*. Morgan Kaufmann, San Francisco, CA.

[22] Lieberman, H., Paterno, F., Klann, M. and Wulf, V. 2006. End-user development: An emerging paradigm. *End User Development*. (2006), 1–8.

[23] Mackay, W.E. 1990. Patterns of sharing customizable software. *Proc. Conference on Computer-Supported Cooperative Work* (Los Angeles, CA, 1990), 209-221.

[24] Masui, T. and Nakayama, K. 1994. Repeat and predict: Two keys to efficient text editing. *Proc. 12th Conference on Human Factors in Computing* (1994), 118–130.

[25] Nardi, B.A. 1993. *A small matter of programming: Perspectives on end user computing*. The MIT Press.

[26] Nielsen, J. and Molich, R. 1990. Heuristic evaluation of user interfaces. *Proc. 8th Conference on Human Factors in Computing Systems* (1990), 249-256.

[27] Peintner, B., Dinger, J., Rodriguez, A. and Myers, K. 2009. Task assistant: Personalized task management for military environments. *IAAI-09*. (2009).

[28] Ruvini, J. and Dony, C. 2000. APE: Learning user's habits to automate repetitive tasks. *Proc. 5th Conference on Intelligent User Interfaces* (2000), 229–232.

[29] Tassey, G. 2002. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*. (2002).

[30] Wolfman, S., Lau, T., Domingos, P. and Weld, D. 2001. Mixed initiative interfaces for learning tasks: SMARTedit talks back. *Proc. 6th Conference on Intelligent User Interfaces* (2001), 167–174.

[31] Wright, P.C. and Monk, A.F. 1991. The use of think-aloud evaluation methods in design. *SIGCHI Bull.* 23, 1 (1991), 55-57.

[32] Ye, Y. and Fischer, G. 2005. Reuse-conducive development environments. *Automated Software Engineering*. 12, 2 (2005), 199–235.

[33] Yorke-Smith, N., Saadati, S., Myers, K.L. and Morley, D.N. 2009. Like an intuitive and courteous butler: A proactive personal agent for task management. *Proc. 8th Conference on Autonomous Agents and Multiagent Systems-Volume 1* (2009), 337–344.