End-User Mashup Programming: Through the Design Lens

Jill Cao[1], Yann Riche[2], Susan Wiedenbeck[3], Margaret Burnett[1], Valentina Grigoreanu[1,4]

[1]Oregon State University
Corvallis, OR
{caoch,burnett}@eecs.oregonstate.edu

[2]Riche Design
Seattle, WA
yann@yannriche.net

[3]Drexel University
Philadelphia, PA
sw53@drexel.edu

[4]Microsoft Corporation
Redmond, WA
valeng@microsoft.com

ABSTRACT

Programming has recently become more common among ordinary end users of computer systems. We believe that these end-user programmers are not just coders but also *designers,* in that they interlace making design decisions with coding rather than treating them as two separate phases. To better understand and provide support for the programming and design needs of end users, we propose a design theory-based approach to look at end-user programming. Toward this end, we conducted a think-aloud study with ten end users creating a web mashup. By analyzing users' verbal and behavioral data using Schön's reflection-in-action design model and the notion of ideations from creativity literature, we discovered insights into end-user programmers' problem-solving attempts, successes, and obstacles, with accompanying implications for the design of end-user programming environments for mashups. The contribution of our work is three-fold: 1) the methodology of using a design lens to view programming, 2) evidence, through insights gained, of the usefulness of this approach, and 3) the implications themselves.

Author Keywords

End-User Programming, Design, Mashups

ACM Classification Keywords

D.2.6 [Software Engineering]: Programming Environments—Interactive environments, D.2.10 [Software Engineering]: Design.

General Terms

Human Factors, Design

INTRODUCTION

(Overheard in a lunchroom): "Hi Mike, I liked your Venn Diagram design."

Mike is a professional software developer. As the conversation continued, it quickly became clear that his Venn Diagram design was something that emerged while he was *programming*—there was never any exit from the programming activity to engage in something most people would recognize as a *design* activity. Yet, the software developers at the table consistently referred to the result as a "design". Further, as his process was dissected, it was revealed to be an iteration through design possibilities, experimentation, and evaluation—in short, the steps identified in the literature as being the components of design.

So, what counts as design? Traditionally, design in software engineering has been considered as a front-end process followed by implementation. As such, the devising of a solution and its implementation were considered to be separate and sequential processes. This view has been continuously challenged by psychologists studying programming activities [31]. In particular, Gray and Anderson referred to design cycles that contained not just the traditional view of design as up-front planning, but also translating the abstract solution to implementation and then revising the implementation and/or one's understanding of the solution [8].

Recently, the software engineering community has adopted development methodologies that iterate between design and coding. Examples include agile development, Rational Unified Process, and the spiral model [28]. However, within these methodologies, design and implementation activities are still considered as separate, albeit iterative and conversing, phases of the software process.

We believe that what is seen as *just coding* from the view of traditional software engineering in fact is peppered at the microlevel with design decisions and, as such, much of it can be viewed as *designing*. We believe this is true not only of professional developers like Mike, but also of end-user programmers.

Nardi defined end-user programmers as being distinct from professional developers in that end users' programs are not the end in itself, but rather a means to accomplish their own tasks or hobbies [20]. End-user programmers often do not have professional computer science training, and there are a variety of research systems and tools aimed at this audience; examples include an accountant creating spreadsheet formulas (which are computation instructions) to keep track of a budget in Excel or a web-literate end user building a quick mashup to facilitate the planning of a night at the movies.

To investigate the role and impact of tiny instances of design that permeate the programming process, especially by end-user programmers, we adopted a *design lens* through which to view the activities of ten end-user programmers creating a web mashup. Our hope was that this investigation would demonstrate this approach's ability to shed critical insights on end-user programming. Thus, our research questions were:

CHI 2010, April 10–15, 2010, Atlanta, Georgia, USA.

RQ1: Can we describe end users' programming behaviors using theories and findings from design?

RQ2: Is doing so beneficial? For example, what implications for tool design can we discover?

We use these questions to explore a methodology for applying design perspectives to programming. We then present evidence, through insights gained, of the usefulness of using this methodology. Finally, along with the insights themselves into end-user programmers' problem-solving attempts, successes, and obstacles, we present associated implications for the design of end-user programming environments for mashups.

BACKGROUND, THEORIES AND RELATED WORK

Theoretical and Empirical Background

In the design community, Schön's reflection-in-action is an important design model that describes practitioners' ways of approaching ill-defined problems [26]. The process has three phases. *Framing* involves understanding and defining the problem. *Acting* aims to transform the current situation to a better one, or to learn more about the situation. *Reflecting* looks back on actions to assess their consequences and implications. The process is an iterative "conversation" [26], with moves from framing to acting to reflecting, and sometimes back to major reframing.

When designers, professional or not, sit down to create a design, they are likely not looking for repetition but for expressions of creativity. The creativity community has evolved theory and associated concepts that point to creativity, namely the three concepts of ideational fluency, flexibility, and elaboration. The literature has long argued for quantity of ideas as an indicator for creativity [5, 10]. Guilford introduced the concept of *ideational fluency*, i.e., the rate of generating ideas related to the creative output. Empirical evidence supports the construct validity of using ideational output as a measure for quality of responses [18]. *Flexibility* is defined as generating different types of ideas [10, 25]. Flexibility can be recognized when an individual moves from one ideational category to another [25]. *Elaboration* is described as the ability to extend basic information to a rich web of information [10]. In our study, we relate our findings to these three concepts.

Related Work

Research shows that professional software developers take different paths in the early stages of a design process [11]. One path is to fully specify the design problem early on. In this top-down, breadth-first approach each successive level of the decomposition is more detailed. The refinement process continues until the problem is fully specified. This approach is successful if the problem is well structured, with well-defined goals, knowledge of the domain, and no novelty in the problem. Lacking these characteristics, the alternative path is opportunistic decomposition, in which the software developer jumps into the design using a data-driven approach. Empirical studies, e.g., [11, 30], show that ill-structured problems lead to changes in high-level goals and new requirements. Thus, they come to the conclusion that the software design process *should* be opportunistic in these cases. Opportunism also appears in novice programmers' behavior. Studies show that novices designing a program will initially attempt to use top-down design, but it often fails because the programmers do not have the ability to decompose the problem, nor do they have stored plans to build on [13, 21]. Consequently, they start writing code without a plan, resulting in a bottom-up design. While the studies above involved professional developers' and novice programmers' approaches, our study gives us a chance to see those of end-user programmers.

Professional developers may prefer to specify the design problem early, but like novice programmers, end users often go directly to programming, grabbing opportunities as they arise. This lack of design planning is reminiscent of "debugging into existence" [22], i.e., ignoring analysis and incrementally developing a small part of the system then iteratively using the debugger to refine and correct problems. However, recent research suggests that design planning by end users is feasible. In an exploratory study [23], end-user web developers successfully carried out a design planning task before developing the application, and this was reflected in the implementation. Our work differed from [23] in that we did not prescribe design planning methods to participants but rather let them work however they preferred.

The work of Kannengiesser and Zhu [15] applied the function-behavior-structure (FBS) design model to various software design methods, e.g., the Rational Unified Process. The aim of this work was to develop a basis for empirical investigations of software design processes using the FBS model. Our work differs from theirs in several ways: we applied Schön's reflection-in-action model instead of the FBS model because it is suitable for describing practitioners' behaviors, our participants were end users rather than professional software developers, and our study was empirical instead of analytical.

EMPIRICAL STUDY

To see if we could apply design-related theories and findings to describe end users' programming activities, and to find out the benefits of doing so, we observed ten participants engaging in an end-user programming task, i.e., creating mashups. Mashups are web applications that interactively combine data from multiple internet sources [32]. We chose mashups because it is an emerging end-user programming paradigm. To achieve this task, participants used an online visual programming environment called Microsoft Popfly Mashup Creator. (Microsoft stopped supporting Popfly on August 24, 2009.)

Participants and Procedure

This study included four female and six male college students from a wide variety of majors (e.g., biology, nutrition science). None were computer science students or had taken computer science courses beyond the elementary level. One female and four males had past programming experience

either in high school, college, or both (one male had one course; the rest had two). All participants were comfortable with web browsing.

We used the think-aloud approach, conducting the study with one participant at a time. Participants first filled out a background questionnaire and worked on a hands-on tutorial in which they were allowed to ask questions (see Tutorials and Task). They then completed a self-efficacy questionnaire adapted from [6] to the specific task of end-user mashup creation. We collected self-efficacy scores because self-efficacy has previously been found to impact end users' approaches to programming tasks [3, 4, 9]. Participants then practiced the think-aloud procedure before proceeding to the main task. When participants stopped making progress, the researcher administered an additional mini-tutorial to help them (see Tutorials and Task). The data we collected included a video capture of participants' interactions with the environment (including their facial expressions), and participants' final mashups.

Environment

In Popfly, users build mashups using basic programming constructs called *blocks*. Each block performs a set of *operations* such as data retrieval and data display. Each operation takes *input* parameters to allow customization. Blocks are connected to form a network in which the output of a block can be used as input for adjacent blocks. Figure 1 shows a mashup example in which the Flickr block sends a list of images about "beaches" with their geographical coordinates to the Virtual Earth block (Figure 1: top and middle) to display them on a map (Figure 1: bottom). In Popfly, blocks are listed in different categories, which users can search. Additionally, users may share their mashups with others for reuse and modifications. Shared mashups can be retrieved using a textual search.

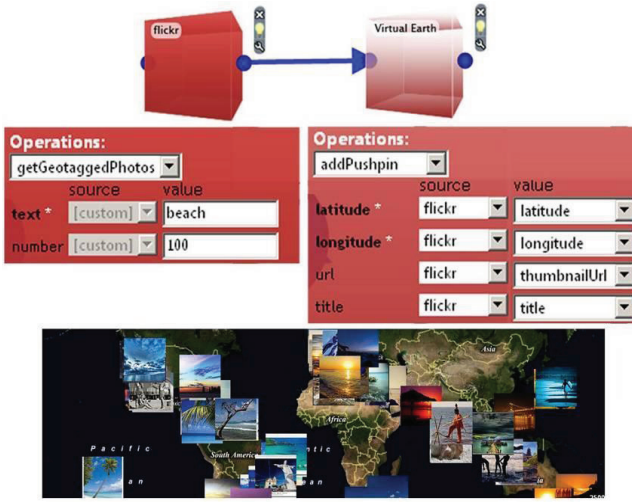

Figure 1. The pre-task tutorial example mashup in Popfly Mashup Creator. Top: the blocks. Middle: some of the blocks' settings. Bottom: results generated by pressing the Run button (not shown).

Tutorials and Task

The pre-task 20-minute tutorial provided participants with an introduction to mashups, and included two live examples of mashups before a short hands-on session. The hands-on session familiarized users with Popfly's basic features, how to search and modify other people's mashups, and the help feature. Figure 1 shows the mashup participants created during the tutorial.

During the task, participants who had not made progress for 15 consecutive minutes received an additional 5-minute tutorial to help them regain productivity. The tutorial consisted of creating two mini mashups. The decision on delivering this tutorial was based on the participants' demonstrating difficulty in generating new ideas to approach the task. Although the mid-task tutorial may have influenced participants' behaviors, we compare its effect to encountering a well-chosen example. The mid-task tutorial was given to half of the participants (two males and three females).

The task involved creating a mashup about movies shown in a city. Paper, pens and sticky notes were provided. The mashup required the following pieces of information: 1) a list of local theaters, 2) movies shown at each theater along with information, e.g., running and show times, 3) a picture, and 4) a news story for each movie. Prior to the study, we refined the experimental setup and the task with pilot runs.

METHODOLOGY

We coded the study's transcript with three code sets (Table 1): *reflection-in-action* [26] commonly used for studying design activities [2], *ideations* devised based on creativity literature [10], and *barriers* developed for the analysis of end-user programming [16].

For reflection-in-action, we used one code for each of the three steps in the reflection-in-action theory. *Framing* described events in which participants tried to understand and define the problem, either by generating a hypothesis to explore, or by gathering information to narrow down the design space. *Acting* described events in which participants started or changed their mashups. *Reflecting* described events in which participants evaluated their actions. Table 1 shows examples of each.

We built upon Guilford's notion of "ideational fluency" to create the ideations codes. As mentioned in the Background section, ideational fluency refers to rate of generating ideas [10]. To account for this notion, we coded expansions and contractions of the participant's working set of major ideas. Expansion describes a new idea to solve the problem, or the elaboration of an existing idea. In contrast, contraction is the abandonment of an existing idea. In our analysis, we only coded expansion/contraction if there was unambiguous evidence of an idea addition/deletion through their verbalization or action. As a result, this code set mainly expresses how ideation processes were reflected by actions carried out in the workspace. Finally, we coded five of Ko et al.'s programming barriers from the end-user programming literature [16] (see Table 1).

Code	Example
Reflection-in-action (mode switches)	
Framing	*It looks like I have to have multiple VirtualEarth.*
Acting	*[Adds another VirtualEarth block]*
Reflecting	*So it gives me the theaters, and the movies themselves.*
Ideations (instances)	
Ideas for blocks	Expansion: *[Adds LocalMovies to the workspace]*
	Contraction: *[Removes block LocalMovies]*
Ideas for which blocks to connect	Expansion: *So I need to connect LocalMovies to VirtualEarth*
	Contraction: *[Removes link from LocalMovies to Flickr]*
Ideas for block dependencies	Contraction then expansion: *[Changes value for title from Local Movies' Theater Name to Local Movies' Movie Name]*
Within-block ideas	Expansion: *[Types THEATER CITY STATE in title field]*
Barriers (instances)	
Design	*So I'm going to start all over. [Removes all blocks] I still don't understand…*
Selection	*Now I'm searching for information about each movie. I need to go where?*
Coordination	*I cannot see any pictures or MSN News from the results even though I had connected them together.*
Use	*I didn't use the right options.*
Understanding	*I don't know what happened and why it didn't work.*

Table 1. Code sets

For each code set, two researchers coded small portions of the transcripts independently and compared inter-coder agreement until they reached an 80% agreement covering at least 20% of the transcripts. Researchers then split up the remaining work and coded independently.

RESULTS

In existing environments for end-user programming such as the emerging ones for building mashups, there is no support for phases other than implementation. Thus, any designing that takes place occurs in the context of implementing and evaluating a program. To study the consequences of this attribute of end-user programming environments, we applied the reflection-in-action theory to examine how design was integrated into our participants' actions.

In this section, we present insights gained from our observations into mashup programming, as discovered through the application of the lens of design. First, we show an overview of how participants cycled through the different steps of the reflection-in-action model, namely framing, acting, and reflecting. Next, we present detailed insights gained using the design lens in each of the three stages. For each result, we provide implications for the design of mashup environments.

To provide context for the rest of the paper, we first provide the participants' success levels in achieving the given task. In particular, participants' IDs are ordered by their success levels as measured by the number of requirements they achieved during the task (listed in parentheses): F1(4), F2(2.5), F3(2), F4(2); M1(3.5), M2(3.5), M3(3), M4(3), M5(2.5), and M6(2). Half points indicate partially fulfilled requirements, e.g., not all movies showing a picture.

Reflection-in-action cycles

To get an overview of participants' behaviors, we graphed the result of the reflection-in-action code set over time. As Figure 2 shows, participants made extensive use of all three phases, iterating tightly through the reflection-in-action cycles. In particular, we identified three common patterns: *stair-step*, *w*, and *restart*. The *stair-step* pattern refers to a succession of consecutive episodes of framing, acting and reflecting. The *w* pattern refers to participants switching more than one time from framing to acting and back or from acting to reflecting and back. The other pattern, which occurred occasionally, was the *restart* pattern, in which reflection led to a return to the framing stage. An example of each pattern is illustrated in Figure 2. Notably absent was any kind of waterfall-like pattern that would have featured a fairly long period of framing alone first, then a fairly long sequence of acting without returning to framing, then a sequence of reflecting. This indicates that our participants used a highly iterative development style—not one characterized by lack of design, but rather one peppered with numerous instances of "micro design".

Framing the Problem

"Create a mashup to…"—but how? In order to begin, one needs to have a grasp of the problem. The notion of framing captures efforts to understand and define the problem [26]. We discovered two issues participants encountered in the framing stage. First, successful participants' framing efforts often produced actionable ideas to guide their actions whereas the unsuccessful ones' often did not. Further, we found that unsuccessful framing episodes were often followed by design barriers. Second, participants' inclination to reframe in the face of failure differed. We discuss each of these issues next.

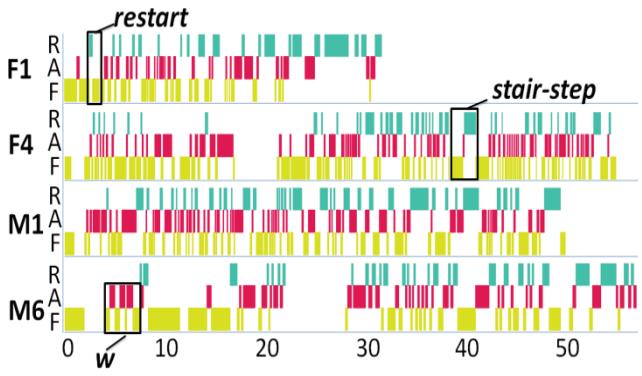

Figure 2. Reflection-in-action: examples from four participants: framing (bottom category of y-axis), acting (middle), and reflecting (top) over time. Blanks indicate time spent outside of reflection-in-action, e.g., taking the mid-task tutorial.

Good framing output guides effective actions

When analyzing our participants' framing episodes, we noticed that the successful episodes were able to suggest actionable ideas as to how to proceed with the task, whereas the unsuccessful often failed to do so. This difference drew our attention to the importance of framing's output, i.e., ideas that guided actions. In fact, visiting the framing phase, whether the participant exited with or without output, was often critical to the success of what came next.

For instance, M1's framing usually produced output in the form of actionable ideas, and his subsequent actions made direct use of those ideas. As an example, earlier in his task, he had set up Flickr to get pictures of movies. Then he discovered a problem in minute #11—his mashup did not return pictures. He re-entered the framing phase briefly in minute #11, producing the hypothesis that the pictures' sorting criterion might be wrong. He translated this idea to action immediately (minute #12):

M1: "Theater address may not be right." [Changes the sorting criterion in Flickr] "Sort by movie names"

By contrast, F4, the least successful participant, often failed to produce actionable ideas from her framing. For example, numerous times she filtered out possible ideas before even trying to follow up on them, such as at minute #4:

F4: "Flickr. Settings." [Looks at getGeoTaggedPhotos which is the default operation and getPhotos. Leaves the default selected.] "Ok I'm doing this wrong…"

F4's low self-efficacy (3.3, vs. an average of 3.48 for females and 3.8 for all) may have hindered the production of such output, perhaps because it turned her focus toward her own capabilities and away from solving the problem itself. When her framing did not produce outputs, F4 had no inputs for the action phase. She flailed, choosing actions to try at random, often repeating ideas that she had already tried multiple times:

F4: …"Didn't work…Click to get mashing ideas" [Reads mashing suggestions:] "GeoNames, Flickr…" [Hovers over Phonebook. Picks GeoNames. Hovering, reads:] "get latitude and longitude" <which is the default> "Oh I keep on doing that."

Why did participants leave the framing phase without output? One common event tied to this phenomenon was *design* barrier instances. Ko et al. characterized design barriers as: "I don't know what I want the computer to do" [16]. Ten out of 16 design barriers were followed by framing episodes with no outputs. Among the framing episodes following the remaining six design barriers, two ended with ideas too vague to act upon, and one resulted in a repeating idea that had failed earlier.

Implications: The close tie between design barriers and unproductive framing suggests that end-user mashup environments can improve end-user programmers' framing efforts by suggesting ways to overcome design barriers. For example, F4 specifically sought ideas from the environment (e.g., see above quote), but she was unable to find them at the level she sought. Tools to assist end-user mashup programmers to refine their understanding of the problem and possible solution ideas could help prevent end-user programmers from coming away from their framing efforts empty-handed. According to Schön, experts own repertoires of past approaches. They bring these repertoires to a new situation by "imposing" a previously useful frame on it, testing the fit by seeing if their actions in the new situation contradict the reused frame. Thus, one promising avenue to assist end users in framing would be providing users with examples serving as such a repertoire. Examples are common in end-user programming environments, and were available in Popfly—but when examples were available in this environment, attempts to learn from examples failed (17 out of 21 among all participants). The problem was that participants were unable to find the *right* examples or to distill useful information from them. This suggests the need for better support for helping users find the examples they need to address the problems they are having. Work such as [14] may inform better design of tools to support utilizing examples.

If an idea fails, reframe and get a new one

Using the ideations codes, we noticed that some participants shared the same ideas but the degrees to which they were attached to those ideas varied greatly. Some participants refused to discard unworkable ideas, and we viewed that as *inflexibility*. As mentioned before, flexibility, the ability to produce a variety of ideas, is critical to creative output. One way to achieve this is through what Schön called *reframing*; that is to change one's definition of the problem to approach it from a different angle, which allows for the discovery of very different solution ideas.

F4 was an example of inflexibility in her refusal to reframe. She had the idea in minute #9 that she needed a map when in fact using a map was not a viable solution to the task. Other than a brief detour at minute #40, she stayed with that idea throughout the session, trying to get movie information and pictures to show up on a map. When her idea failed, instead of reframing or looking for other alternatives, F4 turned to a "get mashing ideas" tool in Popfly that lists blocks that could communicate with blocks already in the workspace. This produced actionable ideas (the suggested blocks), but these ideas came from the environment, not from her head. There was no evidence that she reflected on what had gone wrong with her previous attempts, nor attempted in reframing to rethink the problem. Instead, she simply repeated actions she had tried before, with no progress in the mashup itself or in evolving her understanding of the problem or potential solutions.

On the contrary, flexibility in reframing did not seem to be difficult for the more successful participants; they seemed to recognize the time to abandon nonproductive ideas. M1 is an interesting contrast to F4, because he started with exactly the same idea as F4, i.e., that pictures needed to be

placed on a map, and then tried to use exactly the same blocks as F4, which occurred in minute #6.

But unlike F4, when he did not succeed with that idea, he abandoned it. After only two attempts to get pictures to show up on a map failed, he reentered the framing phase to look for other possible approaches, at which point he came across the Local Information tab that led to Local Movies. By minute #12, he had already taken that idea into the acting phase to pursue the Local Movies idea.

The uncertainty in the reframing stage of reflection-in-action was difficult for even the most successful participant. For example, F1 said, "I don't know what I'm doing". It especially challenges people with low self-efficacy like F4, because according to self-efficacy theory, low self-efficacy often leads to low flexibility [1]. Like F4, low-self-efficacy people may attribute failures to their own lack of abilities, thereby pursuing poor ideas too long.

Implications: "What-if" features might help with inflexibility and unwillingness to reframe for low self-efficacy users. For example, a tool that would allow users to make assumptions about what a block will output might enable users to explore assumptions in multiple ways. One way might be to focus on testing the assumption. A second way might be to focus on companion blocks compatible with that assumption. A third way might be to focus on competing blocks supporting the same assumption.

Acting upon Ideas

Acting upon ideas can be regarded as "just" implementation. Even so, its interwoven relationship with design decisions sheds insights on the way ideas progressed in our participants' mashups as well as obstacles to such progression.

In transferring ideas to action, one obvious reason our participants took these actions was to follow up on ideas or hypotheses generated in the framing stage. A second reason was to produce a specific outcome as distinguished from those generated by hypotheses or a goal of exploration. A third reason was exploratory— participants acted to explore and to see what would happen in order to understand the situation better. These goals for acting are consistent with the reflection-in-action theory [26], but in addition we identified issues with the current support of acting in the environment, namely the lack of support for tinkering, elaboration, and parallel explorations of ideas.

Explore and tinker… not so effectively

Schön characterizes exploratory actions as "probing, playful activity by which we get a feel for things" [26]. Research in education [24] and end-user programming [4] have pointed to the benefits of tinkering. We took note of tinkering behaviors by looking at the output from framing and the types of ideations people had. We noticed that participants sometimes left the framing phase without concrete ideas to act on. In those occasions, participants tinkered, generating fodder for reflection, and sometimes new ideas or hypotheses that might lead to later developments. For example, M3 tinkered with the options of the Local Movies block without prior expectations as to what the changes would bring about. By tinkering he discovered the usefulness of the Local Movies block, i.e., the ability to deliver theater information, so he retained the block in his mashup and built other ideas around that.

> *M3: "I'm just trying to figure out how to get the program to run to show movies around CITY but I can't figure it out... I'll just keep clicking around 'till I get it… Try a different operation to see if it works… So far I've found out the theaters within CITY…"*

M2 and M4 tinkered excessively with blocks' connections, reflected in part by the large number of *which blocks to connect* ideas (see the dark gray shade in Figure 3; M2 and M4's portions are surrounded by rectangles). Although tinkering sometimes led to useful outcomes, more often it did not:

> *M2: [Links Local Movies to Yahoo Images and MSN News, which feed to Block Inspector. Runs. Nothing shows] "So, let's try all these in series."[Does that. Runs. Nothing shows] "Nope. So, I was on the right track before."*

As these examples illustrate, tinkering did not consistently occur with reflection, impeding participants' understanding of the design options available to them in the form of blocks supplied by the environment or how blocks may work together. Partially to blame is the cost of carrying action to reflection in the environment. Popfly's runtime view (Figure 1 bottom), the primary facilitator for reflection, is separate from the implementation interface (Figure 1 top and middle). This separation made it difficult for participants to cross-reference mashups with their output. Moreover, participants were only allowed to view the runtime results for the entire mashup as opposed to those originating from tinkering with a portion of the mashup. Because of these attributes, reflection tended to slow down ability to act.

Implications: Literature in end-user programming has shown that tinkering with reflection can be helpful, but tinkering *without* reflection has been associated with negative outcomes [4]. In our environment, a barrier to carrying action to reflection was the cost of running. On the other hand, previous tinkering research has shown that when the cost of running can also be too low, encouraging some end

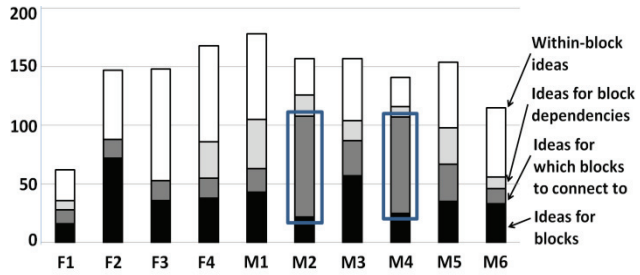

Figure 3. Ideations counts: number of all types of ideations.

users (usually males) to tinker without bothering to reflect [4]. Thus, in order to encourage tinkering productively, the cost of crossing the bridge from action to reflection needs to be carefully considered, so that it is neither too high nor too low. Grigoreanu et al. has shown that it is possible to influence tinkering behaviors through feature design [9].

Elaborate, but with moderation

Elaboration is an important component of creativity. By analyzing the ideations and barriers codes, we discovered both good and poor elaboration behaviors. For example, F1 was successful at elaborating her ideas systematically. This was depicted by her organized engagement with all types of ideas. Her ideation processes often followed a pattern: picking a candidate block (*ideas for blocks*), examining its options (*within block ideas*), connecting it to other blocks (*ideas for which blocks to connect*), and adjusting settings of blocks to account for the inclusion of the new block (*ideas for block dependencies*). As a result of adequate elaboration, she was able to distinguish good ideas from poor, and act effectively toward solving the problem.

In contrast, elaboration was problematic for most participants. Two major issues were: lack of elaboration and excessive elaboration. Lack of elaboration was pinpointed by excessive addition and removal of blocks in the workspace. Two participants, F2 and M3, were particularly affected by this problem (see black in Figure 3 for F2 and M3). M3 encountered multiple *selection* barriers, which refer to difficulties in not knowing what block to use for a desired behavior. Because of this, he excessively added and removed blocks leading to a failure to elaborate on potentially successful ideas based on those blocks.

> *M3: "So I don't really know what blocks to use… CinemaTopTen came up before - I don't know if it's useful or what it does but I'll try it… There's nothing… I'll see what Cigarettes is 'cause that seems interesting…"*

Some participants demonstrated the opposite behavior, attempting to elaborate excessively but failed to gain benefits from doing so. In particular, F4 encountered *use* barriers (not knowing how to use a block) and *coordination* barriers (making blocks to work together) in trying to refine existing ideas, but regardless of her difficulties, she persisted. These difficulties prevented her from being able to elaborate effectively on her ideas and in turn led to an *understanding* barrier, i.e., not knowing why the program behaved the way it did. The following example shows that in the face of a use barrier, she randomly fiddled with the block's settings, and thus failed to elaborate effectively.

> *F4: "Do I need to change the source for all of these <parameters for operation>?" [Sighs. Changes the settings back and forth. Runs] "Why doesn't it show? I don't know what I'm doing wrong." [Keeps on trying without success]*

Implications: Both phenomena of under- and over-elaboration highlight mashup environments' lack of support for various levels of design from the abstract to the concrete. We suggest that under-elaboration lies in the possibility that participants may have perceived elaboration as more costly than simply choosing another idea. The reason might be that the environment encouraged detailed implementation too early. Similarly, we argue that over-elaboration is linked to the same issue with the environment. For example, in order to test a tentative idea that a block might be useful, rather than being able to make the high-level assumption that it is useful and proceed with the rest of the design, the user had to go *all the way,* specifying various settings for the block and integrating it with the rest of the mashup in order to test that idea. In cases where an environment solely provides detailed implementation mechanisms, there is a risk of users to be lured into "trying to make it better" rather than thinking about the bigger picture. This phenomenon is particularly important in design [27] and has led to an important body of research on supporting sketching in computerized design practice (e.g., [7]). Similar effort has been made to support "sketching phases" in user interface development (e.g., [17]). However, these systems have been targeted to professional designers rather than casual end-user programmers.

Backtracking: explore ideas in parallel and what else?

Backtracking refers to instances in which participants returned to a previous state of the mashup after exploring other ideas. We identified these instances by diagramming ideations in the workspace in a time-wise fashion. *All* of our participants backtracked *multiple* times. There were three types of backtracking: to pursue alternative ideas, to revert back to a more successful state, and accidentally re-entering a previous state.

First, some participants were trying to experiment with multiple idea alternatives, but as with almost all programming environments, there was no support for this. For example, within only three minutes, M3 backtracked to the same state three times (Figure 4). With each trial, he experimented with a block for retrieving/displaying pictures that he hadn't used before. M3's experimentation would have been less time-consuming if he could have done his experiments in parallel and compared results side by side.

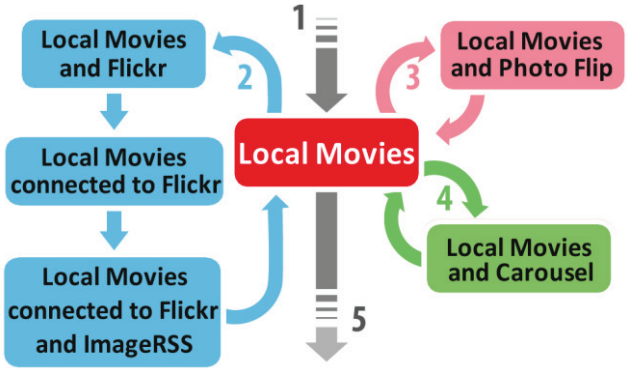

Figure 4. M3's backtracking over three minutes. Each box represents a different state of his mashup. The numbers show the order in which M3 (re-)entered and exited LocalMovies.

The second way participants used backtracking was to retreat from a path, getting back to a "safe" state. Once back in a working state, the participant usually went back to framing, to think of other ideas to try. But this tended to be error-prone, because sometimes participants had trouble recalling the exact details of that state.

M2: "So how was this working before?"

M5: "I'm gonna save more often now, like if I screwed up I could still get something to come up...."

The third way participants backtracked was by accident, trying to find their way without meaning to return to previous states. F4 did this. Although she did not intend to backtrack, she sometimes recognized a state when she stumbled into it again.

F4: [Adds GeoNames. Hovers over it. Reads description] "Get latitude and longitude. Oh I keep on doing that…"

Implications: Similarly to designers [19], end users would benefit from the ability to explore ideas in parallel. Systems featuring this capability have been implemented for professionals, e.g., [12, 29]. However, such systems remain largely absent from end-user programming, with a notable exception in [12]. Additionally, programming environments should support end users' need to return to an earlier salient step in their design, for instance by permitting them to bookmark their exploration. Our participants relied on their memory to do this, which was error-prone. Finally, since backtracking is detectable, it might be possible to gently map the user's journey through state space, to avoid the wasted effort of returning to a state multiple times by accident.

Reflecting upon Acting

Analyzing participants' reflection phases, their actions in between reflections, and *understanding* barriers they had, we identified two salient issues with the support of reflection in Popfly. First, some participants carried out a large number of actions before they reflected and thus missed out on the opportunity to identify the impact of each action. Second, once the mashups' results were shown, participants lost the ability to refer back to the program's logic (as the Edit interface was separate from the Run view), and hence they could not efficiently debug. Not surprisingly, nearly all participants experienced understanding barriers, i.e., not knowing why the program did what it did.

Actions were reflected by ideations in the workspace. For example, having an *idea for block* meant adding a block to the mashup. Thus, upon visually exploring the occurrences of the ideations codes, we noticed that F4 underwent many actions before reflecting on them. For each participant, we then calculated the number of actions carried before an evaluation of the mashup, i.e., running it. Figure 5 provides a visualization of the number of ongoing actions between runs. F4 clearly stands out, as she made many changes (46 actions) to her mashup before the first run which happened at minute #26. To a lesser extent, F2 and M5 demonstrated a similar pattern (minute #33 and minute #13 respectively). These participants had difficulty evaluating *which* action caused the changes in results. M5 eventually recognized that this strategy was not working for him, realizing that his program had become so complex, he could not debug it. At that point, he started removing blocks, which rapidly turned his progress around.

M5: "Simplicity" [Runs. Theater and movie info shows up.] "Oh, ok. There we go. I was getting way too complicated." "It works well to run the program at each step."

In contrast, the most successful participant, F1, reflected upon her actions frequently. In this example, she did only two actions (adding a block and selecting an operation for it) before she reflected by running her mashup:

F1: "Ooh, Local Movies" [Adds it. Looks into block's settings] "getTheaters <AndMovies>"<default operation> "So I just hit run" [looks at results] "Theaters. Ok so I have a really long list of movies. And show times."

Additionally, we noticed that five participants had on occasions no actions carried out between runs. One possible reason was that the environment separated the mashup's output from its logic so when running the mashup, its logic was no longer available for reference. Thus, participants had to memorize one screen before switching to the other. In fact, three participants took notes on the outputs before going back to editing the mashups. Memorization is taxing, and the cost of running could also have deterred participants from frequent runs to enable reflection.

Implications: These phenomena suggest two implications. First, there is a need for mashup environments to not only reduce effort of running per se but also the effort of marrying the runtime output with the program's logic itself. The environment could for instance provide micro-evaluations of local portions of the mashup during the implementation phase (e.g., what is the output of this particular block if I set it up like this). Moreover, the environment could provide references between the outcomes of the mashup and its log-

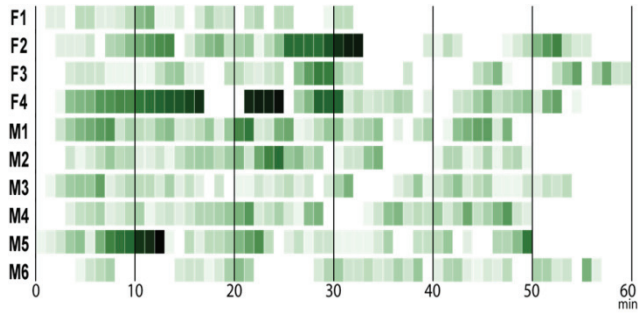

Figure 5. Accumulation of idea actions (expansions and contractions) before run. Color gets darker as idea actions accumulate. Color is reset to very light whenever participant hits "Run". (White spaces are pauses in which no actions take place.) For example, F4 accumulated 46 idea expansions and contractions before she first hit "Run" in minute #26.

ic, by highlighting relationships between them. However, solutions to these problems are inherently difficult for mashups, since their performances also rely on remote providers of information that may not always be responsive.

Second, while professional programmers and even novice computer science students get a lot of practice honing their problem-solving strategies for debugging such as isolating variables, end-user programmers may not have developed debugging strategies like these. Tools for debugging by end-user programmers could provide hints for debugging strategies, as demonstrated in spreadsheet software [9].

PERSPECTIVES ON THE DESIGN LENS

Consideration of programming through a design lens provided unique insights into the process of programming by these end users. Traditionally, empirical studies of programmers generally apply theories from psychology, use a bottom-up grounded theory approach, or test hypotheses about behavior. Further, that literature considers design and programming to be two different, albeit sometimes highly iterative, phases. Our approach, in contrast, implies that *every choice* our participants made, large or small, could be viewed through a design lens. Doing so amounts to considering programming as the paths of ideas from their beginning to their end.

We primarily used the reflection-in-action framework as a tool for our analysis, deriving it into a code set sufficiently robust for our purpose. The theories from design, creativity, and programming literature contributed to the code set as well, and moreover helped us to pinpoint, explain, and interpret the patterns we found. We developed two useful tactics for understanding paths of ideas. The first was the use of visualization tools to view patterns of the interactions between code sets at multiple levels of abstraction. The second was the use of triangulation, arriving at the same answer from more than one perspective, i.e., analyzing data using design, creativity, and end-user programming perspectives concurrently, merging and comparing their output. These two aspects were intertwined.

Regarding visualizations, we found two types to be particularly helpful in understanding how ideas evolved. We call the first type "idea graphs". These were graphs representing the *state* of the participants' ideas that they elaborated upon and retracted explicitly in the workspace. Figure 4 is an abstract view of one of these; the detailed version we used depicted every major state transition of a participant's "idea set" with all arrows labeled with actions triggering the transitions.

The second main visualization type represented the coded data over time. For this type of analysis, we found that the ability of the visualization software to quickly create combinations of codes into a new "supercode" to explore emergent patterns was crucial. For example, we supplemented Figure 5 with annotations denoting interesting events from other code sets, which particularly highlighted issues with elaboration. Exploring the co-occurrences of codes from different code sets allowed for the exploration of the data from different perspectives simultaneously (e.g., using design, creativity codes, and barrier codes). In doing so, we effectively conducted a triangulation process by analyzing our data from different angles.

The design lens proved useful to us at getting a perspective on end-user programmers' ideation processes. Ideally, we suggest that this method should be combined with more traditional ways to study end-user programming, allowing a triangulation process involving two major perspectives: traditional approaches of understanding programmers, together with the emphasis on ideations from design and creativity literature.

CONCLUSION

In this paper, we have presented a design theory-based approach to investigating programming by end users. We demonstrated the usefulness of this approach by applying the reflection-in-action design model and the ideation notion from creativity literature to the think-aloud protocols from ten participants creating web mashups. The results revealed ample opportunities for environments to better support end-user programming as a design activity.

Therefore, our work makes three contributions: 1) the methodology of applying a theory-based design perspective to programming, 2) evidence of the usefulness of using this approach through insights gained, and 3) the insights themselves into end-user programmers' problem-solving attempts, with implications for design of end-user programming environments for mashups. Implications included support for meaningful tinkering, for effective reflection, and for exploration of multiple design alternatives in parallel in end-user programming environments.

Using design theory as a perspective on end-user programming thus shows promise in helping researchers to better understand the problems faced by end-user programmers, aiming toward future environments that can avoid the kinds of problems encountered by some of our participants.

F4: "This is so hard for me. Why is it so difficult?"

ACKNOWLEDGMENTS

This work was supported in part by NSF grants 0917366, 0325273 and 0324844.

REFERENCES

1. Bandura, A. Self-efficacy: Toward a unifying theory behavioral change. *Psychological Review 8*, 2 (1977), 191-215.
2. Bayazit N. Investigating design: A review of forty years of design research. *Design Issues 20*, 1 (2004), 16-29.
3. Beckwith, L. Burnett, M., Wiedenbeck, S., Cook, C., Sorte, S., and Hastings, M. Effectiveness of end-user debugging software features: Are there gender issues? In *Proc. CHI 2005,* ACM Press (2005), 869-878.

4. Beckwith, L., Burnett, M., Grigoreanu, V., and Wiedenbeck, S. Gender HCI: What about the software? *IEEE Computer 39*, 11 (2006), 83-87.
5. Boden, M. *The Dimensions of Creativity*. MIT Press Cambridge, London, England, 1994.
6. Compeau, D. and Higgins, C. Computer self-efficacy: Development of a measure and initial test. *MIS Quarterly* 19, 2 (1995), 189-211.
7. Do, E. and Gross, M. Inferring design intentions from designers' sketches – An investigation of freehand drawing conventions in design, In *Proc. CAADRIA'97* (1997).
8. Gray, W. D. and Anderson, J. R. Change-episodes in coding: When and how do programmers change their code? *Second Workshop on Empirical Studies of Programmers*, Ablex Publishing Corp., Norwood, NJ, 1987.
9. Grigoreanu, V., Cao, J., Kulesza, T., Bogart, C., Rector, K., Burnett, M., and Wiedenbeck, S., Can feature design reduce the gender gap in end-user software development environments? In *Proc. VL/HCC 2008,* IEEE (2008), 149-156.
10. Guilford, J. P. *Intelligence, Creativity and Their Educational Implications.* Robert R. Knapp, San Diego, CA, 1968.
11. Guindon, R. Designing the design process: Exploiting opportunistic thoughts. *Human-Computer Interaction 5*, 2 (1990), 305-344.
12. Hartmann, B., Yu, L., Allison, A., Yang, Y., and Klemmer, S. Design as exploration: Creating interface alternatives through parallel authoring and runtime tuning. In *Proc. UIST 2008*, ACM Press (2008), 91-100.
13. Jeffries, R., Turner, A. A., Polson, P. G., and Atwood, M. E. The processes involved in designing software. In *Cognitive Skills and their Acquisition*, Anderson, J. R. (Ed.), Lawrence Erlbaum Associates, Hillsdale, NJ, 1981, 255-283.
14. Herring, S., Chang, C., Krantzler, J., and Bailey, B. P., Getting inspired! Understanding how and why examples are used in creative design practice. In *Proc. CHI 2009*, ACM Press (2009), 87-96.
15. Kannengiesser, U. and Zhu, L. An ontologically-based evaluation of software design methods, *The Knowledge Engineering Review 24,* 1 (2009), 41-58.
16. Ko, A. J., Myers, B., and Aung, H. Six learning barriers in end-user programming systems. In *Proc. VL/HCC 2004*, IEEE Computer Society (2004), 199-206.
17. Landay, J. A. and Myers, B. A., Sketching interfaces: Toward more human interface design, *IEEE Computer 34*, 3(2001), 56-64.
18. Milgram, R. M. Creativity: An idea whose time has come and gone*,* in *Theories of Creativity*. Mark, R. A. and Robert, A. S. (Ed.). Sage Publications, London, UK, 1990.
19. Myers, B., Park, S. Y., Nakano, Y., Mueller, and G., Ko, A. How designers design and program interactive behaviors. In *Proc. VL/HCC 2008,* IEEE (2008), 177-184.
20. Nardi, B. *A Small Matter of Programming: Perspectives on End-User Computing*. MIT Press, Cambridge, MA, 1993.
21. Rist, R. S. Knowledge creation and retrieval in program design: A comparison of novice and intermediate student programmers. *Human-Computer Interaction 6*, 1 (1991), 1-46.
22. Rode, J. and Rosson M. B. Programming at runtime: Requirements and paradigms for nonprogrammers' web application development. In *Proc. VL/HCC 2003*, IEEE (2003), 23-30.
23. Rosson, M. B., Sinha, H., Bhattacharya, M., Zhao, D. Design planning in end-user web development, In *Proc. VL/HCC*, IEEE (2007).
24. Rowe, M. B. *Teaching Science as Continuous Inquiry* (2nd. Ed.), McGraw-Hill, New York, NY, 1978.
25. Runco, M. A. *Creativity Theories and Themes: Research, Development, and Practice.* Elsevier Academic Press, Burlington, MA, 2007.
26. Schön, D. A., *The Reflective Practitioner: How Professionals Think in Action*. Basic Books, New York, NY, 1983.
27. Simpson, M. and Viller, S. Observing architectural design: Improving the development of collaborative design environments. In *Proc. CDDVE'04* (2004).
28. Sommerville, I. *Software Engineering* (8th Ed.), Pearson Education Limited, Harlow, England 2006.
29. Terry, M., Mynatt, E., Nakakoji, K., and Yamamoto, Y. Variation in element and action: Supporting simultaneous development of alternative solutions. In *Proc. CHI 2004*, ACM Press (2004), 711-718.
30. Visser, W. Organization of design activities: Opportunistic, with hierarchical episodes. *Interacting with Computers 6*, 3 (1994), 239-274.
31. Visser, W. Designing as construction of representations: a dynamic viewpoint in cognitive design research. *Human-Computer Interaction 21*, 1 (2006), 103-152
32. Wong, J. and Hong, J. I. Making mashups with marmite: Towards end-user programming for the web. In *Proc. CHI 2007,* ACM Press (2007), 1435-1444.