In Proceedings of the Second International Symposium on End User Development, March 2-4, 2009, Siegen, Germany (to appear).

Males' and Females' Script Debugging Strategies

Valentina Grigoreanu[1,2], James Brundage[2], Eric Bahna[2], Margaret Burnett[1], Paul ElRif[2], Jeffrey Snover[2]

[1] Oregon State University, School of Electrical Engineering and Computer Science, Corvallis, Oregon, USA 97331
{grigorev, burnett}@eecs.oregonstate.edu

[2] Microsoft, One Microsoft Way, Redmond, Washington, USA

{t-valeng, jamesbru, ebahna, pelrif, jsnover}@microsoft.com

Abstract. Little research has addressed IT professionals' script debugging strategies, or considered whether there may be gender differences in these strategies. What strategies do male and female scripters use and what kinds of mechanisms do they employ to successfully fix bugs? Also, are scripters' debugging strategies similar to or different from those of spreadsheet debuggers? Without the answers to these questions, tool designers do not have a target to aim at for supporting how male and female scripters want to go about debugging. We conducted a think-aloud study to bridge this gap. Our results include (1) a generalized understanding of debugging strategies used by spreadsheet users and scripters, (2) identification of the multiple mechanisms scripters employed to carry out the strategies, and (3) detailed examples of how these debugging strategies were employed by males and females to successfully fix bugs.

Keywords: Gender, Debugging, Scripting, Debugging Strategies.

1 Introduction

At the border between the population of professional developers and the population of end-user programmers, lies a subpopulation of IT professionals who maintain computers, and they accomplish much of their job through scripting. As Kandogan et al. argue, this population has much in common with end-user programmers [14]: as in Nardi's definition of end-user programmers, they program as a means to accomplish some other task, not as an end in itself [22]. Scripting is also becoming much more common by end-user programmers themselves through the advent of end-user oriented scripting languages for the desktop and the web. However, despite the complexity of some scripting tasks, little attention has been given to scripters' specific debugging needs, and even less to the impact that gender differences might have on script debugging strategies and the mechanics used to support them.

We therefore conducted a qualitative study to address this gap by identifying the debugging strategies and mechanisms scripters used. *Strategy* refers to a reasoned plan or method for achieving a specific goal. *Mechanisms* are the low-level tactics used to support those strategies: through environment and feature usage. Our work was guided by the debugging strategies reported in an earlier end-user debugging study with spreadsheet users [32].

There are several reasons to ask whether strategies used by scripters working with a scripting environment might be different from strategies used by end-user programmers with a spreadsheet system. First, the populations are different; for example, one might expect scripters to have more experience in debugging per se than spreadsheet users. Second, the language paradigms are different: scripting languages are control-flow oriented, in which programmers focus primarily on specifying sequence and state changes, whereas spreadsheet languages are dataflow oriented, in which programmers focus primarily on specifying calculations (formulas) that use existing values in cells to produce new values. The language paradigm differences lead naturally to a third difference: the environments' debugging affordances themselves are different, with scripting environments tending toward peering into sequence and state, whereas spreadsheets' affordances tend more toward monitoring values and how they flow through calculations. Therefore, the research questions we investigated were:

RQ1: What debugging strategies do scripters try to use?

RQ2: What mechanisms do scripters employ to carry out each strategy?

RQ3: How do our findings on scripters' strategies relate to earlier results on strategies tied with male and female spreadsheet users' success?

Thus, the contributions of this paper are in (1) identifying the strategies scripters try to use in this programming paradigm, (2) identifying the mechanisms scripters use to carry out their different strategies, and (3) exploring details of successful uses of the strategies by males and females.

2 Background and Related Work

Although there has been work in how to effectively support system administrators in *creating* their scripts [14], we have been unable to find work addressing scripters' debugging *strategies*. Instead, most of the work on script debugging has been on tools to automatically find and fix errors (e.g., [35, 37]).

However, there has been considerable work on professional programmers' debugging strategies, and some work on end-user debugging strategies. One study on professional programmers' debugging strategies classified debugging strategies as forward reasoning, going from the code forward to the output, and backward reasoning, going from the output backward through the code [15]. See Romero et al. for a survey of professional programmers' debugging strategies [29].

End-user programmers have elements in common with novice programmers, so the literature on how novice programmers differ from experts is relevant here. For both novices and experts, getting an understanding of the high-level program structure before jumping in to make changes relates to success [19, 21]. However, experts have been found to read programs differently from novices: reading them in control flow order (following the program's execution), rather than spatial order (top to bottom).

In end-user programming, gender differences have been found in attitudes toward and usage of end-user programming and end-user programming environment features [3, 4, 5, 10, 13, 16, 27, 28, 30, 38]. Especially pertinent is a series of end-user debugging studies reporting gender differences in debugging strategies for spreadsheets [9, 32]. The first of these studies pointed to behavior differences that suggested strategy differences, and the second reported a set of eight strategies end users employed in their debugging efforts. In both of these studies, the strategies and behaviors leading to male success were different from those leading to female success. For example, in [32], dataflow strategies played an important role in males' success, but not females'. Prabhakararao et al.'s spreadsheet debugging study with end users also reported a strong tie between using a dataflow strategy and success [25], but participant gender was not collected in that study.

In fact, gender differences that relate to processing information and solving problems have been reported in several fields. One of the most pertinent works is the research on the Selectivity Hypothesis [20, 23]. It proposes that females process information in a comprehensive way (e.g., attending to details and looking for multiple cues) in both simple and complex tasks. Males, on the other hand, process information through simple heuristics (e.g., following the first cue encountered), only switching to comprehensive reasoning for complex tasks.

Self-efficacy theory may also affect the strategies employed by male and female debuggers [1]. Self-efficacy is a person's confidence about succeeding at a specific task. It has shown to influence everything from the use of cognitive strategies, to the amount of effort put forth, the level of persistence, the coping strategies adopted in the face of obstacles, and the final performance outcome. Regarding software usage, there is specific evidence that low self-efficacy impacts attitudes toward software [3, 11], that females have lower self-efficacy than males at their ability to succeed at tasks such as file manipulation and software management tasks [33], and that these differences can affect females' success [3].

Gender differences in strategies also exist in other problem-solving domains, such as psychology, spatial navigation, education, and economics (e.g., [8]). One goal of this paper is to add to the literature on gender differences in problem solving relating to software development, by considering in detail the usage of different strategies by male and female scripters.

3 Study

3.1 Participants

Eleven IT professionals (eight males and three females) volunteered to participate in the study by responding to invitations on an IT-related internet forum and on a PowerShell email discussion list. Participants received software as gratuity. Although we had hoped for equal participation by females and males, female IT professionals are in short supply, and only three signed up. Almost all participants had a technical college degree (in computer science, engineering, or information systems), with the exception of two males whose education ended after high school. Despite their technical degrees, six of the eleven participants rejected the label of "software developer." Those who did classify themselves as software developers were the three females and two of the eight males; the remaining six males described themselves as IT professionals or scripters. All participants reported that, in their everyday jobs, they accomplished their IT professional tasks using PowerShell. As examples of these regular tasks, participants mentioned moving packages, moving machines out of a domain, modifying the registry, initializing software, automating IT tasks, automating tests, and creating test users on servers.

All participants had written two or more scripts within the past year, using Windows PowerShell. The females had written fewer scripts in the past year than the males (number of scripts written by females: 2, 3, and 5; number of scripts written by males: 6, 6, 7, 20+, 30, 30+, 50, 100+).

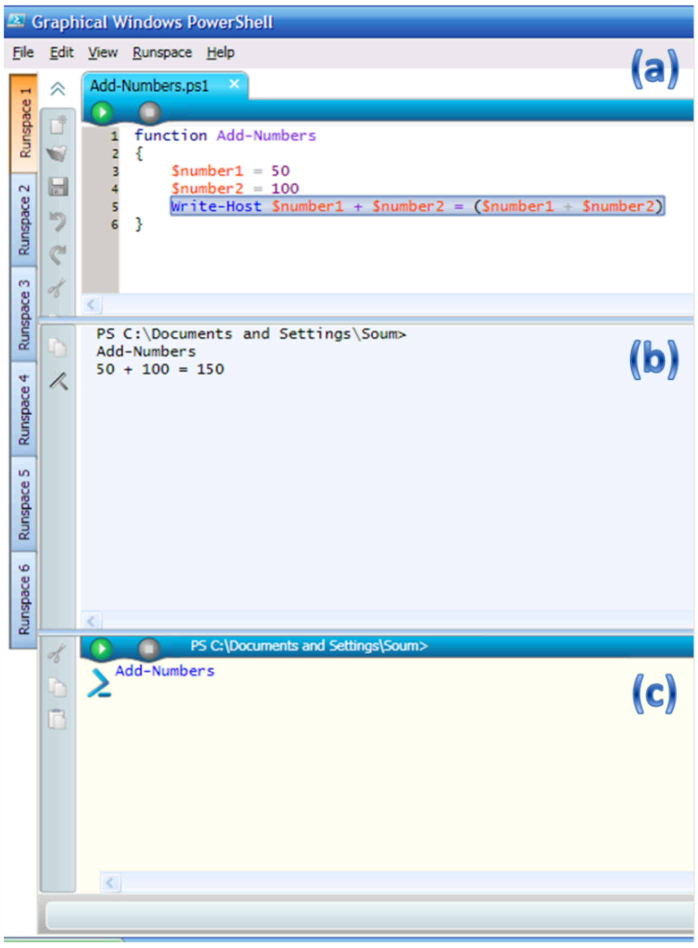
3.2 Scripting Language and Environment

PowerShell is a new implementation of the traditional Command Line Interface and scripting language developed by Microsoft, which aims to support both IT professionals' and developers' automation needs. We used an as-yet unreleased version of this language and environment in our study. PowerShell supports imperative, pipeline, object-oriented, and functional constructs. Its pipelining, unlike traditional UNIX commands that pipeline *text* to one another, pipelines *objects* to one another. PowerShell has both a command line shell and a graphical scripting environment, and participants used both. Both the command line and graphical scripting environment provide common debugging features such as breakpoints, the ability to step into, error messages, and viewing the call stack. Fig. 1 shows a version of the graphical scripting environment that is similar to the one our participants used.

3.3 Tasks, Procedure, and Data Collection

We instructed the participants to debug two versions of a PowerShell script, which included a "main" section and eight called functions, each of which was in a separate file within the same directory. We used the same script (two versions) for both tasks in order to minimize the amount of time participants spent getting an understanding of the scripts so as to maximize the amount of time they spent actually debugging.

The script was a real-world script that one of us (Brundage) had previously written to collect and display meta-data from other PowerShell scripts. We introduced a total of seven bugs, which we harvested from bugs made by the script's author when he originally wrote the script. Each version of the main script contained one different bug. The eight functions called by both versions contained five other bugs. The seven total bugs fell into two categories: three errors *in data*: using an incorrect property, allowing the wrong kind of file as input, and omitting a filter; and four *errors in structure*: an assignment rather than a comparison, an off-by-one error, an infinite loop, and omitting the code that should have handled the last file.



```
function Add-Numbers
{
    $number1 = 50
    $number2 = 100
    Write-Host $number1 + $number2 = ($number1 + $number2)
}
```

```
PS C:\Documents and Settings\Soum>
Add-Numbers
50 + 100 = 150
```

```
PS C:\Documents and Settings\Soum>
Add-Numbers
```

Fig. 1. This is the graphical version of the Windows PowerShell environment. (a) Scripts are written in the top pane, and the example shows a function that adds two numbers. (b) The output pane displays the result of running the script or command. (c) The command line pane is used exactly like PowerShell's command line interface. In this case, it is running the function to add two numbers. This figure is an adaptation of a figure in [36].

After participants completed a profiling survey, we gave them a description of what the script was supposed to do. Participants then debugged one version of the script using a command line debugger and a second version using a graphical debugger. The order of the script versions and environments was randomized to control for learning effects.

Participants were instructed to talk aloud as they performed their debugging tasks. Data collected included screen captures, video, voice, and measures of satisfaction. After completion of each task, participants were given a post-session questionnaire that included an interview question about the debugging strategies they used.

3.4 Analysis Methodology

We analyzed the data using qualitative content analysis methods. We analyzed two data sources: participant responses to the questionnaire, and the videos. Because we wanted to measure the extent to which males' and females' debugging strategies identified in previous work [32] would generalize to our domain and population, we began by mapping that code set to the Powershell domain of the current study.

One researcher applied these codes to participants' post-session open-ended interview responses about strategies. We asked the same strategies question as in the earlier work mentioned above, but it was asked verbally, rather than on paper. This led to generalizations of a few of the codes and the introduction of a few new codes.

Two researchers then used this revised code set as a starting code set for the videos. They independently coded 27% of the videos (six tasks from various participants), achieving 88% agreement. The dually coded set included the three participants described in 4.2, for which the two researchers also collaboratively analyzed the circumstances, sequences, and mechanisms, resolving any disagreements as they arose. This also resulted in our final code set, which is given in Section 4. The first researcher then analyzed and coded the remaining videos alone.

4 Results

4.1 Scripters' Debugging Strategies

Our scripters used variants of seven of the eight strategies the spreadsheet users used [32], plus three others. The spreadsheet strategy termed "fixing formulas" (in which participants wrote only of editing formulas, but not of how they figured out what, where, or how to fix the bug or of validating their changes) was not found in our data.

As Table 1 shows, five of the strategies they used were direct matches to the earlier spreadsheet users' strategies, two were matches to generalized forms of the spreadsheet users' strategies, and three arose that had not been viable for the spreadsheet users. However, even for the direct matches, the mechanisms scripters used to pursue these strategies had differences from those of the spreadsheet users.

Direct Matches. *Testing* is trying out different values to evaluate the resulting values. Some of the mechanisms used by these scripters would not traditionally have been identified as testing, yet they clearly are checking the values output for correctness—but at finer levels of granularity than has been possible in classic software engineering treatments of the notion of testing.

Specifically, we noticed three types of testing mechanisms used by participants: testing different situations from a whole-program perspective, incrementally checking variable values, and incrementally testing in other ways. The first type is classic testing methodology to cover the specifications or to cover different parts of the code (testing both the antecedent and the consequent parts of an if-then statement, for example). The latter two are informal testing methods to see whether, after having

Table 1. Participants' responses when asked post-session to describe their strategies in finding and fixing the bugs (listed in the order discussed in the text).

Strategy	Definition
Direct Matches	
Testing	Trying out different values to evaluate the resulting values.
Code Inspection	Examining code to determine its correctness.
Specification Checking	Comparing descriptions of what the script should do with the script's code.
Dataflow	Following data dependencies.
Spatial	Following the spatial layout of the code.
Generalized Matches	
Feedback Following	Using system-generated feedback to guide their efforts.
To-Do Listing	Indicating explicitly the suspiciousness of code (or lack of suspiciousness).
New Strategies	
Control Flow	Following the flow of control (the sequence in which instructions are executed).
Help	Getting help from people or resources.
Proceed as in Prior Experience	Recognizing a situation (correctly or not) as one experienced before, and using that prior experience as a blueprint of next steps to take.

executed part of the code, the variables displayed reasonable values. For example, from a whole-program perspective:

Female P0721081130 ran the code and examined both the error messages and the output text in order, rather than focusing on either one or the other.
Female P0718081400 tried different contexts by cd-ing back to the root directory in the command line before running the file again using the menu.
Male P0717080900 changed the format of the output so that he could understand it more easily when he ran the program.

However, incrementally checking variable values was much more common, and participants did it in many different ways:

Male P0718081030 hovered over variables to check their values.
Male P0117081130 also hovered over, but in conjunction with breakpoints to stop at a particular line to facilitate the hover.
Female P0718081400 ran the code by accessing it through the command line interface. Others preferred reaching it through the menu.
Female P0718081400 typed the variable name in the command line.

Male P0717080900 added temporary print statements to output variable values at that point in time.
Male P0721081330 added temporary print statements to check whether a particular part of the code was reached/covered by the input.
Female P0718081400 would have liked to use a watch window to examine variable values.
Male P0718081030 examined an entire data structure using tabs (auto-complete) to determine the correctness of its property values.

Other forms of incremental testing focused on running part of the code to check its output. For example:

Female P0718081400 did not understand why she was getting an "access denied" message and therefore tried performing the action manually with Windows Explorer and navigating to that directory (to see if she would get the same message in that different context).
Female P0718081400 first ran a variable to see its output, and then started adding the surrounding words to get more information about that variable and how it is being used.
Female P0717081630 used "stepping over" to see the script's output appear incrementally as she passed the line in the code that produced it.
Male P0721080900 wanted to break into the debugger once a variable had a particular value.

Primarily only the first category of testing is supported by tools aiming to support systematic testing for professional programmers or end-user programmers (e.g., WYSIWYT [6]). However, our scripters were very prone to incremental testing, and although the scripting environment gives them good access to checking these values, there is no support in that environment or most others for using this incremental testing to *systematically* track which portions of the code are tested successfully, which have failed, and which have not participated in any tests at all.

Code inspection is examining the code to determine its correctness. Code inspection is a counterpart to testing with complementary strengths [2]. It is heavily relied upon in the open source community [26]. Not surprisingly, as in the spreadsheet study, testing and code inspection were the most common strategies. Participants' mechanisms for code inspection revealed a surprisingly large set of opportunities for supporting code inspection better in scripting environments, spreadsheets, and other end-user programming environments.

Besides simply reading through the code, some of the basic mechanisms the participants used were:

Female P0718081400 opened up all of the files in the same directory as the script (to view functions the main script was calling), and quickly scanned through all of them one after the other.
Male P0718081030 resized the script pane to show more of the script.
Male P0717080900 used the "Find" function to jump to the part of interest.

Male P0718081030 used the command line to find out all the contextual information he could about a variable he was inspecting (its type, for example).

Male P0717080900 used the integrated scripting environment as a code editor for the command line task because he disliked inspecting the code without syntax highlighting.

The above five mechanisms may seem obvious, but many end-user programming environments do not support these functionalities. For example, they are not well supported in spreadsheets; in that environment performing these actions is awkward and modal. Given the heavy reliance on code inspection by the participants in both this study and the previous spreadsheet study, a design implication for end-user programming and scripting environments is to provide support for the flexible and easy ability to inspect large amounts of the code when desired.

Finally, there were many instances of integration between testing and code inspection, such as this participant's fine-grained mixing of the two:

Female P0718081400 hovered over variables in the code view for simultaneously seeing both the code and output values.

Most participants in the earlier spreadsheet study also used testing and code inspection together. The preponderance of mixing these strategies suggests that programming environment designers should strive to support this mixture, allowing "drill down" into related testing information during code inspection, as in the example above, and conversely allowing drill down into related code information during testing. Getting directly to the code that produced certain values is well supported in spreadsheets and in some end-user languages and environments such as KidSim/Cocoa/Stagecast [12] and Whyline [17], but is rarely present in scripting environments.

Specification Checking is somewhat related to code inspection, but involves comparisons: namely, comparing descriptions of what the script should do with the script's code. This strategy is not well supported in any scripting or end-user programming environment—code comments are the primary device to which users in these environments have access for the purpose of specification checking.

Both the spreadsheet study and this one provided (informal) specifications in the form of written descriptions of the intended functionality, and these were widely used by both the previous study's spreadsheet users and the current study's scripters. In addition, they relied on comments and output strings embedded in the code for this purpose, as in the examples below.

Male P0717080900 read the informal description handout related to the script.

Male P0717080900 read the comments in the code related to what that part of the code was supposed to do.

Male P0117081130 looked in the code for the places producing constant string outputs, with the view that those string outputs helped describe what nearby code what supposed to do.

Female P0718081400 read the comments one-by-one, as she was reaching the parts that they referred to in control flow order.

Thus, specification checking is an under-supported strategy for both spreadsheet users and scripters.

Dataflow means following data dependencies through the program. Following dataflow is a natural fit to the dataflow-oriented execution model of spreadsheets, and some spreadsheet tools provide explicit support for it such as dataflow arrows and slicing-based fault localization tools [6]. Even in imperative programs, dataflow mixed with control flow (i.e., "slicing") is commonly used [34], and ever since Weiser's classic study identified slicing as an important strategy for debugging [34], numerous tools have been based on slicing. Our scripters followed dataflow a little, but it was not particularly common, perhaps because the scripting environment did not provide much explicit support for it:

Female P0717081630 said, "Wish I could go to where this variable is declared."
Female P0717081630 tried to "find all references" to a variable, in any file.
Male P0718081030 wanted to know how a particular variable got to be a certain value, and therefore followed the flow of data to see what other variables influenced this variable, and how it got to be the value it was.

Spatial is simply following the code in a particular spatial order. For example, scripts can be read from top to bottom. (This is different from following execution order; execution order deviates from top to bottom at procedure calls, loops, etc.) Most participants demonstrated a little of this strategy, but nobody relied on it for very long. It was fairly uncommon in the spreadsheet study as well, in which fewer than 10% of the participants mentioned that strategy.

Generalized Matches. The two strategies that matched generalizations of strategies observed in the spreadsheet study were Feedback Following and To-Do Listing.

Feedback Following is using system-generated feedback to guide debugging efforts. This is a generalization of the strategy "Color Following" in the spreadsheet study. To draw users' attention to them, the spreadsheet system colored cells' interiors to show their likelihood of containing errors (based on the judgments made by users about the correctness of each cell's value). The users who followed this type of feedback directly were considered to be color following. The scripting environment used certain messages (not colors alone) to draw users' attention to code with possible bugs, a generalization upon following colors toward possible bugs.

Our script participants paid particular attention to the feedback messages, including reading them, navigating backward and forward in them, looking at more or fewer of them, and drilling down to get more information about them. For example:

Male P0117081130 looked at the last error message.
Male P0718080800 changed the display settings so as to show only the first error message.
Male P0717080900 cleared the command line screen so he could easily scroll up and stop at the first error message.
Female P0718081400 resized the output window to see more of the messages at once.

Female P0718081400 opened up Windows Explorer to better understand what path the error message is talking about.

To-Do Listing is indicating explicitly the suspiciousness of code (or lack of suspiciousness) as a way to keep track of which code needs further follow-up. Some spreadsheet users did this by checking cells off or X-ing them out. (These features were designed for another purpose, but some participants repurposed them to keep track of things still to check.) Like the spreadsheet users, our scripters found mechanisms to accomplish to-do listing, such as:

Male P0117081130 put a breakpoint on a line to mark that line as incorrect, and to stop on it whenever he ran the code.
Female P0721081130 closed files that she thought to be error-free, leaving possibly buggy ones open.
Male P0718081030 used pen and paper to keep track of stumbling points.
The same male, P0718081030, also mentioned sometimes using bookmarks to keep track of stumbling points.

Keeping track of things to do and things done is a functionality so dear to computer users' hearts, they have been reported to repurpose all sorts of mechanisms to accomplish it, such as appropriating email inboxes [7] and code commenting [31] for this purpose. Yet, other than bug trackers (which do not work at the granularity of snippets of code), few programming environments support to-do listing. A clear opportunity for designers of end-user programming environments and scripting environments is providing an easy, lightweight way to support to-do listing.

New Strategies. Finally, there were three strategies that had not been present in the spreadsheet study: control flow, getting help, and proceeding as in prior experiences.

Control Flow means following the flow of control (sequence in which instructions are executed). Pennington found that expert programmers initially represent a program in terms of its control flow [24]. Since spreadsheets do not provide a view of execution flow, it is not surprising that following control flow did not arise in the spreadsheet study. The scripting environment, however, provided multiple affordances for viewing control flow, and participants used them. For example:

Male P0717080900 used the call stack to see what subroutines were called and in what order.
Female P0718081400 placed a breakpoint on the first line to run the script in control flow from there in order to understand it.
Male P0117081130 stepped over and into to examine and execute the code in the order it was run.

Providing support for following control flow is relatively widespread in programming environments for professional programmers, but less so for end-user programming environments. A notable exception is the approach for allowing control flow following in the rule-based language KidSim/Cocoa/Stagecast [12], which features the ability step through the program to see which rules fire in which order.

Help means getting help from other people or resources, a common practice in real-world software development. For example, Ko et al. reported that developers often sought information in hard-to-search sources, such as coworkers' heads, scanned-in diagrams, and hand-written notes [18]. In our study, examples of following help included searching for help on a bug using Google Search, consulting the internal help documentation in order to set a breakpoint, or asking the researchers what a particular function does. This strategy was not available in the spreadsheet study but our script participants used it extensively.

Female P0717081630 sought help from the observers.
Male P0718081030 sought online help.
Male P0117081130 used the interface's help menu item.
Female P0718081400 used the command line's "-?" and "/?" commands.
Male P0718081030 used the function key to bring up the internal help. Later, he also brought help up on a particular word by first highlighting it and then hitting the function key.

Finally, one participant attempted to integrate external help with code inspection:

Male P0718081030 restored down the help window, to be able to look at the code and still have the help in an open window next the code.

Proceeding as in Prior Experience was recognizing a situation (correctly or not) as one experienced before, and using that prior experience as a blueprint of next steps to take. Sometimes the recognition was about a feature in the environment that had helped them in the past and sometimes it was about a particular type of bug. Once recognition struck, participants often proceeded in a trial-and-error manner, without first evaluating whether it was the right path. For example:

Male P0717080900: "Ah – I've seen this before. This is what must be wrong."
Female P0718081400: "It obviously needs to go up one directory."
Male P0721080900 said: "Just for kicks and giggles, let's try this."
Male P0718081030 felt something strange was going on and, from an earlier experience, decided that it was PowerShell's fault. He therefore closed the environment and opened it up again.

We suspect that proceeding as in prior experience is quite widespread, but it has not been reported in the literature on debugging. Given humans' reliance on recognition in everyday life, this strategy could be having a powerful influence on how people debug. It is an open question whether and how designers of debugging tools might leverage the fortunate aspects of this and take steps to help guard against the unfortunate aspects.

4.2 Sequential Usage of Strategies: Three Participants

To investigate how the participants used these strategies when succeeding, we analyzed three participants in detail. The first two were the most successful male (who fixed four bugs in one task) and the most successful female (who fixed one bug in one task). We then analyzed a male with the same scripting experience as the female (who also fixed one bug in one task). Each of these participants thus provided at least one successful event to analyze, in addition to several failed attempts. Fig. 2 shows the sequence of strategies used in one of the two tasks by these participants.

As an aside, the overall low success rate on the number of bugs fixed was expected, because we deliberately designed the tasks to be difficult, so that strategizing would occur even with expert scripters. For example, one of our participants (the most successful male) was extremely experienced, having written more than 100 PowerShell scripts in the past year. He fixed all seven bugs in the two tasks. (Reminder: the figure shows only one of those two tasks.)

The most successful female described herself as a software developer. She was about 30 years old, and had nine years of scripting experience (in JavaScript, PowerShell, Perl, and Bash/UNIX Shell Scripting). Within the past year, she had written about five PowerShell scripts and was a frequent PowerShell user, normally using it about two to three times per week.

As Fig. 2 shows, after reading the task description, this female began by running the script: “First thing I’m going to do is to try to run it to see what the errors are.” Using the error message which stated there was an error at a line which contained “Type = ‘NewLine’” because “Type” is a read-only property, she navigated directly to that line of the script. She right away noticed that the equal sign was doing an assignment instead of a comparison, thereby finding the first bug (the dashed bar at the beginning of her session in Fig. 2). But, although she knew what the error was, she fixed it incorrectly based on her prior experience with other languages (the solid line with a dot followed by 45 seconds of testing). Fortunately, testing her change made her realize that her fix was incorrect: “Ok, perhaps it was wrong...” Despite her experience with scripting and using PowerShell, she said she felt silly about not remembering what the correct syntax was, but that it is due to her not writing scripts from scratch in PowerShell often, but rather reusing and extending existing scripts.

Knowing what she wanted the program to do but not the syntax to accomplish it, she started to use code inspection to find a suitable fix by looking for examples in related code: “That’s why I usually start looking at other files, to see if there’s an ‘equal’ type thing.” She went on to skim two other PowerShell files, rejecting two Boolean operators she did not believe would fix her problem. However, the second one, even though it was not exactly what she needed, was close enough to enable her to fix the bug by patterning her change after that code: “Aha! ‘–like’ isn’t it because that would be like a ‘starts with’ type thing. So, maybe I need to do ‘-eq’?” This use of code inspection is what enabled her to actually fix the bug, and is a good example of *how* increased use of this strategy might have led to greater female success in [32].

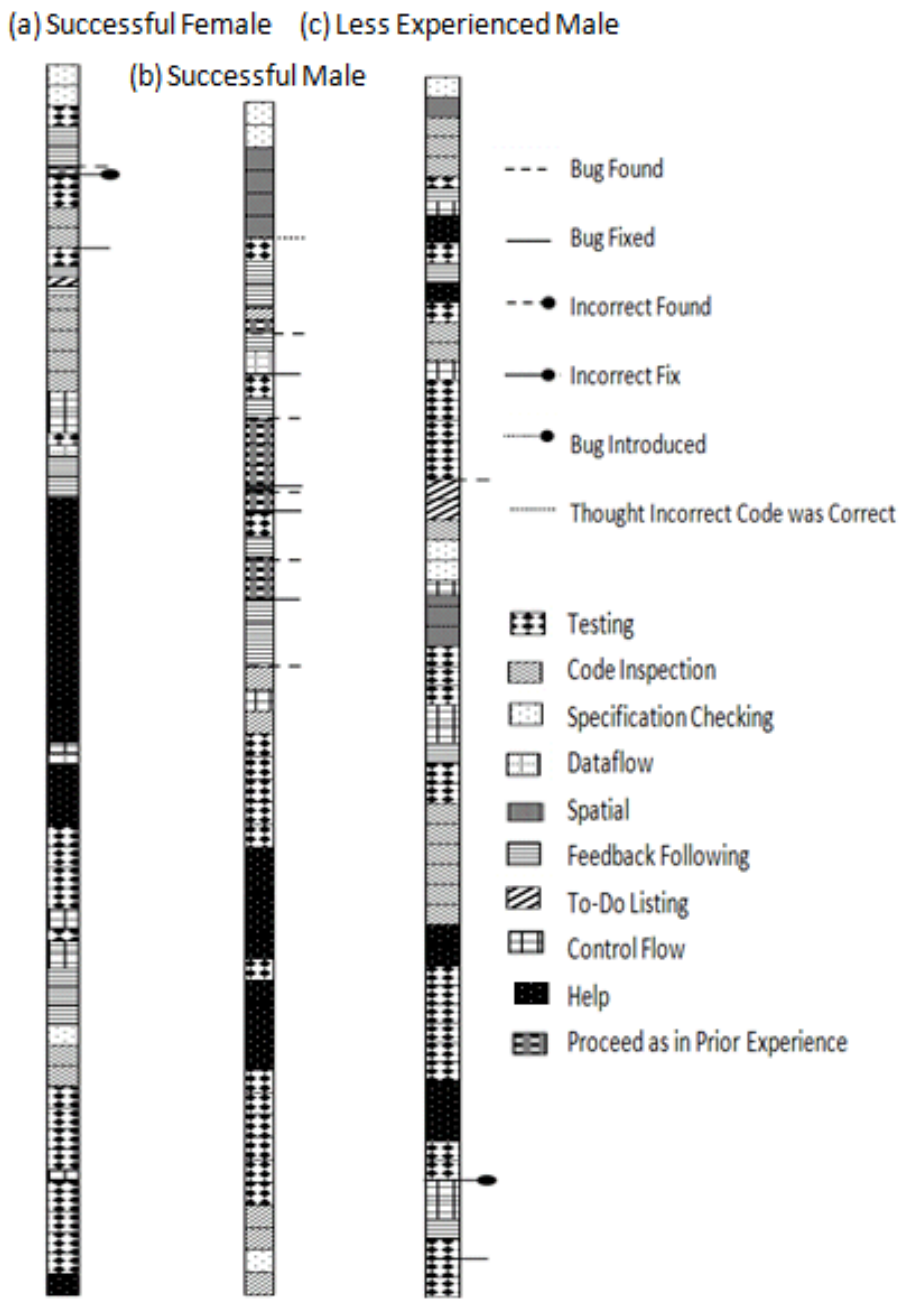


Fig. 2. The strategies used by three participants during one of the two tasks, as well as when bugs were found and fixed. Each patterned square is a 30-second use of the strategy shown in the legend, and the lines display a bug found / fixed also shown in the legend. The start of the session is at the top and the end at the bottom.

The female's use of code inspection to actually *fix* the bug above, rather than just to *find* it, is interesting. It suggests a possible new debugging functionality, code mini-pattern recognition and retrieval, to support searching and browsing for related code patterns to use as templates. The female's beneficial use of code inspection in this study is consistent with the results from [32] that code inspection was statistically tied to female spreadsheet users' success. These combined results suggest the following hypotheses to more fully investigate the importance of code inspection to female debuggers:

Hypothesis 1F: Code inspection is tied to females' success in *finding* bugs.

Hypothesis 2F: After a bug has been found, code inspection is tied to females' success in correctly *fixing* the bug.

Hypothesis 3F: Environments that offer explicit support for code inspection strategies in fixing bugs will promote greater debugging success by females than environments that do not explicitly support code inspection strategies.

In contrast to the female, for males, code inspection did not appear to be tied to success, either in the earlier spreadsheet study or in this one. As the figure shows, the successful male used very little of it, and used none in the periods after finding, when working on actually fixing the bugs. Although the low-experience male did use code inspection, it did not seem to help him very much. Thus, we predict that a set of hypotheses (which we will refer to as 1M, 2M, and 3M) about males like the female-oriented Hypotheses 1F, 2F, and 3F will produce different results in follow-up research, because instead of emphasizing code inspection, the periods near the low-experience male's successful finding of a bug and near his successful fixing of the bug contained a marked emphasis on testing. (We will return to this point shortly.)

The successful male, whose sequence of strategies is also shown in Fig. 2, was a very experienced scripter. He described himself as a scripter (not as an IT professional or developer) and had 20 years of experience writing scripts in languages such as Korn Shell, BIN, PowerShell, Perl, and Tcl. He had used PowerShell since its inception and had written over 100 PowerShell scripts within the past year alone.

After reading the task instructions, the successful male did not begin as the female did by running the script, but instead first began by reading the main script code from top to bottom for a couple of minutes, "The first thing I'll do is to read the script to find out what I believe it does." Once he got to the bottom of the script, he stated that "this code didn't seem to have anything wrong with it," denoted by the dotted line in Fig. 2. He was incorrect about this.

After the dotted line, this successful male switched to running the script to see its outputs (testing) and to consider the resulting error messages (feedback following). The first error message this male pursued was the second error message that the successful female had also tried addressing: "cannot find path because it does not exist." Without even navigating to the function to which that the error referred, the successful male was able to draw from his prior experience, immediately hypothesizing (correctly) that the error was caused by a function call in the main script that used the "name" property as a parameter, rather than the "full name" property of a file. He stated, "I know that the file type has a 'full name' property, so that's what we need to do." After changing the code, to check his change before really declaring it a fix, he opened the function that the error message referred to, checking to see how the file name that was being passed as a parameter was being used (dataflow). At this point, he declared the first bug fixed, and reran the script to see what problem to tackle next. He used a similar sequence of testing, feedback following, and prior experience for the next three bugs he found and fixed.

But when the successful male found the fifth bug (see the fifth dashed line in Fig. 2), he did not have prior experiences relevant to fixing it. As the right half of the figure shows, he spent the rest of the session trying to fix it, mainly relying on a combination of fine-grained testing (checking variable output values) and help (documentation internal to the product on debugging PowerShell scripts), with bits of code inspection, control flow, and specification checking also sprinkled throughout.

Thus, the successful male provided interesting evidence regarding code inspection, testing, prior experience, and dataflow. We have already derived hypotheses about code inspection, and we defer hypotheses about testing until after discussing the second male. Regarding prior experience, both the successful male and the successful female drew on prior experience in conjunction with feedback following, but the female's prior experience had negative impacts when she tried to fix a bug by remembering the syntax from a different language. The interplay between feedback following and proceeding as in prior experience is thought-provoking, but there is not as yet enough evidence about this interaction and gender differences for us to propose hypotheses for follow-up.

Dataflow, however, was also a successful strategy for the males in [32], and this successful male's experience with it suggests exactly *where* it might be contributing to males' success:

Hypothesis 4M: Dataflow is tied to males' success in *finding* bugs.
Hypothesis 5M: After the bug has been fixed, dataflow is tied to males' success at *checking their fix*.
Hypothesis 6M: Environments that offer explicit support for dataflow will promote greater debugging success by males than environments that do not explicitly support dataflow.

We do not expect the corresponding female Hypotheses 4F, 5F, and 6F to show significant effects, since we have seen no evidence of it in either study.

The successful female was much less experienced than the successful male, so we also compared her strategies to those of a less experienced male to obtain insights into strategy differences due solely to experience. This male had nearly identical experience to the female: 10 years of scripting experience (in CMD, VBScript, PowerShell, T-SQL, and SSIS). In the past year, he had written about six PowerShell scripts, and used PowerShell about three times per week.

Like the successful male, this less experienced male also started out with inspecting the code from top to bottom. The less experienced male examined most of the script very carefully, highlighting the lines he read as he went along. He used several strategies (including testing, feedback following, control flow, and help) to better understand a construct he had never run into before. After about four minutes of trying, he noted not completely understanding that part of the code and assumed that it was correct (which was true), stating that the part he had been studying seemed like a "red herring" and "a no-op". He therefore went on to examine the next line.

Directly following about three minutes of incremental testing (running only fragments of the code at a time to see what they output), the lower-experience male found a bug (dashed line in Fig. 2). At that point, he stated "I'm making a note of a bug that's here; that we're not making a path here… And we're going to fail, because the system is simply not going to find those files." After having made the note, he went on trying to use several strategies (mainly testing and code inspection) to understand the rest of the code.

In the earlier study, we saw some evidence pointing in the direction of to-do listing being a strategy used more by females [32], and two of the three females used it in this study too. This male employed a pen-and-paper version of to-do listing, but to-do listing was so scarcely used overall in this study (perhaps since it was not supported by the environment) that we could not derive hypotheses based on these data alone.

By inspecting the code in control flow order, the less experienced male realized that an incorrect property used for one of the variables was the cause of the faulty output. Returning to the first bug he had written down on paper, he succeeded at fixing the bug through the use of testing. Specifically, he copied that variable and its property into the command line and ran the command. He stated that the output was incorrect, since it was the name of the file instead of its full path. Using tab-completion in the command line, he deleted the property, and tabbed through the list of all properties. He then also used a command to output a list of all properties and skimmed through them, wondering, "Is there a FullPath property?" There, he found a "FullName" property. He tried it out by typing the variable name and property in the command line again. The output was exactly what he wanted, so he put that small code fragment into the script's code, thereby fixing the bug. This suggests a possibility that a programming environment that supports *systematic debugging-oriented testing* mechanisms, such as tracking incremental testing and testing of small fragments of code, may be helpful to testing-oriented debuggers.

The testing evidence from both males above, combined with that of the previous study, suggests the following hypotheses for follow-up investigation.

Hypothesis 7M: Testing is tied to males' success in *finding* bugs.
Hypothesis 8M: After a bug has been found, testing is tied to males' success at correctly *fixing* the bug.
Hypothesis 9M: After the bug has been fixed, testing is tied to males' success at *evaluating their fix*.
Hypothesis 10M: Environments offering explicit support for incremental testing and testing of small code fragments will promote greater debugging success by males than environments that do not explicitly support incremental testing strategies.

We are also proposing identical hypotheses for testing with females (7F, 8F, 9F, and 10F). Our prior study provided no ties between testing and success by females, so we do not predict significant effects for 8F-10F. However, the successful female in this study used testing in conjunction with feedback following to successfully find a bug; 7F might therefore also hold true for females.

As we have been bringing out in our hypotheses, the above evidence from all three participants suggests that the debugging stage at which a strategy is used (finding a bug, fixing a bug, or evaluating a fix) might have an influence on females' and males' success with the strategy, and we consider this to be an interesting new open research question. For example, although everyone successfully found at least one bug by incorporating testing, only the lower experience male *fixed* a bug using that strategy. One concrete instance of this open question is, therefore, whether there is a difference in *how* males and females use testing. For example, might males incorporate testing into both finding and fixing, whereas females use it for only in the finding stage? We express this open question as a general hypothesis:

Hypothesis 11MF: Males' and females' success with a strategy differs with different debugging stages (finding a bug, fixing a bug, or evaluating a fix).

5 Conclusion

This paper presents the results from a think-aloud study we conducted to see how well end-user programmers' spreadsheet debugging strategies generalize to a different population and a different paradigm: IT professionals debugging Windows PowerShell scripts. Our results show that:

- All but one of the strategies found with the spreadsheet users also applied to IT professionals debugging scripts, along with three more that emerged. The seven strategies we observed in both studies were: *testing*, *code inspection*, *specification checking*, *dataflow*, *spatial*, *feedback following* (a generalization of the strategy previously termed *color following)*, and *to-do listing*. In addition, we observed the following three strategies that had not been present in the spreadsheet study: *control flow*, *help*, and *proceeding as in prior experience*.
- The mechanisms scripters used revealed several opportunities for new features in scripting environments, such as support for *systematic* incremental testing, for easy inspection of large amounts of code and of code mini-patterns, for "drill down" into related testing information during code inspection and into related code information during testing, for informal specification checking, and for to-do listing.
- The evidence of the earlier statistical study on spreadsheets combined with the qualitative analysis of this study's participants produced several detailed hypotheses on gender differences in successful strategy usage.

Perhaps the most important contribution of this study is that it raised a significant new open question: whether males' and females' uses of debugging strategies differ not only in *which* strategies they use successfully, but also in *when* and *how* they use those strategies.

Acknowledgements. We thank the participants of our study and are grateful to Jennifer East, Curtis Posadas, and Siddhika Nevrekar for recruiting them. Finally, we thank the anonymous reviewers, whose feedback helped us significantly in improving the paper.

References

1. Bandura, A. Social Foundations of Thought and Action. Prentice Hall, Englewood Cliffs, NJ (1986).
2. Basili, V., Selby, R. Comparing the Effectiveness of Software Testing Strategies, IEEE Trans. Soft. Eng. 13, 12, pp. 1278-1296 (1987).
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. ACM CHI 2005, pp. 869-878, (2005).
4. Beckwith, L. Kissinger, C., Burnett, M., Wiedenbeck, S., Lawrance, J., Blackwell, A., and Cook, C. Tinkering and Gender in End-User Programmers' Debugging, In Proc. ACM CHI 2006, pp. 231-240, (2006).
5. Beckwith, L., Inman, D., Rector, K., and Burnett, M. On to the Real World: Gender and Self-Efficacy in Excel, In Proc. IEEE VLHCC, (2007).
6. Burnett, M., Cook, C., and Rothermel G. End-User Software Engineering. Comm. ACM 47, 9, pp. 53-58 (2004).
7. Danis, C., Kellogg, W., Lau, T., Stylos, J., Dredze, M. and Kushmerick, N. Managers' Email: Beyond Tasks and To-Dos, In ACM CHI Extended Abstracts, pp. 1324-1327, (2005).
8. Gallagher A., De Lisi R., Holst P., McGillicuddy-De Lisi A., Morely M., Cahalan C. Gender Differences in Advanced Mathematical Problem Solving, J. Experimental Child Psychology 75, 3, pp. 165-190 (2000).
9. Grigoreanu, V., Beckwith, L., Fern, X., Yang, S., Komireddy, C., Narayanan, V., Cook, C., and Burnett, M., Gender Differences in End-User Debugging Revisited: What the Miners Found, IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 19-26, (2006).
10. Grigoreanu, V., Cao, J., Kulesza, T., Bogart, C., Rector, R., Burnett, M., and Wiedenbeck, S. Can Feature Design Reduce the Gender Gap in End-User Software Development Environments? IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 149-156,(2008).
11. Hartzel, K. How Self-Efficacy and Gender Issues Affect Software Adoption and Use. Communications of the ACM 46, 9, pp. 167-171 (2003).
12. Heger, N., Cypher, A. and Smith, D. Cocoa at the Visual Programming Challenge 1997, Journal of Visual Languages and Computing 9(2), pp. 151-169, (1998).
13. Ioannidou, A., Repenning, A., and Webb, D. Using Scalable Game Design to Promote 3D Fluency: Assessing the AgentCubes Incremental 3D End-User Development Framework, IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 47-54 (2008).
14. Kandogan, E., Haber, E., Barrett, R., Cypher, A., Maglio, P. and Zhao, H. A1: End-User Programming for Web-based System Administration, ACM UIST'05, pp. 211-220, (2005).
15. Katz, I. and Anderson, J. Debugging: An Analysis of Bug-Location Strategies. In Human-Computer Interaction, Volume 3, pp. 351-399 (1988).
16. Kelleher, C., Pausch, R., and Kiesler, S. Storytelling Alice Motivates Middle School Girls to Learn Computer Programming, In Proc. ACM CHI 2007, pp. 1455-1464, (2007).
17. Ko, A.J. and Myers, B.A. Designing the Whyline: A Debugging Interface for Asking Questions about Program Failures. In Proc. ACM CHI 2004, pp. 151–158, (2004).
18. Ko, A., DeLine, R., and Venolia, G. Information Needs in Collocated Software Development Teams, International Conference on Software Engineering, pp. 344-353, (2007).
19. Littman, D. C., Pinto, J., Letovsky, S., and Soloway, E. Mental Models and Software Maintenance. In E. Soloway and S. Iyengar (Eds), In Proc. ESP. Ablex, Norwood, NJ, pp. 80-98 (1986).

20. Meyers-Levy, J. Gender Differences in Information Processing: A Selectivity Interpretation. In P. Cafferata & A. Tybout, (Eds) Cognitive and Affective Responses to Advertising. Lexington, Ma, Lexington Books, (1989).
21. Nanja, N. and Cook, C. An Analysis of the On-Line Debugging Process. In G. M. Olson, S. Sheppard, and E. Soloway (Eds.), In Proc. ESP. Ablex, Norwood, NJ, (1987).
22. Nardi, B. A Small Matter of Programming: Perspectives on End-User Computing, MIT Press, Cambridge, Mass. (1993).
23. O'Donnell, E. and Johnson, E. The Effects of Auditor Gender and Task Complexity on Information Processing Efficiency. Int. J. Auditing 5, pp. 91-105, (2001).
24. Pennington N. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. Cognitive Psychology, 19(3), pp. 295-341. (1987).
25. Prabhakararao, S., Cook, C., Ruthruff, J., Creswick, E., Main, M., Durham, M. and Burnett, M. Strategies and Behaviors of End-User Programmers with Interactive Fault Localization, IEEE Symposia on Human-Centric Computing Languages and Environments, pp. 15-22, (2003)
26. Rigby, P., German, D. and Storey, M. Open Source Software Peer Review Practices: A Case Study of the Apache Server, International Conference on Software Engineering, pp. 541-550, (2008).
27. Rode, J.A. An Ethnographic Examination of the Relationship of Gender & End-User Programming, Ph.D. Thesis, University of California Irvine, (2008).
28. Rode, J.A., Toye, E.F. and Blackwell, A.F. The Fuzzy Felt Ethnography - Understanding the Programming Patterns of Domestic Appliances. Personal and Ubiquitous Computing 8, pp. 161-176 (2004).
29. Romero, P., du Boulay, B., Cox, R., Lutz, R., and Bryant, S. Debugging Strategies and Tactics in a Multi-Representation Software Environment. International Journal on Human-Computer Studies 61, pp. 992-1009 (2007).
30. Rosson, M., Sinha, H., Bhattacharya, M., Zhao, D. Design Planning in End-User Web Development, In Proc. VLHCC, IEEE (2007).
31. Storey, M., Ryall, J., Bull, R.I., Myers, D., and Singer, J. TODO or to bug: Exploring How Task Annotations Play a Role in the Work Practices of Software Developers, International Conference on Software Engineering, pp. 251-260, (2008).
32. Subrahmaniyan, N., Beckwith, L., Grigoreanu, V., Narayanan, V., Bucht, K., Drummond, R., Fern, X., Wiedenbeck, S., Burnett, M., Testing vs. Code Inspection vs. … What Else? Male and Female End Users' Debugging Strategies, In Proc. ACM CHI, (2008).
33. Torkzadeh, G. and Koufteros, X. Factorial Validity of a Computer Self-Efficacy Scale and the Impact of Computer Training. Educational and Psychological Measurement 54, 3, pp. 813-821, (1994).
34. Weiser, M. Programmers Use Slices When Debugging, Comm. ACM 25, 7, pp. 446-452 (1982).
35. Whitaker, A., Cox, R., and Gribble, S. Configuration Debugging as Search: Finding the Needle in the Haystack, 6th Symposium on Operating System Design and Implementation, (2004).
36. Windows PowerShell Wikipedia entry. http://en.wikipedia.org/wiki/Powershell. Website accessed on August 20th, 2008.
37. Yuan, C., Lao, N., Wen, J., Li, J., Zhang, Z., Wang, Y., and Ma, W. 2006. Automated known problem diagnosis with event traces. In Proc. ACM Sigops/Eurosys European Conference on Computer Systems, (2006).
38. Zang N. and Rosson, M.B. What's in a Mashup? And Why? Studying the Perceptions of Web-Active End Users, *IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 31-38, (2008).