Report Number: wucse-2009-5

2009

Non-programmers Identifying Functionality in Unfamiliar Code: Strategies and Barriers

Paul Gross and Caitlin Kelleher

Source code on the web is a widely available and potentially rich learning resource for non-programmers. However, unfamiliar code can be daunting to end-users without programming experience. This paper describes the results of an exploratory study in which we asked non-programmers to find and modify the code responsible for specific functionality within unfamiliar programs. We present two interacting models of how non-programmers approach this problem: the Task Process Model and the Landmark-Mapping model. Using these models, we describe code search strategies non-programmers employed and the difficulties they encountered. Finally, we propose guidelines for future programming environments that support non-programmers... **Read complete abstract on page 2.**

Non-programmers Identifying Functionality in Unfamiliar Code: Strategies and Barriers

Paul Gross and Caitlin Kelleher

Complete Abstract:

Source code on the web is a widely available and potentially rich learning resource for non-programmers. However, unfamiliar code can be daunting to end-users without programming experience. This paper describes the results of an exploratory study in which we asked non-programmers to find and modify the code responsible for specific functionality within unfamiliar programs. We present two interacting models of how non-programmers approach this problem: the Task Process Model and the Landmark-Mapping model. Using these models, we describe code search strategies non-programmers employed and the difficulties they encountered. Finally, we propose guidelines for future programming environments that support non-programmers in finding functionality in unfamiliar programs.

2009-5

Non-programmers Identifying Functionality in Unfamiliar Code: Strategies and Barriers

Authors: Paul Gross and Caitlin Kelleher

Corresponding Author: grosspa@cse.wustl.edu

Abstract: Source code on the web is a widely available and potentially rich learning resource for non-programmers. However, unfamiliar code can be daunting to end-users without programming experience. This paper describes the results of an exploratory study in which we asked non-programmers to find and modify the code responsible for specific functionality within unfamiliar programs. We present two interacting models of how non-programmers approach this problem: the Task Process Model and the Landmark-Mapping model. Using these models, we describe code search strategies non-programmers employed and the difficulties they encountered. Finally, we propose guidelines for future programming environments that support non-programmers in finding functionality in unfamiliar programs.

Type of Report: Other

Non-programmers Identifying Functionality in Unfamiliar Code: Strategies and Barriers

Paul Gross and Caitlin Kelleher
Department of Computer Science and Engineering – Washington University in St. Louis
grosspa@cse.wustl.edu, ckelleher@cse.wustl.edu

Abstract

Source code on the web is a widely available and potentially rich learning resource for non-programmers. However, unfamiliar code can be daunting to end-users without programming experience. This paper describes the results of an exploratory study in which we asked non-programmers to find and modify the code responsible for specific functionality within unfamiliar programs. We present two interacting models of how non-programmers approach this problem: the Task Process Model and the Landmark-Mapping model. Using these models, we describe code search strategies non-programmers employed and the difficulties they encountered. Finally, we propose guidelines for future programming environments that support non-programmers in finding functionality in unfamiliar programs.

1. Introduction

Some research predicts that as many as 25 million US workers will perform some job-related computer programming tasks in 2012 [27]. The Bureau of Labor Statistics expects less than 3 million of these workers to be professional programmers [27] and whether all these positions will be filled by formally trained programmers is debatable [4]. Hence there could be about 22 million workers programming without formal training. In addition to the large community of workers performing some programming, there are rapidly growing user communities exploring programming in home or recreational contexts (such as web programmers creating mashups [33]). Many of these users lack formal computer programming training and may want to learn relevant new skills as needed.

Suppose an end-user working on a programming task has an idea for functionality to add to her program. Currently, tutorials and examples written to illustrate specific concepts or techniques may be the best learning resources available to her. Yet with the rise in code repositories a much richer resource exists: source code available on the web. Reading and understanding this source code may be difficult for an end-user and near impossible for a non-programmer.

However, a person can determine a program's utility by observing functionality in the program's output (e.g., Javascript rollovers on a web page or an Excel macro highlighting unique cells). Suppose instead of finding a tutorial or example program that demonstrates how to achieve a particular goal, a user could instead find a program that exhibited the desired functionality and adapt the responsible code to fit his or her context.

To enable non-programmers to select and learn from programs they find on the web we must first understand how non-programmers approach finding code responsible for an observed program behavior.

This paper describes an exploratory study in which non-programmers were asked to find, and in some cases modify, code responsible for specific functionality within unfamiliar programs. In completing the code search tasks, users leveraged *landmarks*, verbally identified points of interest, in both the output and code. They used these landmarks to build *mappings* between code and output and to determine code relevance. We describe this process using two interacting models: the Task Process model and the Landmark-Mapping model. Using these models, we contextualize the strategies and problems non-programmers encountered in finding responsible code. Based on the problems our users encountered, we suggest guidelines for designing programming environments that support new programmers in finding particular code sections in unfamiliar programs.

2. Related Work

We have related work in several areas: Code Navigation, Code Comprehension, and Novice

Debugging. We are not aware of other work focused exclusively on code search by non-programmers.

2.1. Code Navigation

Code navigation studies the strategies programmers use to find relevant areas of concern in code. Most of this research focuses on professional programmers. Recent code navigation studies [16], [18] suggest the navigation process users employ relates to Information Foraging theory [24]. Information Foraging was introduced in the context of web navigation and posits when we search for information we rely upon "information scent" to estimate the probability of finding relevant information by following a particular link. Other work hypothesizes a relationship between code navigation and real world spatial navigation by use of landmarks [5]. Empirical studies suggest systematic navigation practices promote task success [26], and that users of different genders may employ different navigation strategies [8].

2.2. Code Comprehension

Code comprehension researches the mental models programmers use to represent code and how they construct these models. Studies in this area are typically concerned with memory recall of program construction. Our work focuses on short-term program comprehension and its use for non-programmers in code navigation. Two fundamental code comprehension models are generally accepted: top-down [2], where users work to relate program goals to code, and bottom-up [23], where users focus on understanding code elements and then relate these to program goals. Other work suggests experts mix these models in making inquiries [19], and opportunistically choose a model [29].

Brooks [2] suggests beacons as stereotypical code snippets that imply a specific, larger functionality (e.g., a variable swap implies a sort function) aiding programmers to quickly identify common functions. Further work investigates the existence of beacons [1] and suggests experts and novices do recognize a sort beacon [30], [31], while others suggest novices do not reliably detect beacons [6].

Beacons and landmarks (from code navigation) are similar concepts, but Cox [5] distinguishes them by suggesting "that beacons are a component of a landmark". For instance a big outdoor hamburger sign may indicate a restaurant. The sign is a beacon indicating the function of a building. Having found the restaurant it can be used as a navigational landmark.

Novice code comprehension studies observe that novices tend to read code sequentially, line by line, in a bottom-up fashion that ignores control flow information [3], [10], [22]. Other research posits that novices' comprehension strategies differ with familiarity and domain knowledge [17], [32]. Some work suggests that fixing a novice's navigation strategy does not significantly impact their comprehension [21].

2.3. Novice Debugging

Novice code debugging investigates the strategies employed and weaknesses exhibited by novices in attempting debugging tasks. Novice debugging research focuses on users who have a working knowledge of programming models (e.g., sequential execution) and program construction. While still novices they are more skilled than non-programmers. McCauley et al. [20] provide a recent area survey.

Katz and Anderson [11] studied novice debugging and observed two general search strategies: forward reasoning, where "search stems from the actual, written code", and backward reasoning, where "search starts from incorrect behavior of the program". One example of a backward reasoning strategy is "simple mapping" where a novice tries to correlate a specific output result to a line of code. Other debugging strategy research identifies more general strategies such as mental tracing and hand execution [9] and end-user strategies for spreadsheet debugging [13], [25], [28].

Ko and Myers [14] observed end-users inclination towards interrogative debugging and created the WhyLine interface to support it [15].

3. Methods

We conducted an exploratory study in which we asked non-programmers to identify and, in some cases, modify code responsible for specific functionality in the program output in unfamiliar programs.

3.1. Storytelling Alice

For this study we used the Storytelling Alice programming environment [12]. Storytelling Alice allows users to create interactive 3D animated stories by writing programs that invoke methods (e.g., `turn`, `say`, `walk`) on objects (e.g. fairies, trees, people). Storytelling Alice users construct programs using a drag-and-drop interface that prevents syntax errors. The environment supports most programming constructs taught to beginning programmers. Figure 1 illustrates adding a line of code in Storytelling Alice.

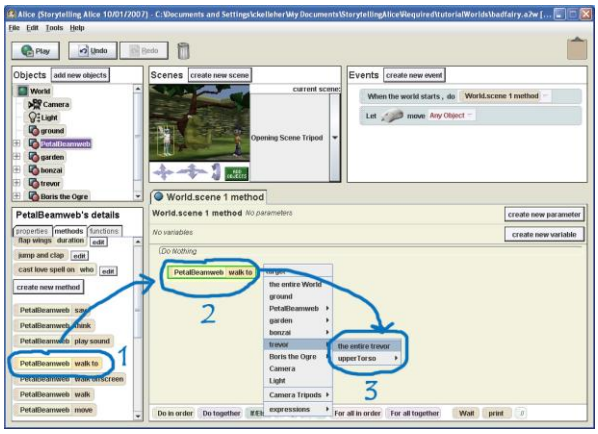

Figure 1. Storytelling Alice where a user programs by (1) dragging a method, (2) dropping it into the code pane, and (3) selecting parameters.

3.2. Instruments

To better understand the properties of programs users are likely to find on the web, we randomly selected 15 programs submitted to the Alice.org user forums for review. Based on the properties of these programs and informal observations of beginning programmers searching through unfamiliar code, we constructed four Storytelling Alice programs that vary along several dimensions:

- *Dialog vs. No Dialog*: Pilot users focused on dialog as a unique marker in the code, but dialog was not present in all the selected programs.
- *Descriptive vs. Ambiguous Object/Method Names*: The selected programs often used poorly chosen or misleading names for methods and objects.
- *Long vs. Short Programs*: The selected programs ranged in length from 25 lines to over 300 lines.
- *Modular vs. Long Code Blocks*: Some Alice programs divided code into appropriate methods while others had long code blocks that contained repeated code sections.
- *More Concurrent Execution vs. Less Concurrent Execution*: Many pilot users relied on sequential execution in searching for actions and struggled with actions occurring simultaneously. However, all of the selected programs used concurrency. Concurrency use ranged from a few concurrent statements to at least 35 concurrent threads.
- *Interactivity vs. Passivity*: Some selected programs contained interactive elements built using events; others were non-interactive stories or animations.

We describe our four programs and their placement in these dimensions in Table 1. We constructed a series of five tasks of varying complexity for each program.

3.3. Study Sessions

The study took place in single, two-hour long sessions. At the beginning of a session, participants filled out a short survey about their computing experience and completed the in-software tutorial provided with Storytelling Alice. The in-software tutorial includes three chapters that introduce users to navigation, program construction and editing, creation of user methods, and the use of events.

3.3.1. Study Task Types. The study included two types of tasks: *bounding* tasks and *modification* tasks.

Bounding tasks required participants to mark the beginning and end of the code responsible for the functionality identified in the video. We refer to these markers as *beginning bounds* and *ending bounds*. This type of task simulates a user who has found a program with an interesting feature and wants to find the code that implements that feature.

Modification tasks ask participants to make a very specific change to the code which affects the functionality as indicated to the user. Modification task videos included titles indicating that the task requires a modification and showing the initial output, an intentionally minimal description of what to change, and the target output.

To avoid providing linguistic cues that might bias participants' search strategies, we presented tasks using

Table 1. A description of the four programs used in the tasks and their properties

Program Name: Description	Dialog	Object/Method Names	Program Length	Modularity	Concurrency	Interactivity
Fish World: three fish swim around and make motions at one another	No	Descriptive	Short	Not Modular	Less	Passive
Woods World: creatures argue about teddy bear, three main methods concurrently execute	Yes	Descriptive	Short	Modular	More	Passive
Magic Trees: two kids discover fairies hidden in trees, large main code block	Yes	Ambiguous	Long	Not Modular	More	Passive
Race World: two students race, winner is randomly determined, user throws bananas	Yes	Ambiguous	Long	Modular	More	Interactive

short video clips of a given program's output. In each video, we highlighted target object(s) and actions using a red box. We faded all other objects in the world.

3.3.2. Task Completion. To ensure participants understood bounding and modification tasks we asked each subject to complete one task of each type in a practice Storytelling Alice program. After completing the two practice tasks, participants completed a series of experimental tasks. We generated the task series by randomizing the presentation order of the four programs and the five tasks for each program. The randomization was intended to prevent any ordering effects. Each participant completed as many tasks as he or she could during the allotted time for the study.

For both the bounding and modification tasks, the target sections of code were embedded within much larger programs. Participants searched through the code and watched both the video and the running program to identify target actions to search for. We asked participants to think aloud while completing these tasks.

3.4. Data

We collected a pre-study demographics and computer experience survey, video recordings of participants as they used Storytelling Alice, screen captures of participants' Storytelling Alice interactions and participants' modified programs.

3.5. Participants

Fourteen adults (university students or employees) participated in the study. Twelve had no prior programming experience. Two participants had previous exposure to programming, one "at least 5 years ago" and the other more than 20 years before. Participants reported using computers an average of 23 hours per week. Participants primarily used web browsers, email, and office productivity applications.

3.6. Analysis

The two authors independently coded each session video. The coding scheme consisted of two types of information: *searches*, and *landmarks*.

3.6.1. Searches. For each search, we coded beginning and ending times for the search, the search space and the participants' search target. Searches could occur in four spaces: the video, the running program, the Storytelling Alice code pane, and other Storytelling Alice panes (e.g. object tree, object details, events).

3.6.2. Landmarks. As users searched for specific functionality within an unfamiliar program, they often verbally referenced specific features in the output (the video or running program) or the program itself (code pane or other panes). For example, a participant might say, "The fish gets bigger and turns" while watching the output. We call these features landmarks as suggested by Cox [5] because the verbalizations are often coupled with code navigational logic (e.g., "The fish spins before he turns to face the camera").

For each landmark, we coded the landmark content, the data type (e.g., object, action, text) and the source (video, running program, code pane or other panes). Additionally, we recorded a specific reason for the usage of each landmark. A landmark might be used as a temporal comparison or identified as included in or excluded from the participant's search target. This landmark record gives insight into the information used by subjects in search and how that information is used to find responsible code.

3.6.3. Other Data. We also transcribed participants' statements about their progress or mental models and noted any solutions they generated.

3.7 Error Analysis

To ensure coding consistency, the two authors independently coded two 10 minute sections of two user sessions. The authors reviewed the codings to establish coding guidelines and then independently coded all the remaining sessions. The completed codings have an 82% agreement rate.

4. Results

Finding target code in an unfamiliar program is difficult for non-programmers. Overall, participants generated correct solutions for only 41% of their tasks (33% of bounding tasks and 72% of modification tasks). Participants completing modification tasks frequently tested and changed their answers which contributed to their greater success. Some participants spent more than twenty minutes on a single task.

We present two models that describe how non-programmers approach finding target code in unfamiliar programs. The Task Process model (see Figure 2) represents the task workflow participants used when attempting a task. To account for the information created and used by subjects during the

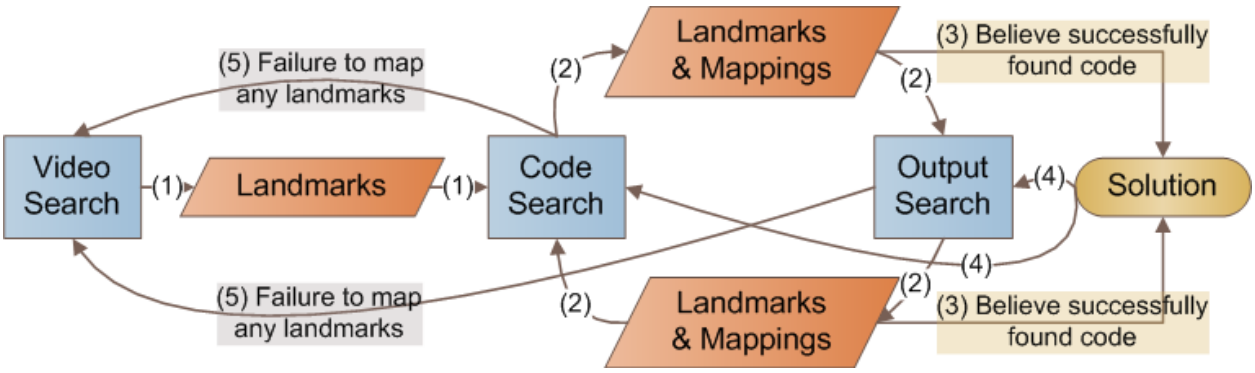

Figure 2. The Task Process Model represents the typical task workflow when a subject attempted a task. The model is broken into five transition sections indicated by the numbers in parenthesis.

Task Process model, we created the Landmark-Mapping model (see Figure 3). This model contains both code landmarks and output landmarks. As participants work through tasks, they develop *mappings* between code and output landmarks.

4.1. Task Process Model Section (1)

The Task Process model is broken into a series of numbered transitions (see Figure 2).

Participants began a task along path (1) by watching the task video. While watching the task video for the first time, 45% of the time participants verbally noted video landmarks (e.g., "the centaur turns" or she says 'Thank you, I'm free'"). Denoting these landmarks added them to the participant's *output landmark set* as indicated in the Landmark-Mapping model.

Two common failures can be seeded in this early section.

Object and Action Encoding (12/14 users, 12/20 tasks): When a user identifies a landmark, he or she encodes that landmark using a description (e.g., "[the pig is] pointing at the cage" or "[the] pig raise's his] right arm"). When users search for these actions in the code, they often do so by looking for key phrases such as "pointing" or "right arm". If they fail to find these phrases the search is never resolved.

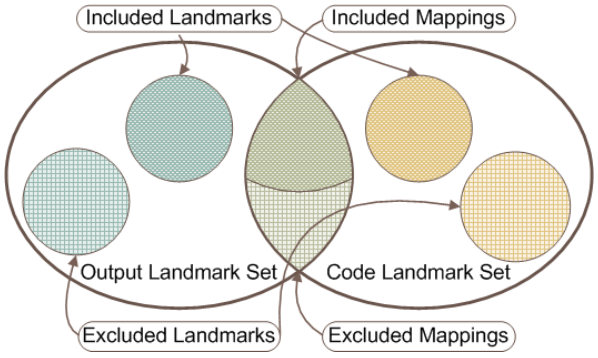

Figure 3. The Landmark-Mapping Model organizes landmarks identified by subjects into two sets that correspond to landmark identification space.

Memory Failure (7/14 users, 8/20 tasks): Sometimes a participant misremembers actions in the video. This can lead the participant to incorrectly use landmarks.

4.2. Task Process Model Section (2)

Having registered a landmark or landmarks from the initial video viewing, participants transitioned to program code and began a "Code Search". In 72% of initial code searches participants verbalized a landmark as the search target. As they navigated the code, 57% of participants identified additional *code landmarks* to search for in the task video or the running program. These code landmarks were added to the *code landmark set* in the Landmark-Mapping model. When participants successfully identified a code section they believed accounted for a landmark, they formed a *mapping* [11]. In the Landmark-Mapping model mappings are in the intersection of the output landmark and code landmark sets.

Participants cycled between "Code Search" and "Output Search" while adding to and refining their landmarks and mappings until they had enough mappings to generate a solution. As the size of the landmark sets grow, participants may begin to organize them into subsets. Participants designated 20% of actions as occurring before or after an existing landmark to include or exclude them from searches. In the Landmark-Mapping model, they are denoted as excluded and *included landmark* subsets.

In this Task Process model section, participants often used the following strategies to build mappings:

Text and Semantic Search (14/14 users, 20/20 tasks, 20% of searches): In a text and semantic search, the participant has identified a target and is scanning either for specific text or for text semantically similar to their

target. This type of search frequently fails when the participant cannot reconcile their description of the landmark (e.g. "[the pig is] pointing at the cage") with a specific line or lines of code.

Temporal Search (14/14 users, 19/20 tasks, 14% of searches): A temporal search occurs when a participant uses temporal information to reason about where the functionality identified in the video is located relative to another landmark. This can help users to narrow the code search space. For instance, in the statement "So it's gotta be somewhere in the part where `basketball3` is front of her, before [`Melly`] turns" the participant identifies two landmarks and uses them to reason about where the functionality identified in the video should lie.

Comprehensive Search (14/14 users, 17/20 tasks, 7% of searches): Participants' focus can switch from global to local when they identify a mapping with high confidence. Comprehensive searches typically occur in a small code section anchored on a specific landmark that is part of a mapping. If the participant believes that the anchor landmark is relevant to the solution he or she may use this strategy to find more supportive temporal landmarks. If the participant does not believe the anchor is relevant, he or she can use the strategy to exclude the current region from the solution. In one comprehensive search, a participant began by identifying an anchor: "So I'm looking for `Dewdrop Willowwind`. So here's `Dewdrop Willowwind` turning to face the camera." Next, the participant maps nearby lines of code: "And [`CordFlamewand`] turn to face the camera. They turn to face the camera and then they all move forward. So this is the moving forward thing [in the video]." This second mapping helped the participant validate the original mapping.

Exhaustive Search (11/14 users, 10/20 tasks, 2% of searches): If the previously discussed strategies are unsuccessful, participants may turn to less structured and more desperate strategies. In an exhaustive search the participant searches the entire recognized code space (note: participants may not search some method implementations because they do not recognize they can). We observed two stages of exhaustive search. In the first stage, participants search any editable method associated with a target character. Failing the first stage, a participant searches all editable methods available regardless of whether they relate to any landmarks or targets they are looking to find.

Not all search strategies are intended to generate a solution. Two common fallback strategies are intended to generate more potential search targets. *API Search* (7/14 users, 8/20 tasks, 1% of searches) occurs when a participant selects an object and scans that object's list of methods to identify new search targets similar to their landmarks. *Explorative Search* (8/14 users, 8/20 tasks, 3% of searches) is a last resort search in which participants appear to randomly click through the interface. Sometimes these random explorations lead the participant to a piece of information that helps the participant formulate a new (productive) search.

4.3. Task Process Model Section (3)

The process of cycling between code and output searches continued until a subject believed their mappings correctly identified a reasonable approximation of the responsible code region. As previously indicated, most solutions are incorrect. Although there are many reasons for incorrect solutions, three failures appear frequently in this Task Process Model section:

Method Interpretation (13/14 users, 12/20 tasks): Participants' abilities to form correct mappings were fundamentally tied to their interpretations of a method's behavior given its name and parameters. A method can provide too many cues, too few cues or inappropriate cues about its function. Missing or misleading cues may cause a participant to inappropriately store a landmark in the included or excluded set of the Landmark-Mapping model. Engebretson and Wiedenbeck call methods' ability to express their functionality role-expressiveness [7].

Lack of Temporal Reasoning (10/14 users, 10/20 tasks): Failure to use temporal reasoning can cause participants to search more code than necessary. They may also fail to utilize operations that can increase the size of their excluded landmark sets (thus reducing the number of landmarks to map). By searching excess code and keeping irrelevant landmarks, participants may create false mappings. Finally, without temporal reasoning a subject may not identify nearby landmarks to verify the correctness of their initial mappings.

Temporal Reasoning Overuse and Ignoring Constructs (13/14 users, 12/20 tasks): Temporal reasoning cannot be naively applied to programs containing constructs such as loops and concurrent blocks or multiple threads of execution. Failure to recognize the changing execution model caused participants to arrive at faulty solutions by incorrectly placing landmarks and mappings into the excluded or included sets of the Landmark-Mapping model.

4.4. Task Process Model Section (4)

For a bounding task, finding a solution required mappings for the first and last action observed hence the transitions back from "Solution" to either "Code

Search" or "Output search". Additionally, participants frequently verified modification task solutions leading to a higher success rate for modification tasks.

4.5. Task Process Model Section (5)

Not all searches or series of searches led to a clear solution. In response to finding no mappings to their landmarks, some subjects turned to *Context Search* (9/14 users, 8/20 tasks, 1% of searches). In a context search, the participant searches the output for actions happening shortly before or after the target functionality. In one case, a participant stated "I was just gonna look again and see…what part in the movie corresponds to …where the `Horse` is highlighted." The participant then identified landmarks immediately before and after the indicated functionality.

Context search usage occasionally gave rise to a common failure we call *Magic Code* (7/14 users, 15/20 tasks). Many participants correctly mapped the temporally related landmarks identified through context search. However, participants then failed to find the original target near these newly identified mappings and concluding: "it is in there, but I can't see it". This conclusion produces an incomplete set of mappings as users may not have mapped the target functionality.

4.6. Relationship to Other Models

Ko et al. [16] studied expert navigation in code maintenance tasks. They propose a model in which developers search for relevant task information, relate this information to previous knowledge to decide their next step, and continue collecting relevant information until they feel they have enough information to implement a solution. Although Ko's model applies to expert programmers, we have found that non-programmers use a similar high-level process. We expand on the task process by suggesting the Landmark-Mapping model as an abstraction to describe how non-programmers collect and organize the information they use to complete their task.

5. Discussion and Conclusions

Insight into how non-programmers search code can inform the design of programming environments that support users in utilizing and learning from found code. While this study focused on participants using Storytelling Alice, we believe the model, strategies, and failures discussed apply to other domains. In particularly domains where most program execution is externally observable such as web sites, user interfaces, and scriptable media authoring environments. To this end, we offer the following design guidelines.

5.1. Connect code to observable output

When users search code for an observed functionality it is essential to help them interpret code in terms of the observed functionality. We could have alleviated our participants' struggles with interpreting code could by showing how the output changed when a line of code executed. To support arbitrary code use by non-programmers, we need to explore how best to provide support in the programming environment that enables users to correctly and quickly form mappings between the code and output.

5.2. Help users reconstruct execution flow

When our participants encountered programs containing programming constructs such as loops, do togethers and method calls, they tended to either interpret all statements as executing sequentially or declare the execution flow incomprehensible. Enabling users to correctly reason about the execution flow can help them to employ temporal reasoning effectively. This has the potential to drastically improve users search efficiency. Often students learn new vocabulary words through contextual clues as they read. As non-programmers explore unfamiliar code, there is an opportunity for programming environments to scaffold users' mental models and reasoning about unfamiliar programming constructs' behavior.

5.3. Provide interactions to fully navigate code

Participants in our study frequently struggled to find all code relevant to a particular search. Incomplete exhaustive searches and participants' magic code creation provide evidence of this struggle. Lacking code navigation affordances is particularly disabling when users will be utilizing code they did not create.

5.4 Help users use poorly constructed code

Programming environments have no control over the properties of code users find on the internet. Yet, lacking other supports, the structure and clarity of code users download can have a profound impact on their success. Programming environments enabling non-programmers to utilize unfamiliar code must help overcome difficulties associated with poorly designed and written code. With an understanding of typical usability problems in user created code, we can build

supports into programming environments that help users to successfully navigate imperfect code. Users are particularly affected by poorly chosen method names. Interfaces enabling users to view details about a method's behavior at the point where that method is invoked can increase the method's information scent and help users decide to explore it or not.

6. References

[1] C. Aschwanden and M. Crosby, "Code Scanning Patterns in Program Comprehension," *Proc. of HICSS*, 2006.

[2] R. Brooks, "Towards a Theory of the Comprehension of Computer Programs," *International Journal of Man-Machine Studies*, vol. 18, pp. 543-554, 1983.

[3] M. S. Carver and S. C. Risinger, "Improving children's debugging skills," in *Empirical Studies of Programmers: 2nd Workshop*., 1987, pp. 147-171.

[4] Committee on Science, Engineering and Public Policy, *Rising above the gathering storm: Energizing and employing America for a brighter economic future*. Washington D.C.: The National Academies Press, 2007.

[5] A. Cox, M. Fisher, and P. O'Brien, "Theoretical Considerations on Navigating Codespace with Spatial Cognition," in *Proc.of PPIG*, 2005, pp. 92-105.

[6] M. Crosby, J. Scholtz, and S. Wiedenbeck, "The Roles Beacons Play in Comprehension for Novice and Expert Programmers," in *Proc. of PPIG*, 2002, pp. 58-73.

[7] A. Engebretson and S. Wiedenbeck, "Novice comprehension of programs using task-specific and non-task-specific constructs," in *Proc. of VL/HCC*, 2002, pp. 11-18.

[8] M. Fisher, A. Cox, and L. Zhao, "Using Sex Differences to Link Spatial Cognition and Program Comprehension," in *Proc. of ICSM*, 2006, pp. 289-298.

[9] S. Fitzgerald, G. Lewandowski, R. McCauley, L. Murphy, B. Simon, L. Thomas, and C. Zander, "Debugging: Finding, Fixing and Flailing, a Multi-institutional Study of Novice Debuggers," *Computer Science Education*, vol. 18, pp. 93 - 116, 2008.

[10] R. Jeffries, "A Comparison of the Debugging Behavior of Expert and Novice Programmers," *Proceedings of AERA annual meeting*, 1982.

[11] I. R. Katz and J. R. Anderson, "Debugging: An Analysis of Bug-Location Strategies," *Human-Computer Interaction*, vol. 3, pp. 351 - 399, 1987.

[12] C. Kelleher, R. Pausch, and S. Kiesler, "Storytelling alice motivates middle school girls to learn computer programming," in *Proc. of CHI*, 2007, pp 1455-1464.

[13] C. Kissinger, M. Burnett, S. Stumpf, N. Subrahmaniyan, L. Beckwith, S. Yang, and M. B. Rosson, "Supporting End-user Debugging: What do users want to know?," in *Proc. of AVI*, 2006, pp. 135-142.

[14] A. J. Ko and B. A. Myers, "Designing the Whyline: a debugging interface for asking questions about program behavior," in *Proc. of CHI*, 2004, pp. 151-158.

[15] A. J. Ko and B. A. Myers, "Debugging Reinvented: asking and answering why and why not questions about program behavior," in *Proc. of ICSE*, 2008, pp. 301-310.

[16] A. J. Ko, B. A. Myers, M. J. Coblenz, , and H. H. Aung, "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance tasks," *Trans. On Software Eng.*, vol. 32, pp. 971-987, 2006.

[17] A. J. Ko and B. Uttl, "Individual Differences in Program Comprehension Strategies in Unfamiliar Programming Systems," in *Proc. of IWPC*, 2003, pp. 175-184.

[18] J. Lawrance, R. Bellamy, M. Burnett, and K. Rector, "Using Information Scent to Model the Dynamic Foraging Behavior of Programmers in Maintenance Tasks," in *Proc. of CHI*, 2008, pp. 1323-1332.

[19] S. Letovsky, "Cognitive Processes in Program Comprehension," in *Papers presented at the 1st Workshop on Empirical Studies of Programmers*, 1986, pp. 58-79.

[20] R. McCauley, S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander, "Debugging: a review of the literature from an educational perspective," *Computer Science Ed.*, vol.18, pp. 67-92, 2008.

[21] R. Mosemann and S. Wiedenbeck, "Navigation and Comprehension of Programs by Novice Programmers," *Proc. of IWPC*, pp. 79-88, 2001.

[22] M. Nanja and C. R. Cook, "An analysis of the on-line debugging process," in *Empirical Studies of Programmers: 2nd Workshop*, 1987, pp. 172-184.

[23] N. Pennington, "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs," *Cognitive Psych.*, vol. 19, pp. 295-341, 1987.

[24] P. Pirolli and S. Card, "Information Foraging," *Psychological Review*, vol. 106, pp. 643-675, Oct 1999.

[25] S. Prabhakararao, C. Cook, J. Ruthruff, E. Creswick, M. Main, M. Durham, and M. Burnett, "Strat-egies and behaviors of end-user programmers with interactive fault localization," in *Proc. of VL/HCC*, 2003, pp. 15-22.

[26] M. P. Robillard, W. Coelho, and G. C. Murphy, "How effective developers investigate source code: an exploratory study," *Trans. on Software Eng.*, vol. 30, pp. 889-903, 2004.

[27] C. Scaffidi, M. Shaw, and B. Myers, "Estimating the numbers of End Users and End User Programmers," in *Proc. of VL/HCC*, 2005, pp. 207-214.

[28] N. Subrahmaniyan, L. Beckwith, V. Grigoreanu, M. Burnett, S. Wiedenbeck, V. Narayanan, K. Bucht, R. Drummond, and X. Fern, "Testing vs. code inspection vs. what else?: male and female end users' debugging strategies," in *Proc. of CHI*, 2008, pp. 617-626.

[29] A. von Mayrhauser and A. M. Vans, "Hypothesis-driven Understanding Processes during corrective maintenance of large scale software," in *Proc. of ICSM*, 1997, pp. 12-20.

[30] S. Wiedenbeck, "Beacons in Computer Program Comprehension," *International Journal of Man-Machine Studies*, vol. 25, pp. 697-709, 1986.

[31] S. Wiedenbeck, "The Initial Stage of Program Comprehension," *International Journal of Man-Machine Studies*, vol. 35, pp. 517-540, 1991.

[32] S. Wiedenbeck and A. Engebretson, "Comprehension Strategies of End-User Programmers in an Event-Driven Application," in *Proc. of VL/*, 2004, pp. 207-214.

[33] J. Wong and J. I. Hong, "Making mashups with marmite: towards end-user programming for the web," in *Proc. of CHI*, 2007, pp. 1435-1444.