

ABSTRACT

Title of Dissertation: TOWARD SYMBIOTIC HUMAN-AI
INTERACTION FOCUSING ON
PROGRAMMING BY EXAMPLE

Tak Yeon Lee, Doctor of Philosophy, 2017

Dissertation directed by: Professor Benjamin B. Bederson
Computer Science

Programming has become a new literacy, but is still inaccessible to ordinary people. Programming-by-example (PBE) is an alternative approach that allows people to teach computers repetitive tasks by demonstrating couple input and output examples of the tasks. While the advancements of PBE have been mainly driven by algorithmic improvements, a growing community of researchers started realizing the importance of issues on the human side of PBE. For instance, inexperienced users often find it hard to provide complete and consistent examples, which is crucial for computers to learn the correct programs. Unfortunately, most PBE systems have limited ways to communicate with users about *what it can or cannot do*, and *how to handle unsuccessful situations*. The lack of symbiotic interaction between human users and PBE engines remain as a major hurdle against a widespread adoption of PBE techniques.

To address the issues on the human side of PBE, this dissertation has four research threads. First, we began with two formative studies to establish a better understanding of inexperienced users' needs and mental models. Second, based on the findings of the formative studies, we developed a Visual Environment for Symbiotic Programming, called VESPY. VESPY interleaves visual programming and PBE techniques, enabling users (1) to decompose complex tasks into small modules on its 2-d grid, and (2) to complete each module by providing input and output examples. Four sample programs demonstrate VESPY's remarkable versatility. However, we also noticed that VESPY still had a number of usability issues. Third, to better understand the usability issues and how to help users out from common mistakes, we conducted an online user study that observed how inexperienced users perform program decomposition and disambiguation, which are the two core activities of PBE. We identified seven types of mistakes, and reaffirmed that informative feedback on those mistakes is crucial for designing usable systems. Finally, we explored the design space of feedback components, in order to understand their impact on user's experience.

My dissertation contributes to the AI and HCI communities with: (i) identification of unmet needs of end-users of the Web; (ii) characterization of non-programmers' mental model; (iii) design process of interleaving visual programming and PBE; (iv) identification of mistakes people make while using PBE; and (v) design and assessment of feedback components for PBE users.

TOWARD SYMBIOTIC HUMAN-AI INTERACTION FOCUSING ON
PROGRAMMING BY EXAMPLE

by

Tak Yeon Lee

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2017

Advisory Committee:

Professor Ben Bederson, chair

Professor Jeff Foster

Assistant Professor Leah Findlater

Assistant Professor Jon Froehlich

Associate Professor Jen Golbeck, Dean's Representative

© Copyright by

Tak Yeon Lee

2017

Statement of Co-Authorship

All work in this dissertation was conducted under the supervision of Dr. Benjamin B. Bederson, and I am the primary contributor to all aspects of this research. Most of the research in this dissertation from Chapters 3 and 5 are updated versions of published papers listed below:

- **Chapter 3:** Lee, T.Y., and Bederson, B.B. Give the people what they want: studying end-user needs for enhancing the web. PeerJ Computer Science 2:e91 <https://doi.org/10.7717/peerj-cs.91>, 2016
- **Chapter 5:** Lee, T.Y., Dugan, C., and Bederson, B.B. Towards Understanding User Behavior of Programming by Example: A Crowdsourced User Study. Accepted for IUI'17: 22nd International Conference on Intelligent User Interfaces, 2017.

Dedication

To my family

Acknowledgements

First of all, I would like to thank my advisor Dr. Benjamin B. Bederson for his endless support and encouragement. Along the long path of this thesis, Ben has always been supportive to my decisions. I learned a lot from Ben about education, research, writing, and even the pronunciation of "image". I also thank Dr. Leah Findlater for her guidance on research methodology and scientific writing. I am very grateful to Casey Dugan, who is the best mentor I have ever met.

I appreciate my dissertation committee: Jeff Foster, Jon Froehlich, and Jennifer Golbeck for their valuable advices and patience.

I would like to thank HCIL night watches and all the precious people who would always be by my side: Uran, Kotaro, Matthew, Meethu, Jonggi, SeokBin, DeokGeun, Joon, Alina, MinKyoung, Angela, SoYoung, YoungSam, HyunJong, and others.

Lastly, I'd like to thank my parents for all the encourage and faith. Most of all, I thank HyeRee for all the love and care.

Table of Contents

Statement of Co-Authorship	ii
Acknowledgements	iv
Table of Contents	v
List of Tables	viii
List of Figures	ix
Chapter 1: Introduction	1
1.1. Motivation.....	1
1.2. Dissertation goals and statement	2
1.3. Approach and overview	3
1.4. Organization of the Dissertation.....	5
Chapter 2: Related Work	7
2.1. End-User Development (EUD).....	7
2.2. End-User Programming (EUP).....	8
2.2.1. End-User Programming (EUP).....	9
2.2.2. Visual programming	11
2.2.3. Dataflow Programming.....	13
2.2.4. Programming-by-Example / Demonstration (PBE / PBD).....	15
2.2.5. Summary	18
2.3. End-User Software Engineering (EUSE)	18
2.3.1. Supporting the exploratory approach of end-user programming	19
2.3.2. Understanding end-user programmer’s mental model.....	19
2.3.3. Preventing mistakes of end-user programmers.....	21
2.4. Human-AI Interaction	23
2.4.1. Direct manipulation vs. Autonomous agent.....	23
2.4.2. Mixed Initiative interaction.....	24
Chapter 3: Formative Study: End-User Needs for Enhancing the Web	27
3.1. Introduction.....	28
3.2. Study 1: End-User Needs on the Web	30
3.2.1. Participants.....	31
3.2.2. Procedure	31
3.2.3. Data and Analysis	32
3.2.4. Result: Challenges	33
3.2.5. Potential Functionality of Web Enhancements.....	42
3.2.6. Design Implications	45
3.2.7. Limitations	48
3.3. STUDY 2: NON-PROGRAMMERS MENTAL MODEL OF COMPUTATIONAL TASKS 48	
3.3.1. Participants.....	48
3.3.2. Method	49
3.3.3. Procedure	53
3.3.4. Data and Analysis	54
3.3.5. Findings	54
3.3.6. Implications.....	57
3.3.7. Limitations	59
3.3.8. Conclusion	59
Chapter 4: VESPY: A Visual Environment for Symbiotic Programming.....	61
4.1. Introduction.....	61
4.2. Design Iteration	62
4.2.1. Version 1: Spreadsheet	62
4.2.2. Version 2: Graph of Multiple Spreadsheets.....	64

4.2.3.	Version 3: List of Operations.....	65
4.2.4.	Version 4: Grid and Semantic Zoom	66
4.2.5.	Version 5: Grid and Pop-up Panel	68
4.2.6.	Version 6: Grid, Pop-up Panel, and Side Panel	69
4.3.	Example Walkthrough	69
4.4.	VESPY System.....	76
4.4.1.	The Grid UI.....	76
4.4.2.	Direct Specification	77
4.4.3.	Programming-by-Example (PBE) Engine	78
4.4.4.	Domain Specific Language (DSL).....	80
4.4.5.	Single-step inference algorithms.....	83
4.4.6.	Multi-step PBE with task recipes.....	87
4.5.	Example Enhancements.....	87
4.5.1.	Example #1: Deep search.....	88
4.5.2.	Example #2: Custom Filter	90
4.5.3.	Example #3: Event Parser for Google Calendar	90
4.5.4.	Example #4: Multi-Attribute Ranking.....	90
4.6.	Preliminary User Study.....	91
4.6.1.	Method	92
4.6.2.	Tasks	93
4.6.3.	Tasks	94
4.6.4.	Results.....	95
4.7.	Discussion.....	95
4.7.1.	PBE vs. Direct Specification.....	95
4.7.2.	Limitations	97
4.8.	CONCLUSION.....	98
Chapter 5:	Understanding Human Mistakes when Programming by Example	99
5.1.	Abstract.....	99
5.2.	Introduction.....	99
5.3.	METHODS	101
5.4.	Experimental System	102
5.5.	SUCCESS RATE.....	105
5.6.	Types of Mistakes.....	106
5.6.1.	Missing steps (found 92 times; 30 were critical).....	107
5.6.2.	Ambiguous cases (29 times; 11 critical).....	107
5.6.3.	Inconsistent or unsupported values (28 times; 8 critical)	108
5.6.4.	Unnecessary steps (15 times; 5 critical).....	108
5.6.5.	Describing with formula (11 times; 7 critical).....	109
5.6.6.	Inconsistent program (3 times; 2 critical)	110
5.6.7.	Empty cases (2 times; 0 critical).....	110
5.7.	LIMITATIONS.....	110
5.8.	CONCLUSION.....	111
Chapter 6:	Experiments on Feedback and Human Mistakes in PBE Systems	112
6.1.	Motivation and Introduction.....	112
6.2.	Experimental System UI.....	114
6.3.	Feedback rules	116
6.3.1.	Missing steps.....	116
6.3.2.	Ambiguous cases	118
6.3.3.	Inconsistent or unsupported values	119
6.3.4.	Unnecessary steps	121
6.3.5.	Describing with formula	121
6.3.6.	Inconsistent program.....	122
6.3.7.	Empty cases	122
6.4.	Methods	122
6.4.1.	Procedure	122
6.4.2.	Closing survey	123

6.4.3.	Compensation	123
6.4.4.	Experimental design.....	123
6.4.5.	Measurements	124
6.4.6.	Participants.....	125
6.5.	Result	125
6.5.1.	The insignificant impact of feedback messages on completion and success rates	125
6.5.2.	Frequency and click rates of feedback.....	128
6.5.3.	Perceived quality of the system and the outcomes	130
6.5.4.	Participant background and behavior.....	131
6.6.	Discussion.....	132
6.6.1.	The insignificant impact of feedback messages.....	132
6.6.2.	Potential reasons and remedies for the high dropout rate.....	132
6.6.3.	Plan for a follow-up experiment: addressing the high dropout rate.....	134
Chapter 7:	Conclusion	136
7.1.	Answers to the research questions	136
7.1.1.	R1. What do end-user programmer need to improve the Web?	136
7.1.2.	R2. How do non-programmers express their programming intent?	136
7.1.3.	R3. Is PBE better than direct specification?	137
7.1.4.	R4. Can inexperienced users perform problem decomposition and disambiguation?..	137
7.1.5.	R5. What is the best feedback design for PBE users?	137
7.2.	Thesis contributions.....	138
7.2.1.	Identification of unmet needs of end-users of the Web	138
7.2.2.	Characterization of non-programmers' mental model.....	139
7.2.3.	Design process of interleaving visual programming and PBE	139
7.2.4.	Identification of human mistakes of PBE	140
7.2.5.	Design and assessment of feedback for PBE users.....	140
7.3.	Future work.....	140
7.3.1.	Crowdsourcing feedback rules to users	141
7.3.2.	Balancing between too much or too little feedback to users	141
7.3.3.	Long-term user study of practical EUP systems.....	142
7.4.	Final remarks	142
Bibliography	143

List of Tables

Table 1. Occupational background of the participants of study 1	31
Table 2. Occupational background of the participants	49
Table 3. In Task 2, the participants were asked to create a filter than removes houses with less than three bedrooms among housing rental posts scraped from Craigslist.com.	51
Table 4. VESPY operations and their required parameters. Subscripted types (e.g. VAL of I _{VAL}) mean that the operation requires the type of the value. I _{DOM} must contain only DOM elements; I _{VAL} can be any type except DOM elements.	82
Table 5. Examples of Substring inference.	85
Table 6. Examples of String Test inference.	85
Table 7. Examples of Number Test inference	85
Table 8. Examples of Arithmetic inference	85
Table 9. Examples of Compose Text inference	85
Table 10. The core set of task recipes in VESPY. If input and output satisfies the condition, the recipe will create temporary nodes (in orange color) and will try to find sub-solution.	86
Table 11. The four tasks for the controlled experiment consist of thirteen problems.	94
Table 12. Wilcoxon signed rank test result of the completion times for each problem. For simple problems that require single steps (P1-P7), the Direct Specification condition equivalent or better performance. However, for complex problems requiring multiple-steps (P8-13), the PBE condition was significantly more efficient (p<0.03)	95
Table 13. With the given description and default examples for each task, participants were asked to add more examples, such as the solution examples shown.	102
Table 14. Success rates (proportion of participants who passed the task) and average numbers of trials for the baseline (Base.) and the experimental (Exp.) conditions. Highlighted cells are significant (p<.05).	105
Table 15. Examples of missing steps	107
Table 16. Examples of ambiguous cases	108
Table 17. Examples of inconsistent or unsupported values	108
Table 18. Examples of unnecessary steps	109
Table 19. Examples of describing with formula	109

List of Figures

Figure 1. Chickenfoot scripting environment running inside the Firefox browser. Users type scripting code in the script editor (left) to automate, customize, and integrate Web applications without examining HTML source code.	10
Figure 2. The Inky command line window. When user types a command in the Input area, the Feedback area shows a list of interpreted and fixed candidates.	10
Figure 3. A screenshot of Scratch, a visual programming environment for creating stories, games, and animations. Children can easily understand and use Scratch’s visual widgets.	12
Figure 4. LabView is a dataflow programming language widely used in laboratories.	12
Figure 5. Sample widgets in Yahoo's Pipes. Users create complex operations by connecting widgets and customizing parameters.	12
Figure 6. A sample mashup in Marmite [86]2] extracts address and other information from a Web page. Users select operators on the left, the widgets in the middle show the data flow, and the table on the right shows the processed data.	13
Figure 7. Quartz Composer can process and render graphical data.	14
Figure 8. The Wrangler interface. Users can select or edit data in the right panel. Then the left-bottom panel shows suggestions of transforming operations based on a user’s latest action. As the user selects one of the suggestions, it will be applied to the data set and appended to the transform script (left-top).	17
Figure 9. The user interface of Karma. By highlighting a segment of text (“Japon Bistro”) in the embedded Web browser (left) and dragging it into the table (right), a user can specify a data retrieval operation.	17
Figure 10. Sorting by year in STEPS. Mock input/output pairs specify each step; nested colored blocks represent structure.	17
Figure 11. WYSIWYT approach highlights potential bugs in spreadsheets. Red borders indicate incorrect cells. Check marks indicate that the cells have passed generated test cases, while question marks indicate that the cells need testing. ..	21
Figure 12. Whyline is a debugging tool in the Alice programming environment. Users can press “why” or “why not” buttons for getting detailed information (e.g. program’s execution history) of specific animated behavior.	21
Figure 13. Conversational Clarification being used to disambiguate different programs that extract individual authors.	23
Figure 14. Program Navigation tab allows users to navigate sub-expressions of a program, and choose among alternative sub-expressions that other programs have suggested.	23
Figure 15. Participants were asked to explain how to draw a histogram of the numbers in the table. In this example, the participant gave histogram bins different codes (A-D), and marked each number with the codes. Since the participant could not put 12 into any bin, he marked the number with question mark and a line that points a missing bin.	50
Figure 16. In this example of Task 2, the participant used scribbles along with verbal statements. For example, the participant wrote variations of keywords that	

indicate “bedroom” used in the list. He/she also circled and underlined the number of rooms in each title to demonstrate the text extraction logic, crossed out titles that did not meet the criteria, and drew arrows from houses to empty slots in the list.	52
Figure 17. In Task 3, participants were asked to describe a simple Mashup program that shows available colors of each individual product in the Main page (top left) extracted from the Product Detail (bottom right) page.	53
Figure 18. The 1 st design of VESPY UI looks like a spreadsheet. Each column represents a list of values. The green arrows represent operations that calculate the next column.	63
Figure 19. The 2 nd design of VESPY UI. Widgets that contain small spreadsheets represent complex program structure such as branching and merging.	63
Figure 20. The 3 rd design of VESPY UI is optimized for showing the description of every operation.	64
Figure 21. The 4 th design of VESPY UI employs a 2D grid and a semantic zoom feature.	65
Figure 22. The 5 th design of VESPY UI includes a pop-up panel that shows details of the currently selected node. The top row represents values of the input nodes and the current node. The middle row explains what operation is assigned to the current node. The bottom row shows a set of operations that users can click to assign to the current node.	67
Figure 23. The VESPY user interface consists of the grid, info, actions, and node details. s can open the UI at any web page by pressing the button on the top right corner of web browsers.	69
Figure 24. A new enhancement is created. The grid UI contains a Trigger node to begin with. The original web page is shown on the right side.	70
Figure 25. User can (1) drag an operation from Actions panel (left) to the grid (center), (2) directly change options of the operation (e.g. “button”, “calculate sum”) in the floating node detail window, and then (3) run the operation by clicking the play button on the right side of the window. Finally, (4) the values of the node will be updated.	71
Figure 26. User can attach new elements to any place in the web page by (1) drag-and-drop an element to the target place, (2) choosing the relative position (before, front, back, after) to the target, and (3) clicking a suggested program in the Actions panel. Then (4) two nodes are added to the grid.	71
Figure 27. User can set an event handler by (1) dragging Trigger operation next to the node containing elements, and (2) setting the correct input channel.	72
Figure 28. s can specify a node that extracts elements at a specific DOM position by (1) create an empty node, (2) click an element of interest and press extract button (repeat twice for extracting a set of elements), and (3) confirm the suggested Extract Element operation in the action panel. Then (4) the empty node is replaced with the node that can extract all the elements at the same position.	73
Figure 29. s can create new elements from values with Create Element node.	73
Figure 30. s can extract specific attributes from elements by (1) creating an empty node next to the elements, (2) clicking the attribute value in the detail window, and (3) confirming the suggested action.	74

Figure 31. A simple enhancement creates a button for calculating total points. Three nodes on the left side create and attach “calculate sum” button to the table. When the button is clicked in runtime, the trigger node executes the following nodes to extract all the points from the table, add them, and attach the result back to the page.	75
Figure 32. A simple enhancement creates a button for calculating total points. Four nodes on the left side attach “calculate total points” button to the Web page. When the button is clicked, the trigger node runs the following nodes to extract all the points from the table, add them, and attach the result back to the page. .	76
Figure 33. Users can bring elements in the input node by (1) clicking the arrow button in the node detail window. (2) When the current node contains elements of the input node, PBE suggests a three-step task that filters the input elements by their properties. (3) Clicking the task will add three new nodes for the filtering task.	78
Figure 34. VESPY PBE suggests single / multiple operation tasks based on the values of the input nodes and the current node. To sort a list of numbers, an user (1) creates an empty node that follows the input node. (2) He starts typing desired output “-5”. However, at this point, PBE can only suggest a task with Number Test + Filter operations. (3) As he typed the sufficient output values, PBE suggests a correct Sort operation. (4) He clicks the suggestion to confirm it as the node’s operation.	79
Figure 35. Filtering a set of table rows by values of a specific column requires the filtered list [c] and the key values for predicate [b]. Users (1) extract key values from the original list, (2).....	80
Figure 36. The syntax of VESPY enhancements.....	81
Figure 37. Representation of the VESPY program. An enhancement consists of multiple nodes. The enhancement in this figure calculates the average of numbers ([1,3,6]) by running the four nodes in the numbered order (1→2→3→4). Each node contains an operation, values, and input nodes. When its preceding node triggers a node, it executes its operation, updates its values, and then triggers its following nodes.	81
Figure 38. An example of Extract Parent operation inference.....	83
Figure 39. The <i>deep search</i> enhancement adds a text input box to the original page. When user types a keyword in the input box, it searches all the linked pages and highlights links whose pages contain the keyword. The main content of the links are attached to the links as well.	88
Figure 40. The custom filter enhancement extracts all the venues from the publication list, and attaches a list of unique buttons. When a button is clicked, it shows only the articles published to the selected venue.	89
Figure 41. The event parser enhancement attaches button to every event in the list. When a button is clicked, it finds an open tab of Google Calendar and fills the input form with the event information.	89
Figure 42. The multi-attribute ranking enhancement adds text boxes to each column header that users can type in their own weight factors. When a factor is changed, it updates weighted total scores and color codes on the right end of the table. It also attaches the Sort button that reorders the table rows by weighted total scores.....	89

Figure 43. An example of the second problem of the Calculating numbers task. Given the two input nodes, the participants need to create an Arithmetic node that multiplies the two node values.....	93
Figure 44. The study UI and basic walkthrough.....	103
Figure 45. The experimental system UI. The TASK section describes the program participants should build. The EXAMPLES contains a table of user-provided examples and feedback from the PBE engine. In the RESULT panel, users press the Teach Computer button to let the PBE engine generate programs based on provided examples, and get feedback. Finally, the HISTORY panel shows all the trials provided for the current task.	114
Figure 46. The mechanism of choosing and locking commands for a step. When the computer generates multiple commands users can choose one among them. Chosen commands are locked to the step, and stay until they got unlocked. ...	115
Figure 47. Probabilities of participants reaching and completing tasks compared across different feedback compositions. Lines indicate the portion of participants who reached specific tasks. Bars indicate the portions of participants who accomplished tasks without giving up. The green line above the other lines suggests that the 'BOTH' setting, which shows both system info and instruction, outperformed the other settings.	127
Figure 48. Probabilities of participants reaching and completing tasks compared to whether the history panel is given or not. Lines indicate the portion of participants who reached specific tasks. Bars indicate the portions of participants who accomplished tasks without giving up. The two lines go along with each other, suggesting that the history panel does not have a strong impact on how many users reached and completed tasks.....	127
Figure 49. # of tasks (and tutorials) that a specific feedback rule was activated and clicked by participants.	129
Figure 50. The closing survey result. The Likert scale ratings generally suggest that the BOTH condition is perceived to be intuitive, effective, and useful to increase the credibility of outcome. However, a few participants perceived the BOTH condition to be hard to understand and ineffective.	129

Chapter 1: Introduction

1.1. Motivation

Programming has become a new literacy, but is still one of the most challenging skills for ordinary people. As of 2016, only 2.54% of the employed workforce in the United States are software developers [83]. To enable ordinary people to perform complex and customizable computational tasks, researchers have proposed the concept of End-User Development (EUD), “a set of methods, techniques and tools that allow users of software systems, who are acting as non-professional software developers, at some point to create, modify, or extend a software artifact” [51]. Since end-user programmers have characteristics different from professional programmers, they need specially designed programming environments, which is the goal of end-user programming (EUP) research. EUP researchers have proposed various approaches for making programming concepts easy to learn, as reviewed in section 2.1. While every EUP approach has its own strengths and weaknesses, the common ground is that end-user programmers need to learn the constructs of such systems, and imperatively specify them.

Programming-by-Example (PBE) is an alternative approach that allows users to teach computers to perform repetitive tasks by demonstrating or providing examples using conventional direct manipulation interface, instead of directly specifying via text-based coding or visual programming techniques [14,50]. Therefore, PBE has been successful in the areas where users can easily demonstrate complete and consistent

examples, such as controlling robot arms, automating repetitive tasks, creating animations, and wrangling structured data (reviewed in Chapter 2.2.4).

Despite its strong potential, the advancements of PBE are mostly driven by technical improvements rather than addressing issues on the human side. For example, users of PBE systems often express frustration at not knowing the capability and limitations of the PBE engine [88]. If users make a mistake while expressing their intent, or if the intent is not expressible, PBE systems would fail without the second plan [44]. There is no easy way to check the correctness of generated programs, especially when extensive test cases are unavailable [54]. Decomposing a complex task into smaller subtasks is a challenge for inexperienced users [25]. In sum, usability issues remain as barriers to widespread adoption of PBE [44].

1.2. Dissertation goals and statement

At a high level, the goal of this dissertation has been to improve the design of PBE systems. More specifically, our goal was to answer the following research questions:

- R1. (Chapter 3) What do end-user programmers need to improve the Web?
 - a. What challenges do end-users experience on the Web?
 - b. What features should EUP system provide to end-user programmers?
- R2. (Chapter 3) How do non-programmers express their programming intent?
- R3. (Chapter 4) Is PBE better than direct specification?
- R4. (Chapter 5) Can inexperienced users perform problem decomposition and disambiguation?
 - a. What mistakes do users make when using PBE?

R5. (Chapter 6) What is the impact of feedback design on user's experience of PBE?

- a. Is showing either *system information*, *instruction*, or *both* helpful for completing tasks, understanding the system, and fixing human mistakes?
- b. Does feedback design affect user's behavior of using PBE features?
- c. Does feedback design affect user's credibility of the programs they make?
- d. Does demographic information affect user's performance and behavior of using PBE features?
- e. Is the history of previous trials helpful for users to understand and fix their mistakes?

We addressed these questions with four research threads: (1) studying inexperienced users' needs and mental models, (2) designing a symbiotic environment that interleaves visual programming and PBE, (3) identifying mistakes that inexperienced users make while using PBE; (4) exploring the design space of feedback for human mistakes.

1.3. Approach and overview

Towards the objectives of the dissertation outlined above, we started by conducting two formative user studies (Chapter 3). First, a semi-structured interview study explored challenges that 35 end-users experience daily, and identified seven categories of web enhancements that would be helpful to be included in future EUP systems. Second, a Wizard of Oz study with 13 non-programmers observed how they naturally explain common computational tasks through conversational dialogue. This study expands existing work with characteristics of non-programmers' mental models. The

findings, though preliminary, suggest that future EUP tools should support multi-modal and mixed-initiative interaction for making programming more natural and easy-to-use.

Building on the findings from the formative studies, we developed VESPY, an end-user programming environment for creating interactive web components (Chapter 4). The development of VESPY was a long iterative process, taking 1.5 years to explore various ways to accommodate visual programming and PBE. The design goal was to interleave visual programming and PBE so that users could decompose complex tasks into modules, and generate solutions for each module by providing input and output examples to the PBE engine. Section 4.5 presents four scenarios of sample enhancements that demonstrate the unique capability and versatility of the approach. We also conducted a preliminary user study with VESPY to compare PBE and direct specification approaches. For complex tasks requiring multiple inferences, PBE outperformed direct specification in terms of user's performance. However, for simple tasks, direct specification was as good as PBE, particularly after participants understood the domain specific language. We also observed that the participants experienced usability issues similar with the other PBE systems.

While PBE systems can be quite difficult for inexperienced users, there is little research on people's ability to accomplish complex tasks by providing examples. Chapter 5 presents an online user study that investigates to what extent inexperienced participants perform decomposition and disambiguation for complex PBE tasks, and identifies types of common mistakes. We developed an experimental PBE system that supports simple tasks (e.g. arithmetic, string extraction, and conditional filtering). Among 161 participants recruited from Amazon Mechanical Turk, only 18.6% (30

participants) could finish the entire study. We identified seven types of common mistakes, and reaffirmed that decomposition and disambiguation are tricky for inexperienced users. In addition, we observed that providing actionable feedback for unsuccessful trials can significantly improve the success rate of users, as compared to simple feedback.

Finally, we explored the design space of feedback for PBE (Chapter 6). First of all, we created three types of feedback messages that included: (1) detection of user intent; (2) system information; and (3) instructions for resolving the current issue. We also developed a *history* panel that shows all the unsuccessful trials for the current task. Using the same experimental system, we compared eight combinations of the feedback design factors. The findings suggest that feedback messages have no significant impact on participants' performance. However, providing both system information and instruction increases the perceived effectiveness of feedback messages. The result also suggests that the high dropout rates and information overloads lowered the validity of the study, we will conduct a follow-up experiment with a revised system and study design. The contributions and future research direction of this dissertation are discussed in Chapter 7.

1.4. Organization of the Dissertation

The rest of this dissertation is organized as eight chapters. In Chapter 2, we discuss a literature review related to my thesis. Chapter 3 reports the Wizard of Oz study result that explores how non-programmers describe computational tasks. Chapters 4 introduces the implementation of VESPY, a visual programming environment that employs PBE techniques. Chapter 5 presents the online user study of how ordinary

people perform PBE decomposition and disambiguation with the seven types of human errors. Chapter 6 reports the follow-up study of the extended feedback components. Finally, Chapter 7 proposes possible future research projects that can extend the current scope of this dissertation.

Chapter 2: Related Work

This chapter provides an overview of end-user development in terms of interaction approaches for making programming accessible for inexperienced people, as well as a brief history of human-AI interaction research. We begin with a general background of how the End-User Development (EUD) paradigm has evolved to make programming accessible to ordinary people (Section 2.1). Section 2.2 describes the End-User Programming (EUP) concept, which is a subset of EUD but focuses on enabling end users to *create* their own programs. We delve into a variety of interaction styles used in EUP systems. In Section 2.3, we review End-user Software Engineering (EUSE), which is another related concept overlapping with EUD and EUP, emphasizing the *quality* of the software that end-users create, modify, and extend. Finally, Section 2.4 reviews research topics that have been advanced toward symbiotic interaction between human and AI.

2.1. End-User Development (EUD)

More and more people use computers on daily basis for diverse, complex and frequently changing needs [9]. Enabling people to solve their own problems is a value in itself. Moreover, professional software developers, who comprise only 2.54% of a total employed workforce in the United States by 2016, cannot fully meet all the needs of the country [83]. EUD is “a set of methods, techniques and tools that allow users of software systems, who are acting as non-professional software developers, at some point to create, modify, or extend a software artifact” [51]. End user programmers are also domain experts such as teachers using a spreadsheet for efficient grading,

interaction designers building working prototypes, and journalists crawling data from Web pages as surveyed by Ko et al. [39].

Spreadsheet applications such as VisiCalc, Lotus 1-2-3, and Microsoft Excel are the first and by far the most successful EUD environment. Although end users may not think they are creating programs, the spreadsheet artifacts they create are actually first-order functional programs [33]. In the early days of personal computers, spreadsheet's EUD support was a major factor for buying expensive machines.

Complex applications such as word processors usually have a lot of functionalities sufficient to satisfy a diverse target user groups but not optimized for a single user. Customization or tailoring of UI is specifying parameters to an existing application to meet the user's needs [9]. For instance, a wide range of tools such as web browsers, word processors, integrated development environments, and even games allow users to add plug-ins and change configurations.

The first step of creating a successful EUD is to understand what additional features people want, and how to enable them to specify those features. In this dissertation, we conducted a formative interview study (Chapter 3) to investigate what problems end-users experience on the Internet, and how they would fix them (Chapter 3).

2.2. End-User Programming (EUP)

End-user programming is defined as “*programming to achieve the result of a program, rather than the program itself*” by Ko et al. [39]. According to the definition, end-user programmer, compared to professional programmers, are less concerned about re-usability, reliability, and security of the programs they create. Instead, end users are

interested in quick-and-dirty ways to solve problems at hands. Programs created through EUP extend functionalities of existing applications (e.g. web pages) or run as stand-alone software. Kelleher and Pausch [36] have surveyed EUP systems, and identified five interaction styles in addition to text-based programming. In this section, we review EUP by their interaction styles [63].

2.2.1. End-User Programming (EUP)

Textual programming is often considered unfriendly to end-user programmers. However, if users had sufficient programming skills, text-based scripting is an efficient and expressive way to use the full functionality of domain-specific languages. For example, early EUP systems for customizing the Web such as Greasemonkey [100] are as versatile as JavaScript, at the expense of requiring professional programming skills. To make the efficiency of text-based programming accessible for end-users, EUP researchers have proposed various interactive supports of textual programming. Chickenfoot [6] automatically identifies page elements matching with user-provided keywords shown in Figure 1. When a user types `click("Go")` command, Chickenfoot finds clickable elements (e.g. hyperlink or button) containing a keyword "Go", and triggers *click* events that are assigned to the elements. Inky [58] has *sloppy syntax* and *rich feedback* features that allows commands with incorrect ordering, missing keyword or parameters. While the user is typing, Inky incrementally and continuously shows *rich feedback* of how it interprets and fixes the command as shown in Figure 2. Many of those supports are later applied to even professional IDE

(e.g. Eclipse) as auto-completion or code quality suggestion plug-ins.

While EUP systems that support other interaction styles (e.g. PBE, visual programming) rarely require end-users to write code from scratch, textual description is still a common representation of existing programs, because once users understand textual description they can easily validate and modify programs [34,37,86,89]. In this dissertation, we employ textual description to present programs generated by PBE

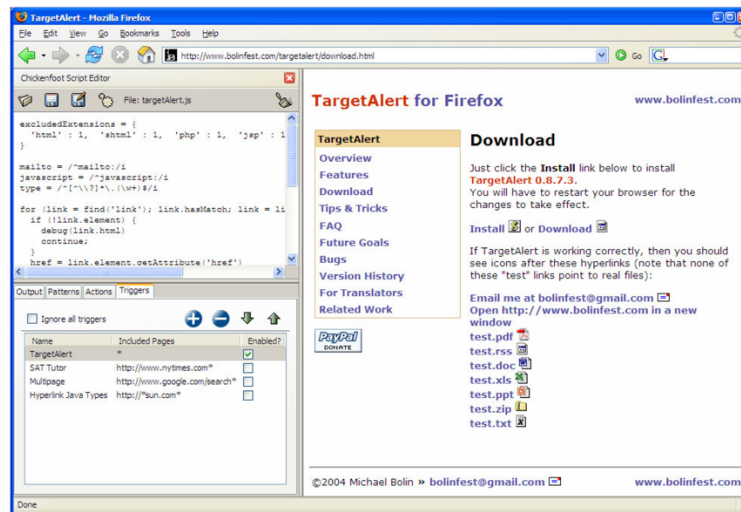


Figure 1. Chickenfoot scripting environment running inside the Firefox browser. Users type scripting code in the script editor (left) to automate, customize, and integrate Web applications without examining HTML source code.

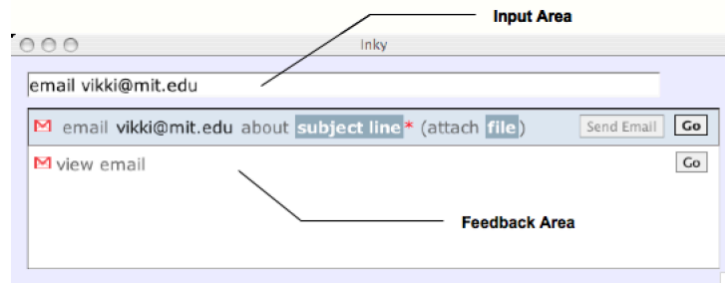


Figure 2. The Inky command line window. When user types a command in the Input area, the Feedback area shows a list of interpreted and fixed candidates.

engines in VESPY (Chapter 4) and the online usability study of PBE (Chapter 6).

2.2.2. Visual programming

To address the steep learning curve of textual coding, many EUP tools employ visual elements to represent low-level language constructs (e.g. commands, control structure, and variables) so that end user programmers can arrange them to build programs, animated stories, and games. Using visual constructs has many advantages. First, the widgets' shapes and colors help users understand program structure and memorize language constructs. Like Lego bricks, connectors of widgets constrain how they should be put together without obscure syntax or punctuation of textual coding. Also the palette of available commands and the options of each command present the tool's capability intuitively. It is not surprising that many visual programming environments have educational purposes, such as Alice [37], LEGOSheets [21], and Scratch [71] (see Figure 3). In spite of their educational benefits, visual programming is often criticized for being impractical for solving real-world problems. For instance, visual blocks take

large spaces on screen, and arranging visual blocks takes much longer than typing code

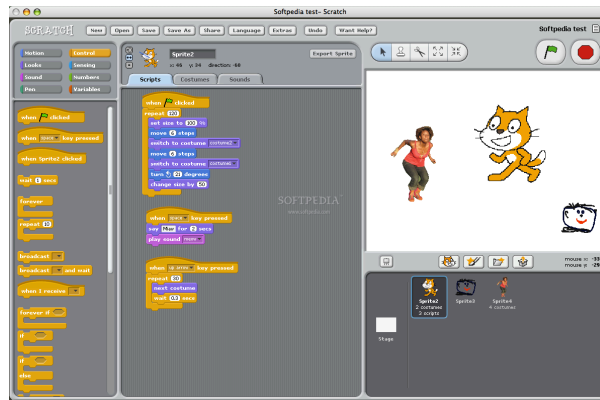


Figure 3. A screenshot of Scratch, a visual programming environment for creating stories, games, and animations. Children can easily understand and use Scratch's visual widgets.

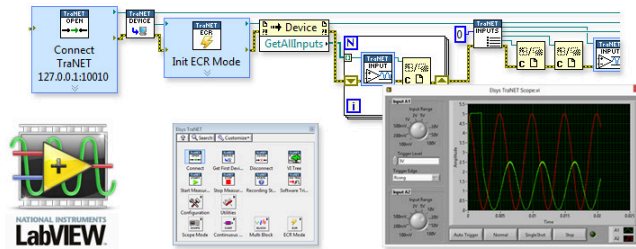


Figure 4. LabView is a dataflow programming language widely used in laboratories.

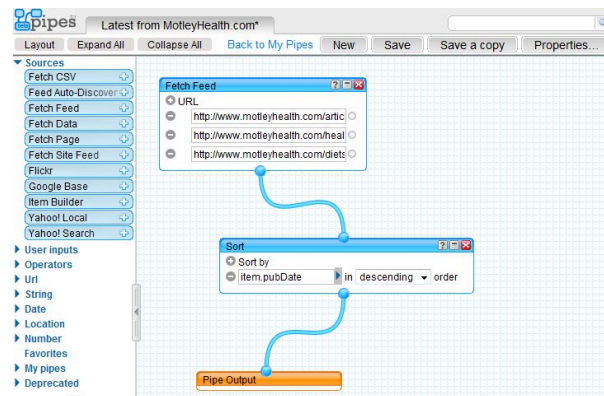


Figure 5. Sample widgets in Yahoo's Pipes. Users create complex operations by connecting widgets and customizing parameters.

[2].

2.2.3. Dataflow Programming

Dataflow programming (DFP) models a program as a direct graph of information flowing between operations [32]. Recently many advancements have been made in visual DFP because complex program structure becomes easy to reason with visualized flow of information. DFP's application domains include signal processing for real-time music / video performance (e.g. Max/MSP¹, Pure Data², VVVV³), processing large amounts of data (e.g. Marmite [86], Karma [81], Yahoo! Pipes [97]), and prototyping interactive UI (e.g. Quartz Composer [99]). DFP falls short when representing complex cyclic control flows such as For-loops and recursions [32]. Thus many DFP tools conceal the entire loop in each operation so that a node deals with a list of input and

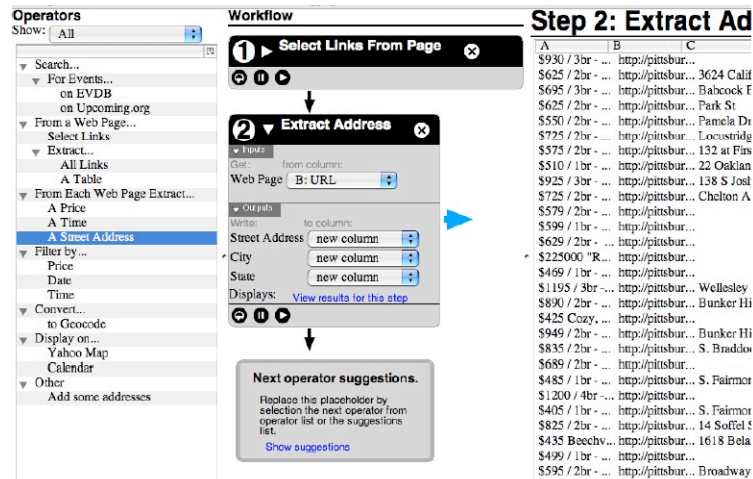


Figure 6. A sample mashup in Marmite [86]2] extracts address and other information from a Web page. Users select operators on the left, the widgets in the middle show the data flow, and the table on the right shows the processed data.

output values without explicit looping.

¹ <http://cycling74.com/products/max>

² http://en.wikipedia.org/w/index.php?title=Pure_Data&oldid=629733021

³ <http://vvvv.org/documentation/vvvv-a-multipurpose-toolkit>

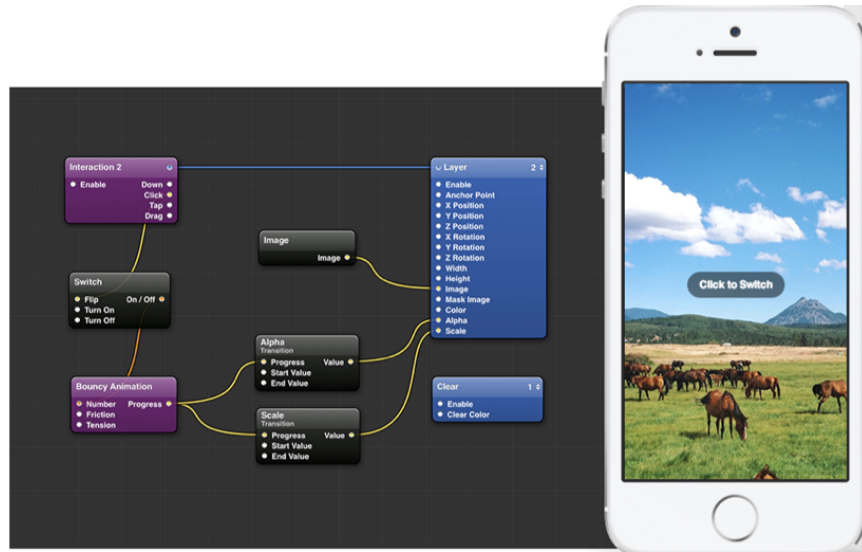


Figure 7. Quartz Composer can process and render graphical data.

LabView⁴ is a well-known DFP for analyzing data in laboratories. Nodes in LabView programs are predefined functions, and connect to each other passing data around as shown in Figure 4. Internet mashup builders have employed dataflow as their program representation. For example, Yahoo! Pipes⁵ (Figure 5) and Marmite (Figure 6) enable users to compose nodes to aggregate, manipulate, and mashup content around the Web. Origami toolkit for Quartz Composer⁶ (Figure 7) is a visual DFP tool for creating interactive design prototypes. Dataflow programming is often confused with visual programming, because both rely on visual elements. The difference is whether visual elements represent either low-level language constructs such as variables, operators, and control flows (in visual programming) or a high-level structure such as sub-process (in dataflow programming). In this dissertation, we employed visual

⁴ <http://www.ni.com/labview/>

⁵ https://en.wikipedia.org/wiki/Yahoo!_Pipes

⁶ https://en.wikipedia.org/wiki/Quartz_Composer

elements for the dataflow approach - allowing users to decompose a complex programming task into small modules.

2.2.4. Programming-by-Example / Demonstration (PBE / PBD)

Programming-by-example (PBE), sometimes called programming-by-demonstration (PBD), is an EUP technique for teaching a computer to perform certain tasks by demonstrating or providing examples using conventional direct manipulation interface, instead of directly specifying via text-based coding or visual programming techniques [14,50]. Given that end users are readily able to demonstrate consistent and complete examples, PBE is supposed to be easier to learn and use than traditional programming. PBE is commonly used for creating animations [55,70], drawing geometric shapes [1], creating macros for repetitive document editing [45] or Web-based processes [47], extracting data from structured documents [42,46,78], transforming data in spreadsheets [23,25], or controlling robot arms [61]. In this dissertation we build an EUP system for data extraction, transformation, and web automation and customization, and we discuss couple PBE systems that are relevant to those topics.

First, generating automation scripts from user's activity is a common use case of PBE. Koala [53] and CoScripter [48] enable end-users to create and share automation scripts to perform Web-based processes. To create an automation script, end-users record their interactions with the Web pages, and correct parts that should be generalized.

Data transformation is a common application of PBE techniques. For instance, Wrangler [34] is a system for interactive data transformation shown in Figure 8. To build a sequence of basic transforms in Wrangler, users demonstrate their intent by

selecting or editing a few examples on the spreadsheet. Then the PBE engine of Wrangler suggests a list of transforming operations ordered by relevancy so that users can choose one of them to apply. The goal of the Wrangler interface is to provide multiple means to add each transformation step so that users can choose the most convenient one for their tasks. Karma [79] (Figure 9) is another example of a data transformation tool that enables users to quickly extract, clean, and integrate data from multiple sources including databases, spreadsheets, text files, XML, and Web APIs. Karma uses PBE techniques that generate data transformation scripts from user's action on its data table.

Text processing tasks (e.g. extracting / replacing substring, reformatting structured text) are tedious and error-prone even for professional programmers. Therefore, text-editing tools often employ PBD / PBE techniques to automate text editing based on a user's keyboard stroke and mouse clicks. For instance, SMARTedit [43] generates macros from repetitive editing actions. Karma [79] and Wrangler [34]) generate corresponding text transforms from multiple pairs of input and output examples. STEPS [89] enables end-users to select and manipulate part of the hierarchical structure of text by example.

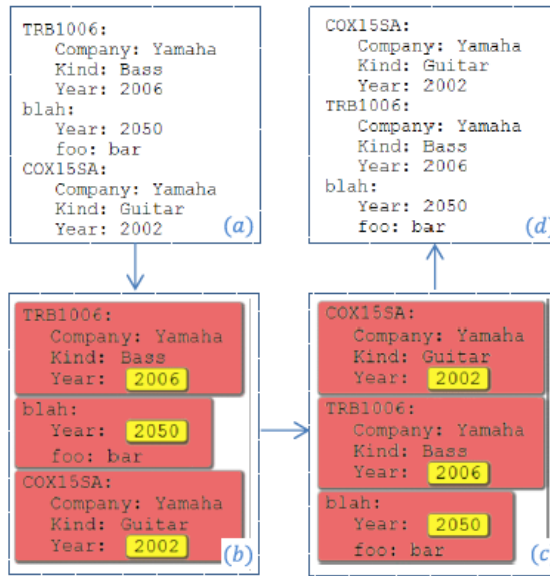


Figure 10. Suggested colored

Year	Property_crime_rate
0	Reported crime in Alabama
1	
2	2004 4029.3
3	2005 3900
4	2006 3937
5	2007 3974.9
6	2008 4081.9
7	
8	Reported crime in Alaska
9	
10	2004 3370.9
11	2005 3615
12	2006 3582

Figure 8. The Wrangler interface. Users can select or edit data in the right panel. Then the left-bottom panel shows suggestions of transforming operations based on a user's latest action. As the user selects one of the suggestions, it will be applied to the data set and appended to the transform script (left-top).

Figure 9. The user interface of Karma. By highlighting a segment of text (“Japon Bistro”) in the embedded Web browser (left) and dragging it into the table (right), a user can specify a data retrieval operation.

2.2.5. Summary

In Section 2.2, we reviewed four interaction styles commonly used in EUP systems. It is noteworthy that none of them is better or worse than the other. Instead, each of them has strengths and weaknesses. Text-based programming is hard to learn, but very effective at describing programs. Visual programming makes it easy to learn basic programming concepts, but not as scalable as text-based or dataflow programming. Dataflow programming provides an effective way to handle the high-level structure of certain programs. PBE allows end users to create programs without learning how to specify them, but it may not be applicable for every task. In fact, most EUP systems employ multiple styles in combination. For example, visual constructs and textual descriptions are commonly used together to describe programs [86,97]. EUP systems provide PBE as well as traditional direct manipulation [34]. In Chapter 4, we build VESPY, an EUP system that employ a combination of dataflow and PBE.

2.3. End-User Software Engineering (EUSE)

End-user programmers may not have the same skills and goals as professional programmers. However, issues of software engineering, such as maintainability, reusability, privacy, and security are essential requirements for the success of EUD. End-user software engineering (EUSE) is a body of research that focuses on systematic and disciplined activities that address the quality of software created by end-users [39]. In this chapter, we review a few research topics in EUSE that are most relevant to this dissertation.

2.3.1. Supporting the exploratory approach of end-user programming

Professional developers usually do not have complete knowledge about the domain, but are supposed to investigate and define requirements of software before starting software development. In contrast, end-user programmers usually have good understanding of their needs, and directly jump into development without specifying requirements or considering other issues of software engineering [72]. End user programmers tend to take evolutionary or exploratory approaches, leaving parts of the design in a rough and ambiguous state. An integrative approach is using community support to help less experience users learn from more experienced end-user programmers. For example, CoScripter community supports end-user programmers to share and extend macro scripts in an enterprise [5]. As another approach, EUP systems often have design critic features that give end-users context-aware design critics for improving their designs [17]. In this dissertation, we propose context-aware critic features to help end-users decompose PBE tasks (Chapter 5 and 6).

2.3.2. Understanding end-user programmer's mental model

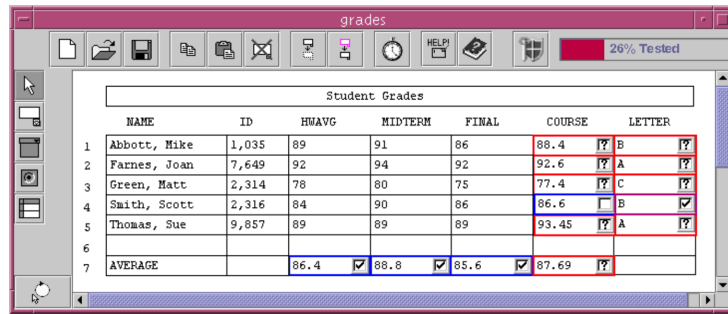
Understanding of end-user's needs and mental model is essential for building successful programming tools [73]. Researchers have studied a wide range of end-user programmers including children [68], teachers, interaction designers [62], or anyone else who would develop programs for professional or personal needs. Keller and Pausch [36] surveyed development environments of novice programmers, mainly focusing on the educational impacts of such settings [51]. Miller [57] examined non-programmers generating procedural instructions in natural language, which resulted in a set of recommended features of programming languages. For instance, he suggested

that *contextual referencing* would be a good alternative method of using variables and traversing data structure. Pane et al. [67] studied vocabulary and structure of non-programmers expressing solutions to computational problems, and identified patterns of imprecise and underspecified information in them.

In this dissertation, I conducted an interview study to examine how people with varying programming expertise express their needs for Web customization (section 3.2), and a Wizard of Oz study to investigate non-programmers describing computational tasks through conversational dialogue (section 3.3). I also examined what mistakes people make while using PBE to solve complex problems (Chapter 5).

2.3.3. Preventing mistakes of end-user programmers

Even though end-user programmers usually create software artifacts for their own needs, small bugs in their code can make critical consequences for failures. For instance, a Texas oil firm lost millions of dollars in an acquisition deal because of a buggy spreadsheet formula [69]. A business Web site with broken links can result in loss of revenue and credibility [73]. In consideration of the quality issues in EUSE, researchers have begun to study what mistakes end-user programmers make, and have proposed supports for preventing such human mistakes. For instance, “What You See Is What



	NAME	ID	HWAvg	MIDTERM	FINAL	COURSE	LETTER
1	Abbott, Mike	1,035	89	91	86	88.4	? B ?
2	Farnes, Joan	7,649	92	94	92	92.6	? A ?
3	Green, Matt	2,314	78	80	75	77.4	? C ?
4	Smith, Scott	2,316	84	90	86	86.6	? B ?
5	Thomas, Sue	9,857	89	89	89	93.45	? A ?
6							
7	AVERAGE		86.4	88.8	85.6	87.69	? ?

Figure 11. WYSIWYT approach highlights potential bugs in spreadsheets. Red borders indicate incorrect cells. Check marks indicate that the cells have passed generated test cases, while question marks indicate that the cells need testing.

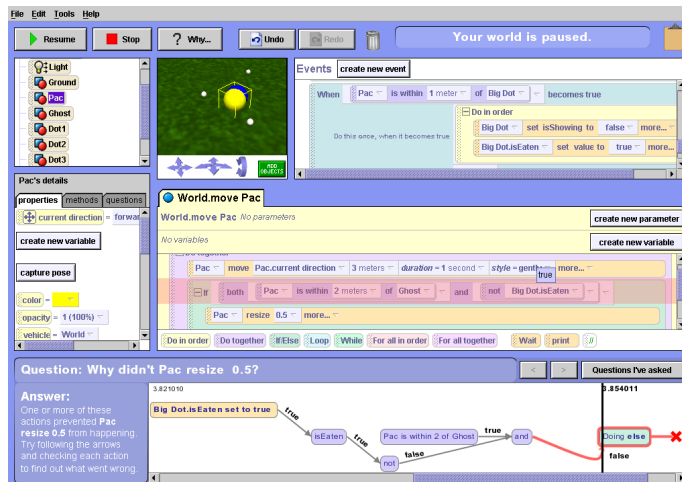


Figure 12. Whyline is a debugging tool in the Alice programming environment. Users can press “why” or “why not” buttons for getting detailed information (e.g. program’s execution history) of specific animated behavior.

You Test” (WYSIWYT) is an end-user testing approach, which helps users systematically validate and find bugs in their spreadsheets [19]. When WYSIWYT finds a potential bug in spreadsheets, it highlights the area with colored borders to attract user’s attention, and adds a tooltip that explain its meaning (Figure 11). *Interrogate Debugging* [40] is another interactive debugging support for the Alice storytelling system. End-user programmers can ask *why did* and *why didn’t* questions for runtime failures in their programmed animations to get detailed information such as the program’s execution history (Figure 12).

End-user programmers using PBE systems also make mistakes. User-provided examples are often ambiguous in that the PBE engine might synthesize an unintended program that is consistent with the provided examples. This can let users lose their confidences in the PBE system, which is a major usability issue of PBE as Lau [44] pointed out. To resolve the ambiguity, researchers have proposed a few interaction models. For instance, Wrangler [35] lets users choose an operation among top candidates. FlashProg [54] have suggested two interaction models. First, *program navigation* (Figure 13) allows users to effectively choose the intended program among a large number of candidates by comparing positive and negative test results. Second, a *conversational clarification* interaction model (Figure 14) asks users specific questions that can effectively resolve ambiguities. A few PBE systems [35,88] support decomposition by allowing users to create multiple operations one-by-one. However, users of such systems are often frustrated at not knowing what the possible primitive operations are [24,88], progress of the current state towards the solution, or intermediate steps to reach the solution [25]. Supporting users in decomposing complex tasks into

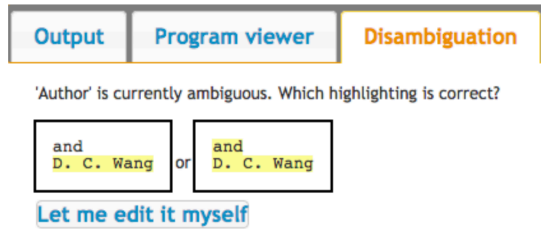


Figure 13. Conversational Clarification being used to disambiguate different programs that extract individual authors.

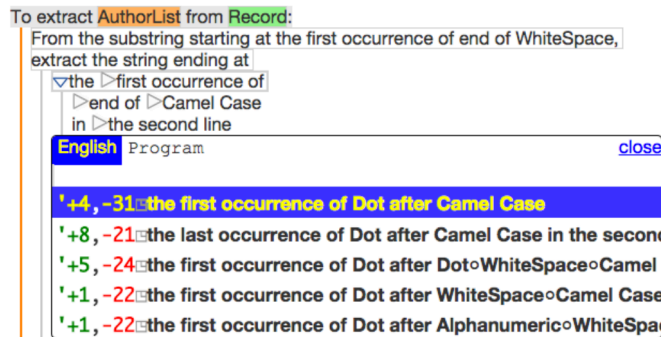


Figure 14. Program Navigation tab allows users to navigate sub-expressions of a program, and choose among alternative sub-expressions that other programs have suggested.

small subtasks and incrementally composing solutions is still an open-ended research question.

2.4. Human-AI Interaction

2.4.1. Direct manipulation vs. Autonomous agent

A great visionary in the beginning of human-computer interaction, Licklider [49] envisioned symbiotic interaction as an optimal collaboration of man and machine, which aims to solve complex problems by tightly coupling human minds and computers. For decades, system developers and researchers have built on his vision, striving for an optimal division of role, responsibility, and initiatives. There was a hot

debate between Ben Shneiderman and Pattie Maes [76] about whether direct manipulation or autonomous agents would be the ultimate form of human-computer interaction. Direct manipulation provides rapid, incremental, reversible actions and feedback that give users the feeling of being in control and the responsibility for the decisions they make [74]. However, as all the initiative has to come from the user, solving complex problems with direct manipulation can be very inefficient and hard to learn [27]. In contrast, autonomous agents proactively keeps track of the user model, and suggest the most likely solutions so that users can delegate complex problems to software agents [76]. The debate did not end up with a winner but an open-ended research question – how to make the two approaches complement each other [29]. For example, autonomous agents can improve the productivity of direct manipulation systems by automating repetitive tasks. Even if users trust autonomous agents and delegate all their tasks, direct manipulation is still important to keep the system comprehensible, predictable, and controllable.

2.4.2. Mixed Initiative interaction

“Mixed-initiative ... refers broadly to methods that explicitly support an efficient, natural interleaving of contributions by users and automated services ... allowing computers to behave like associates ... Achieving ... fluid collaboration between users and computers requires solving difficult challenges.” –[28]

In response to the debate between direct manipulation and autonomous agent [76], mixed-initiative interaction aims to interleave them by letting humans and computers work on shared tasks, monitor each other’s activity, and negotiate who will take an initiative. Eric Horvitz has summarized principles [28] and challenges [29] of mixed-

initiative interaction. Tecuci et al. [20] have introduced seven aspects of mixed-initiative interaction (task, control, awareness, communication, personalization, architecture, and evaluation) to help understand existing mixed-initiative systems and building general design principles. Example applications of mixed-initiative interaction include,

- **Major search engines**⁷ update suggested keywords and searched web pages for every keystroke made by users. The rapid feedback loop enables users to understand what combination of keywords would give better results.
- **Integrated developing environments** predict language constructs that the user is currently typing. It reduces the number of characters to be typed, and also prevents users from making typos.
- **Planning tools (e.g. floor planning CAD [18], meeting scheduler [10], and thermostat [38])** suggest advices (e.g. stove and refrigerator being too far apart in a floor plan) according to a user's activity.

It is noteworthy that the applications of mixed-initiative interaction listed above are exploratory and creative processes where neither users nor computer agents have complete understanding of the problems or the solutions. Instead, users would incrementally refine their goals, based on the solutions suggested by computer agents [26] as shown in the applications such as meeting scheduler [10,96], and thermostat [38].

In spite of the aforementioned opportunities, mixed-initiative interaction may not be the panacea for all usability issues. For instance, users of Proactive Wrangler [25]

⁷ google.com; www.bing.com; search.yahoo.com

did not trust programs that are generated without their initiation. Causal relationships of mixed-initiative systems tend to be more complex than fixed-initiative systems. Developing a mixed-initiative system requires synergistic integration of AI and HCI [20]. From the AI perspective, it might require knowledge representation of the task and user's intention, problem solving and planning, and learning algorithm. From the HCI perspective, it has to design an effective UI for dialogue, intent expression, understanding generated solution, and building trust.

In this dissertation, we applied principles of the mixed-initiative interaction to propose two solutions for usability issues of PBE. First, I proposed a novel interaction model of VESPY that allows both user and the PBE engine to take an initiative of program decomposition (Chapter 4). I also identified patterns of common mistakes that users make while using PBE (Chapter 5), and proposed a novel mixed-initiative feedback mechanism to help users quickly understand and fix mistakes in collaboration (Chapter 6).

Chapter 3: Formative Study: End-User Needs for Enhancing the Web

End-user programming (EUP) is a common approach for helping ordinary people create small programs for their professional or daily tasks. Since end-users may not have programming skills or strong motivation for learning them, tools should provide what end-users want with minimal costs of learning – i.e., they must decrease the barriers to entry. However, it is often hard to address these needs, especially for fast-evolving domains such as the Web.

To better understand these existing and ongoing challenges, we conducted two formative studies with Web users – a semi-structured interview study, and a Wizard-of-Oz study. The interview study identifies challenges that participants have with their daily experiences on the Web. The Wizard-of-Oz study investigated how participants would naturally explain three computational tasks to an interviewer, who acted as a hypothetical computer agent. The two user studies demonstrate a disconnect between what end-users want and what existing EUP systems support, and thus open the door for a path towards better support for end user needs. In particular, our findings from the interview study are (1) analysis of challenges that end-users experience on the Web, and solutions they envision, and (2) seven core functionalities of EUP for addressing these challenges. Findings from the Wizard-of-Oz study include (3) characteristics of non-programmers describing three common computation tasks, and (4) design implications for future EUP systems.

3.1. Introduction

Over the decades, the Web has become the most popular and convenient workbench for individuals and businesses supporting an incredible number of activities. However, developers of Web services cannot completely anticipate future uses and problems at design time, when a service is developed. Thus we can expect users, at use time, will discover misalignment between their needs and the support that an existing system can provide for them [16]. Numerous examples of this misalignment exist. For example, a site designed to support comparison shopping for online shoppers may not meet the needs of shoppers who want to compare prices across different sites and even track daily prices⁸. Another is that people often use customizable applications (e.g. RSS feed readers) to manage ever-growing channels instead of visiting individual sites. More broadly, fraudulent sites and deceptive opinion spam are ongoing concerns for consumers [65]. When a Web page does not match their needs, people often use mashups [15,85,90,91,93], browser extensions and scripts [7,47,60,98] built by third-party programmers. Unfortunately there are not enough third-party solutions to address all 1.4 billion end-user's needs of 175 million websites [78], and enabling end users to develop their own solutions is the goal of end-user programming on the Web (WebEUP).

A clear understanding of end-user needs is essential for building successful programming tools [73]. In this chapter we report two user studies. The first study, a semi-structured interview study addresses the research questions defined at section 1.2:

R1. What do end-user programmers need to improve the Web?

⁸ <http://camelcamelcamel.com>

- a. What challenges do end-users experience on the Web?
- b. What features should EUP system provide to end-user programmers?

The second study addresses,

R2. How do non-programmers express their programming intent?

Answering the above questions is important to have a clear understanding of the direction we should take to develop WebEUP systems that will be useful and effective for a broad range of people.

Prior studies [91–93] characterize potential end-user programmer’s mindset and needs. Researchers also investigated end-user programmer’s real world behavior and software artifacts they created with specific WebEUP tools such as CoScripter [5]. Live collections such as the Chrome Web Store⁹ and ProgrammableWeb¹⁰ are valuable resources that address user needs by community developed scripts and mashups. This chapter reports on an interview study with similar motivations – to investigate what challenges end-users experience and how they would improve – but focuses on unmet needs of 35 end-users on the Web with minimal bias of current technology. Through iterative coding we identify the pattern of challenges that end-users experience. We also suggest seven functionalities of EUP for addressing the challenges - *Modify*, *Compute*, *Interactivity*, *Gather*, *Automate*, *Store*, and *Notify*.

There is a wealth of study for the second research question. Researchers have studied the psychology of non-programmers. Miller [56,57] examined natural language descriptions by non-programmers and identified a rich set of characteristics such as

⁹ <https://chrome.google.com/webstore/category/apps>

¹⁰ <http://www.programmableweb.com/>

contextual referencing. Biermann, Ballard and Sigmon [3] confirmed that there are numerous regularities in the way non-programmers describe programs. Pane et al. [67] identified vocabulary and structure in non-programmer's description of programs. We conducted a Wizard of Oz study with 13 non-programmers to observe how they naturally explain common computational tasks through conversational dialogue with an intelligent agent. The interviewer acted as a hypothetical computer agent, who understands participant's verbal statements, gestures, and scribbles. This study expands existing work with characteristics of non-programmers' mental models.

Findings from the interviews and the Wizard-of-Oz study together demonstrate a disconnect between what end-users need from EUP and what current systems support. In addition to identifying a set of important functionalities that should be included to best support end-users, our findings specifically highlight the needs of social platforms for solving complex problems, and interactivity of programs created with EUP tools to alleviate end-user's concerns about using third-party programs. The Wizard-of-Oz study also shows that future EUP tools should support multi-modal and mixed-initiative interaction for making programming more natural and easy-to-use.

The two studies have the following contributions: 1) identification of unmet needs of end-users of the Web; 2) characterization of non-programmers' mental models describing computational tasks; 3) implications for designing future EUP systems.

3.2. Study 1: End-User Needs on the Web

To better understand end-user needs on the Web, we conducted a semi-structured interview study. The goal was to better understand the challenges that the participants experience, and enhancement ideas that they envision without technical constraints.

Table 1. Occupational background of the participants of study 1

Graduate students		15
	Engineering	8
	Business	4
	Psychology	2
	Education	1
Professionals		12
	IT specialists	8
	Directors and office managers	4
Non-professionals (e.g. homemaker)		8
Total		35

The approach is to qualitatively analyze the participant responses to identify themes that should be considered in the development of future WebEUP systems.

3.2.1. Participants

35 participants (14 males, 21 females) were recruited via a university campus mailing list, social network, and word-of-mouth. They were on average 30.8 years old (SD = 5.1) and had a wide range of occupations as shown in Table 1. Every participant spends at least one hour per day on the Web. 10 out of 35 participants had used at least one programming language, and five participants had created web pages. However, none of them had the experience of end-user programming on the Web. We did not offer any incentive for participation.

3.2.2. Procedure

18 interviews were conducted via a video chat program with shared screen¹¹, while the rest were face-to-face interviews at public areas such as libraries and cafes. I asked participants,

¹¹ Google Hangout (<https://hangouts.google.com/>)

“Show me a couple Web sites that you recently visited, and tell us challenges that you experienced there. If you could hire a team of designers and developers for free, how would you improve the Web sites?”

We recorded (or videotaped for the face-to-face interviews) the participants visiting two to four sites they recently experienced problems. While demonstrating regular tasks on the sites, participants followed the think-aloud protocol. For the challenges they mentioned, we asked them to imagine a team of third-party developers, and to explain to the “team” an enhancement for the Web site. Each interview covered approximately three ($M = 3.02$) sites, and took approximately 20-40 minutes. The study was found to be exempt from IRB review.

3.2.3. Data and Analysis

35 participants demonstrated the use of 92 sites ($M = 2.63$) that included online shopping (24 sites), academic research (17), streaming video (11), news (10), work-related sites (7), forums (5), search engines (5), social network services (4), travel (4), finance (2), review sites (1), job market (1), and weather (1). Note that these frequencies do not correlate the frequency of regular visits but the challenges that our participants experienced. While visiting the sites the participants explained 106 challenges. Every interview video was transcribed, and coded. As an exploratory work, we pursued an iterative analysis approach using a mixture of inductive and deductive coding [8,30]. First, we created a codebook derived from the literature [13,92] and an initial post-interview discussion within the research team. The codebook included types of challenges (lack of relevant information, repetitive operations, poorly-organized information, privacy, security, fake information, bugs), and functionalities required for

doing a wide range of WebEUP tasks (mashup, redesign, automation, social knowledge, sharing, monitoring). To assure high quality and reliable coding, two researchers independently coded ten randomly selected ideas. Analyzing the Inter-Rater Reliability (IRR) of that analysis with Krippendorff's alpha ($\alpha = 0.391$; *total disagreements = 24 out of 255*), we revised the codebook. Then the two researchers coded another ten randomly selected ideas, and achieved a high IRR ($\alpha = 0.919$; *total disagreements = 6 out of 248*). After resolving every disagreement, the first researcher coded the remaining data. Following the guide of thematic analysis [8], we collated the different codes into potential themes, and drew initial thematic maps that represent the relationship between codes and themes. We then reviewed and refined the thematic maps, to make sure that data within a theme was internally coherent, and that different themes were distinguished as clearly as possible. The two following subsections summarize the two groups of themes: challenges that participants experience on the Web, and functionalities of WebEUP for addressing those challenges.

3.2.4. Result: Challenges

Based on the above-described process, four groups of common challenges and enhancement ideas were found which are described in the following sections.

Challenge #1: Untruthful information

While trust is a key element of success in online environments [12], 17 participants reported four kinds of untruthful information on the Internet.

Deceptive ads were reported by three participants. Two of them reported deceptive advertisements that used confusing or untrue promises to mislead their consumers. For

example, P31 gave a poignant example that a local business review site posts unavailable items on the Internet:

“If you're looking for a contractor to work on your home, and other home stuff, [local business review site] shows them with ratings. A few weeks ago I started paying them again for other information, but they have something very frustrating. They have a several page list of mortgage brokers searchable from [search engine]. But when you pay the fee for their service, they have only a fraction of the information. I complained to them, but they have some stories why it is not... Anyways, I canceled my membership without getting my one-month fee refunded.” (P31)

Another participant tried to avoid using an online marketplace because of deceptive ads in it: “I know there are rental houses with good value on [online marketplace], but I do not use it often. There are too many liars on [online marketplace]. Instead I post on [Social Network Service] to get help or recommendations from people that I trust.” (P21)

Links to low-quality content were reported by seven participants. During the interview, two participants clicked broken links to error pages. Five participants reported that they had to spend significant time and effort to find high-quality video links in underground streaming video sites: “At [Underground TV show sites], I have to try every link until I find the first ‘working’ link. By working, I mean the show must be [in] high-resolution, not opening any popup, and most of all as little ads as possible.” (P6) A straightforward solution is to attach quality markers next to the links. However, it is extremely challenging to define a metric of high-quality links that everybody will agree upon.

Virus and Malware was reported by four participants. They were aware of the risks of installing programs downloaded from the Internet, but estimating risks is often inaccurate. For example, two participants stopped using a streaming video site and a third-party plugin worrying about computer viruses, though in fact, those site and plugins were safe.

“I used this streaming link site for a while, but not after a friend of mine told me her computer got infected with malwares from this Web site. I wish I could check how trustworthy the site [is] when using [it].” (P24)

“I have [used popup blocker extension], but am not using [it] now. Those apps have viruses, don't they? I also don't use any extensions.” (P17)

This suggests that end-users may have inaccurate knowledge about the risks of their activities on the Internet. Even though third-party programs provide terms and conditions, and permission requests, users are often ‘trained’ to give permission to popular apps [11] as stated by P27: *“If the site is important to me, I just press the 'agree' button without reading.”*

Opinion spam was reported by four participants. While social ratings and consumer reviews are conventional ways to see feedback on products and information, the reliability of the feedback is often questionable [66]. Four participants reported concerns about opinion spam – inappropriate or fraudulent reviews created by hired people. For example, P31 reflected, *“I saw that some sites have certificates, but they were on their own sites. So, who knows what they're gonna do with that information? [...] For example, I had a terrible experience with a company that I hired for a kitchen*

sealing repair, even though they had an A+ rating on [a local business review site].”

P27 also expressed concerns about fake reviews, *“ratings are somewhat helpful. However, I cannot fully trust them especially when they have 5 star ratings - they might have asked their friends and families to give them high ratings.”* Similar to deceptive ads, opinion spam is a gateway to serious financial risks such as Nigerian scams [12], but there is no simple way to estimate the risk.

Summary. In order to deal with untruthful information, participants would look for more trustworthy alternatives. For example, P21 used a social network service instead of online marketplaces. If participants could not find an alternative source, they would assess the risks and benefits of using the untruthful information, and decide either to give up the task or to take the risk, as P31 said, *“I don’t believe everything on the Internet. But sometimes I have no other choices than to try it with caution.”* The remaining issue is that estimating the risk of untruthful information is often quite difficult.

Challenge #2: Cognitive Distraction

Most participants reported cognitive distractions that make information on the Web hard to understand. We identified four types of cognitive distractions as listed below.

Abrupt design changes were reported by three participants. Websites are occasionally redesigned – from a minor tune-up to a complete overhaul – for good reason. However, it often undermines the prior knowledge of its users, and makes the sites navigation difficult. For example, P22 could not find her favorite menu item because *“the library recently changed its design, making it much harder to find the menu.”* Since she found a button for switching back to the classic design at the end, she didn’t take advantage of new features in the updated design. P24 shared a similar story:

“One day Facebook suddenly changed the timeline to show this double column view. That was very annoying.”

Annoying advertisements were reported by 30 participants. We found that the degree of cognitive distraction varies across different types of ads. For example, ads with dynamic behavior are much more annoying than static banner ads:

“There are popup ads that cover the content and follow your scrolling. Although they usually have very small 'X' or 'Close' buttons, I often miss-click the popup to open the Web page. That's pretty annoying.” (P17)

This finding is consistent with prior research that found display ads with excessive animation impair user's accuracy on cognitive tasks [22]. 16 participants were using browser extensions (e.g. Chrome AdBlock¹²) to automatically remove ads. However, one participant had stopped using it for security and usability issues:

“I have, but am not using [AdBlock] now. Those apps have viruses, don't they? [...] They would be very useful in the beginning, however they also restrict in many ways. For example, the extension sometimes automatically block crucial popup windows. So I ended up manually pressing 'X' buttons.” (P17)

Unintuitive tasks. Six participants reported that several Websites are hard to use. For example, to create a new album in Facebook, users are required to upload pictures first. This task model clearly did not match a participant's mental model: *“I tried to create a new photo album. But I could not find a way to create a new album without*

¹² <http://goo.gl/rA6sdC>

uploading a picture. That was a very annoying experience.” (P18). Another user reported a similar issue of not being able to create a new contact after searching in a mailing list:

“I’m adding a new person to the contact database. I should first search the last name in order not to put duplicate entry. If the name does not exist, it simply shows [0 result found]. Obviously I want to add a new entry, but there’s no button for that. That bugs me a lot, because I have to get back to the previous page and type the name again.” (P16)

Websites with unintuitive navigational structures would require users to do many repetitive trial-and-errors.

“When preparing to visit a touristic place, I look for entrance fee, direction, and other basic information from their official sites. However, some sites have that information deep in their menu structure, so I had to spend much time finding them. I wish those information were summarized and shown in one page. Sometimes it’s hard to find useful images for campsites or cabins. For example, I want to see the image of bathroom, but people upload pictures of fish they caught.” (P33)

Information overload. Five participants reported that excessive and irrelevant information prevents them from understanding the main things that they care about. For example, P22 was disappointed at blog posts full of irrelevant information: *“I was searching for tips to clean my computer. However, most blog posts have very long explanations of why I should keep computers clean without telling how to clean it till the end.” (P22)*

A long list without effective filtering also causes information overload as P2 stated:

“I want these conferences filtered by deadline, for example, showing conferences whose deadlines are at least 1-month from now. Also, if possible, the filter can look at descriptions of each venue and choose ones containing at least three relevant keywords.”

A simple enhancement to solve this problem is to remove unnecessary, excessive information, which is often very hard to decide. For example, P27 criticized an online shopping site for having a lot of unnecessary and irrelevant information. However, when evaluating usefulness of individual components, she became more vigilant, and stressed that her opinions are personal and depending on her current situation.

“I would remove these promoted products on the side bar. However, if these promotions were relevant to my current interest, I would keep them. [...] Shopping cart and Personal coupon box can be useful later. [...] I don't need extra information about secured payment, getting products at the shop, or printing receipts.” (P27)

To enhance websites with an over-abundance of information, participants envisioned creative scenarios including interactivity and design details. For example, P2 proposed to add a custom filter for a long list. P26 wanted to have the personalized summary at the top of a long document with a pop-up window for important information:

“I do not read every Terms and Condition agreement. It's too long and mostly irrelevant. However, it would be useful if hidden charges or tricky conditions were highlighted. I think critical information such as hidden charges can be shown in a pop-up

window. It would be best the most important summary is shown at the top, because I could just click 'yes' without scrolling it down.”
(P26)

Challenge #3: Repetitive Operations

Participants reported tedious and repetitive operations on the Web. Based on them, we identified three common reasons for repetitive operations.

Unsupported Tasks. Seven participants wanted to automate repetitive tasks. Efficient repeating of some of the tasks is unsupported by the websites. For example, four participants wanted to automate simple interactions such as downloading multiple files or clicking a range of checkboxes with a single click.

“[At an academic library], I click the "Save to Binder" button, then select a binder from the drop-down in a new window. Then I click the "save" button then the "done" button, then close the window. It's really annoying to do it over and over. It would be great to create a "save this!" button.” (P4)

Three participants wanted to automate filling the forms of personal / credit card information.

Information from multiple sources. Reported by 20 participants, integrating information from multiple sources is a common practice on the Web [93]. End-users switch between browser tabs to compare information repeatedly, but it can be time consuming since it requires short-term memory to compare information on tabs that are not simultaneously visible. 17 participants wanted to save their time and effort by integrating information across multiple sources. For example, P33 told, *“I often search for videos on YouTube for baby diapers or other things to wear because those videos*

are very helpful to understand usage of products.” Similarly, four participants wanted to integrate course schedule page with extra information available such as student reviews, lecture slides, and reading lists.

Time-sensitive Information. Five participants reported that they regularly check time-sensitive information such as price (3), hot deals (1), second-hand products (1), and other notifications (1). Using price trends as an example, three participants envisioned a complex service that automatically archives price information retrieved from multiple sites, visualizes the price data as timeline graph, and sends email / mobile notifications when the price drops:

“I can imagine that program or Web site will be able to grab information, especially prices from various malls, and compare it automatically. [...] It will also say ‘this is the lowest price for recent three months.’ so that I don't have to visit Amazon and Newegg everyday. [...] I want it to send me email alerts - saying ‘Hey, based on your recent search history on the Canon G15, we found these new deals and prices. It's the lowest price in the last month.’ ” (P21)

“[She opened CamelCamelCamel.com] If I want to buy a bread machine, I search and choose one model. Here the graph shows the price trend of the model. I can make a decision on whether I should buy or wait. Unfortunately, this site only shows products from Amazon.com.” (P33)

Challenge #4: Privacy

Privacy did not come up much, but one participant (P24) expressed strong negative opinions about the way that a social networking service handles her data:

“[At a social network service] a friend of mine told me that if I 'like' her photos or put comment on them, others will be able to see it even if the photos are private. [...] Here's another example that I don't like about [the SNS]. One day I uploaded a family photo, and my family-in-law shared those photos. That's totally fine. However, the problem began when friends of my family-in-law started liking and commenting on my family photos. I received a lot of notifications of those activities by people I do not know at all. I felt a little scared.” (P24)

As another example of privacy issues, P24 believed that her browser tracks her activity history, and shared it with online advertisement companies without her permission, because banner ads on other Web pages show ads related to her previous activity.

3.2.5. Potential Functionality of Web Enhancements

Based on the challenges of the previous section, here we present functionality that we believe future WebEUP systems should consider. The functionality has seven categories: *Modify*, *Compute*, *Interactivity*, *Gather*, *Automate*, *Store* and *Notify*. To our knowledge, *Interactivity*, *Store*, and *Notify* among them were not supported by existing EUP systems for the Web.

Modify. Modification of existing web pages is the most commonly required functionality for 66 out of 109 enhancements. Examples include attaching new DOM elements to the original pages (31 enhancements), removing or temporarily hiding unnecessary elements (15 enhancements), and highlighting information of interest by changing font size, color, or position (5 enhancements). Modification often involves adding new interactive behavior of Web sites (8 enhancements). Existing WebEUP

tools support a wide range of modification such as removing unwanted DOM elements [7], and attaching new DOM elements or interactive behavior to existing elements [78].

Compute. 29 enhancements require a variety of data transformation: filtering elements by user-specified criteria (13 enhancements), extracting specific information from text documents (9), and arithmetic operations (7). While computation is a fundamental part of programming languages, existing EUP systems support it in varying degrees. For example, scripting languages [98] offer an extensive set of language constructs such as general-purpose languages (e.g. JavaScript). Data integration systems [80,87] focus on handling large amount of semi-structured text input, but provide less support on numerical operations. Systems for automated browsing [47,59] provide few language constructs for computation.

Interactivity. 29 enhancements would need interactive components that address the dynamic needs of users. For example, 13 enhancements include triggering buttons, because users wanted to make use of them *in-situ*. Eight enhancements show previews of changes it will make on the original sites so that users can choose among them. Enhancements often require users to configure options such as search keywords, filtering criteria, specific DOM elements based on their information needs (8 enhancements). WebEUP tools often employ predefined interactive components such as buttons and preview widgets [78]. However, none of them enable users to create their own interactivity.

Gather. 18 enhancements gather information from either the current domain (9 enhancements) or external sources (5 enhancements). One example use of information of the current domain is to preview linked resources without clicking, as P5 stated, “*At*

various cosmetic malls, I wish the main listing page showed detailed direction on how to use the products.” In contrast, participants wanted to gather information from external sources that current sites are missing. Information gathering is supported by mashup tools [15,82,87].

Automate. 15 enhancements automate repetitive tasks that include filling in input forms (4 enhancements), downloading multiple images and files (4), page navigation (3), clicking a series of buttons, checkboxes, and links (3), and keyword search (1). Existing WebEUP tools such as CoScripter [47] and Inky [60] support automating repetitive tasks.

Store. 14 enhancements store three types of data while being used. The first type relates to user’s activities such as filling input forms, page navigation, and job applications found in five enhancements. The second type is temporal information periodically gathered from designated sources such as online shopping malls, or ticketing sites found in five enhancements. The last is bookmarks of online resources such as news articles, blog posts, or streaming videos found in four enhancements. Existing WebEUP systems such as CoScripter [5] often provide public repositories for scripts, but none of them allow end-users to create custom storage of usage data.

Notify. Eight enhancements send notifications to users via emails (7 enhancements) or SMS messages (1), periodically or when user-specified events occur. To our knowledge, no existing WebEUP tool supports notification.

3.2.6. Design Implications

Based on the challenges and the potential functionality of Web enhancements, we discuss two design implications for future WebEUP systems and designing Web sites in general.

Social Platform beyond Technical Support

Traditional WebEUP systems focus on lowering the technical barrier of Web programming. For example, mashup tools enable users to integrate information from multiple pages with just a few clicks. Automation tools allow users to create macro scripts through demonstration. Despite the advantage of those technical aids, we noted a few enhancement ideas require domain knowledge of multiple users who have the same information needs. For instance, when end-users want to integrate additional information with original pages, the key question is where the additional information can be found. When users want to focus on an important part of a long text, the key is which part of the text previous visitors found useful (similar to Amazon Kindle's "Popular Highlights" feature.) An example of how a social platform could address the *untruthful information* issue follows. An end-user programmer creates and deploys an enhancement that attaches an interactive component (e.g. button for rating individual hyperlinks) to the original page. Users who have installed the enhancement would use the new component to provide their knowledge (e.g. quality of the linked resources), which will be saved in the enhancement's database. As more data is collected, the enhancement will become more powerful.

To enable end-users to build social platforms in the aforementioned scenario, future WebEUP systems need two functionalities. First, end-user programmers should

be able to create and attach interactive components that collect knowledge and feedback from users. Second, end-user programmers should be able to set up centralized servers that communicate with individual enhancements running in each user's browser, and store collected information. To our knowledge, no prior WebEUP system has fully supported these functionalities for social platforms. However, there are certainly custom solutions of this type that are commonly used such as, for example, Turkopticon¹³ that helps web workers using Amazon Mechanical Turk rate job creators.

Alleviate the Risk of Using Enhancements

According to the *attention investment* framework [4], end-users would decide whether to use an enhancement or not as a function of perceived benefit versus cost. Even though our participants assumed no development costs, we could identify the following concerns about risks of using enhancements.

Uncertain needs. Our participants often had concerns about the dynamic and uncertain nature of their needs and situation. For example, P27 found advertisements on an online shopping site to be annoying, but did not remove the advertisements because of their potential usefulness in the future. WebEUP systems should be able to support interactivity so that users can change configurations or make decisions whenever their needs change. Otherwise end-users will be forced to stop using it, as P26 and P17 did with non-interactive pop-up blockers.

Breaking the original design. Enhancement developers should try to minimize unnecessary change of the original site. Two participants expressed concerns about

¹³ <https://turkopticon.ucsd.edu/>

breaking the original site's design and functionality: *"I think the best part of Craigslist is its simplicity. I might have seen the filters, but did not bother setting them every time I visit this site."* (P20)

Privacy and Security. End-users have significant privacy and security concerns about installing extra software, especially those developed by third-party programmers. Ironically, we observed end-users rarely read legal documents, and are trained to give permissions to popular apps. Future work should confront these practical concerns and design how to communicate potential risks and treatments.

Summary of Design Implications

The seven categories of enhancements can be useful to web site designers as they think about what a wide range of users might want. There is another potential of more directly benefiting from end-user modifications to web sites. Actual enhancements made by end-users could provide valuable feedback for designers of the sites if those desires were expressed via use of a WebEUP tool. For example, designers could learn what kind of information users consider to be untruthful by learning about user feedback on specific information. Repetitive operations could be observed by seeing what modifications users make, etc. Nevertheless, those feedbacks cannot replace WebEUP, as designers and users often have conflicting interests. For instance, designers may not agree to remove advertisements that end-users find annoying since they provide revenue. Some ideas may be useful for specific user groups but not for everyone, and so are not worth pursuing. Ideally, designers should consider providing hooks or APIs that enable end-users to build robust, high-quality enhancements.

3.2.7. Limitations

We made several simplifying assumptions that limit the scope of our findings. First, 35 participants of the interview study were not large enough to represent the entire population of potential end-user programmers. In order to extend the generalizability of the findings, an online survey would be an appropriate method. Second, around half of our participants have non-technical backgrounds, which is an unusual characteristic of end-user programmers. Some of the challenges and solutions they shared could be different from end-user programmers who usually have technical knowledge. Third, in order to minimize technical bias, the semi-structured interview did not provide any technical constraints. Therefore, participants imagined EUP solutions without considering the time and effort of development.

3.3. STUDY 2: NON-PROGRAMMERS MENTAL MODEL OF COMPUTATIONAL TASKS

Programming is difficult to learn since its fundamental structure (e.g. looping, if-then conditional, and variable referencing) is not familiar or natural for non-programmers [67]. Understanding non-programmer's mindset is an important step to develop an easy-to-learn programming environment. This second study builds on the first by examining how non-programmers naturally describe computational tasks common to the WebEUP enhancements described in the first study. The findings suggest both design implications and open-ended research questions for future EUP systems.

3.3.1. Participants

The study was conducted with 13 participants, including five males and eight females, average 33.3 years old ($SD = 5.86$) with varying occupations as summarized in Table

Table 2. Occupational background of the participants

Graduate students		6
	Engineering	3
	Business	2
	Education	1
Professionals		3
	IT specialists	2
	Directors and office managers	1
Non-professionals (e.g. homemaker)		3
Total		13

2. All of the participants were experienced computer users, but they all said that they had not programmed before. The participants were recruited by the university mailing list that we used in the first study. They received no compensation for participation in this study.

3.3.2. Method

The study aims to characterize how non-programmers naturally describe complex tasks without being biased by specific language constructs or interactive components. We employed the Wizard-of-Oz technique [95] where the interviewer acted as a hypothetical computer agent that could understand the non-programmer’s verbal statements, behavioral signals (e.g. page navigation, mouse click), gestures, and drawing on scratch paper, and help them through conversational dialogue. The computer agent (called “computer” from here on) followed the rules listed below.

1. The computer can understand all the literal meaning of participants’ instruction, gestures, and drawings. However, the computer cannot automatically infer any semantic meaning of the task or the material. For example, a rental posting “*4 Bedrooms 3 Lvl Townhome \$1650 / 4br*” is just a line of text to the computer.

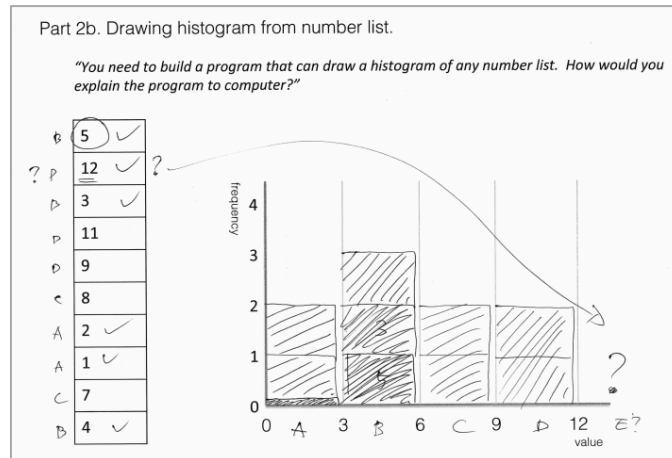


Figure 15. Participants were asked to explain how to draw a histogram of the numbers in the table. In this example, the participant gave histogram bins different codes (A-D), and marked each number with the codes. Since the participant could not put 12 into any bin, he marked the number with question mark and a line that points a missing bin.

- The computer can perceive a pattern from participant's repeated examples and demonstration. For example, if a participant counted numbers within a range 1-3 in a table, the computer asks the participants "Are you counting numbers that are within a specific range?"
- The computer can execute the participant's instruction only if it is clearly specified without ambiguity. Otherwise the computer asks for additional information to resolve it through conversational dialogue like below:

Programmer: *Delete houses with fewer than three bedrooms.*

Computer: *Please tell me more about 'houses with fewer than three bedrooms'. Which part of the page is relevant?*

When the programmer demonstrates a set of examples, the computer will suggest a generalizing statement like below:

Programmer: *Delete this one because it contains 3br.*

Computer: *Do you want me to delete every line that has 3br?*

Table 3. In Task 2, the participants were asked to create a filter than removes houses with less than three bedrooms among housing rental posts scraped from Craigslist.com.

“You want to create a filter that removes houses having less than 3 bedrooms. How would you explain it to the computer?”

Brand New Townhome! \$2200 / 3br - 1948ft² - (Clarksburg)

Lanham 2/1 new deck \$1050 / 1818ft² - (Lanham)

4 Bedrooms 3 Lvl Townhome \$1650 / 4br - (MD)

823 Comer Square Bel Air, MD 21014 \$1675 / studio

... (6 more)...

A sheet of paper containing basic instruction was provided, and the participants could draw or write anything on the paper as shown in Figure 15 and Figure 16.

Task 1. Drawing Histogram

Given a sheet of paper containing a blank histogram and 10 random numbers between 0 and 12 (see Figure 1), the participants were asked to explain the computer how to draw a histogram of the numbers. The blank histogram has four bins (0~3, 3~6, 6~9, and 9~12). The purpose of this task was to observe how non-programmers perform: (1) common data-processing operations (e.g. iteration, filtering, and counting), and (2) visualize numeric data by examples and demonstration.

Part 2a. Removing houses that have less than 3 bedrooms.

"You want to create a filter that removes houses that have less than 3 bedrooms. How would you explain it to computer?"

of br
BEDROOM

- Brand New Townhome! \$2200 / 3br - 1948ft² - (Clarksburg) 3
- Lanham 2/1 new deck \$1050 / 1818ft² - (Lanham) 2 → X
- Se Renta/ 4 Bedrooms ~~3br~~ Townhome \$1650 / 4br - (Camp Springs MD) 4
- 823 Comer Square Bel Air, MD 21014 \$1675 (studio - (Bel Air, MD) 1 → X
- OPEN HOUSE! 7/27 @ 4PM-Magnificent Marlon Townhome Townhouse \$1550 / 1br - 1710ft² - (Upper Marlboro, MD) 1 → X
- 5/2 attached garage \$1325 / 2018ft² - (Greenbelt) 0 → X
- Se Renta Basement \$1100 / 2br - (Maryland, Silver Spring) 2 → X
- SUPER CUTE 1 BEDROOM APARTMENT \$1000 / studio - (College Park, MD) 1 → X
- TH for Rent-With Projector & screen Setup!! \$1890 / 3br - (Gaithersburg, MD) 3
- Greenbelt 3/2 chance of a lifetime \$1325 / 1398ft² - (Greenbelt) 3

Figure 16. In this example of Task 2, the participant used scribbles along with verbal statements. For example, the participant wrote variations of keywords that indicate “bedroom” used in the list. He/she also circled and underlined the number of rooms in each title to demonstrate the text extraction logic, crossed out titles that did not meet the criteria, and drew arrows from houses to empty slots in the list.

Task 2. Custom Filter

We prepared 10 rental postings in Table 3 copied from an online marketplace¹⁴. The participants were asked to create a program that removes houses having fewer than 3 bedrooms. The program consists of three components: (1) extracting text that represents the number of bedrooms in each post (e.g. “3br(s)”, “3bedroom(s)”, “3 BEDROOMS”, “3/2”), (2) a conditional logic for filtering posts with less than three bedrooms, and (3) removing / hiding the filtered houses. The purpose of the task is to

¹⁴ Craigslist.com

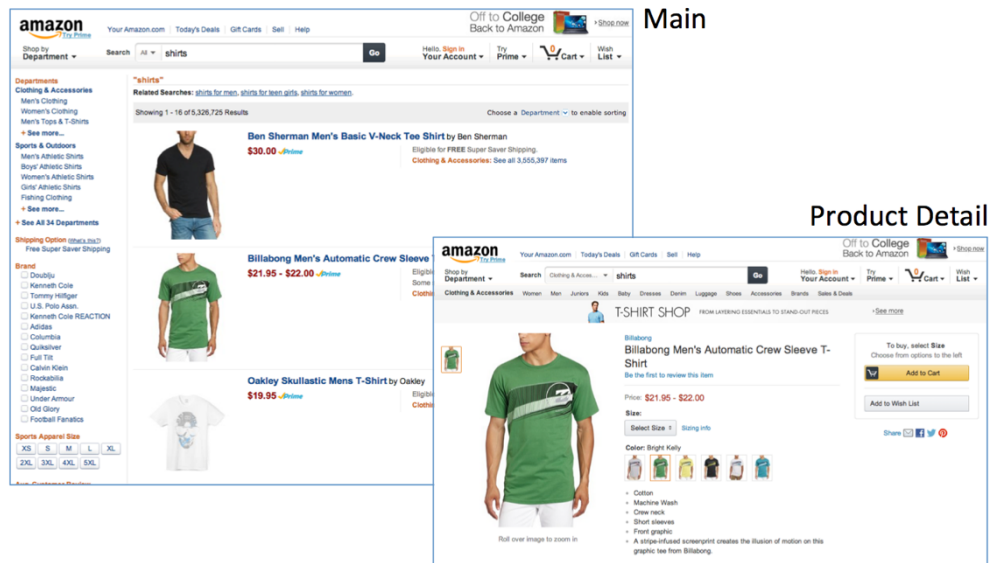


Figure 17. In Task 3, participants were asked to describe a simple Mashup program that shows available colors of each individual product in the Main page (top left) extracted from the Product Detail (bottom right) page.

observe how non-programmers decompose a big task into sub-tasks, specify extraction queries, and refer temporary variables such as sub-strings and selected postings.

Task 3. Mash Up

At Amazon.com, each product has different options (e.g. available colors and sizes) that are shown in the product detail page. The participants are asked to create a program that extracts the available colors from detail pages, and attaches to the product listing. The purpose of the task is to understand how non-programmers would describe copy operations across multiple pages, and event handling.

3.3.3. Procedure

Each session began with a brief interview about the participant's programming experience and occupational background. The interviewer introduced the Wizard-of-

Oz method, and gave an exercise task – ordering the interviewer (acting the hypothetical computer agent) to move a cup to another corner of the table. After participants said they fully understood the concept of the hypothetical computer agent, we started the actual study by introducing the three scenarios in a randomized order. For each scenario, participants were asked to explain the task to the “computer”. Participants were allowed to finish or to give up a task at any point.

3.3.4. Data and Analysis

The entire session was video recorded, and transcribed for qualitative analysis. The transcript of each task consists of a sequence of conversational dialogue between the participant and the interviewer, finger and mouse pointing gestures, scribbles on the paper (Figure 16 and Figure 17; only for T1 and T2), and page scroll and mouse events in the browser (only for Task 3). To analyze the transcript, the first author created the initial codebook derived from the literature [67] and an initial post-interview discussion within the research team. The codebook included how the participants described and what challenges they experienced. While repeating the coding process, a few categories emerged: programming styles, imperative commands, ambiguities, and multi-modal intent.

3.3.5. Findings

In this section we characterize how non-programmers describe computational tasks. Participants were allowed to stop at any moment, but all of them could eventually complete tasks with the computer’s help. Each task took an average of 415.3 seconds ($SD = 217.4$). We did not observe any fatigue effect. Since participants had very limited understanding of the computer at the beginning, most of their initial explanations were

not very informative. Thus the computer asked for further information as the examples below.

(Task 1)

P12: Wouldn't computers draw graph when numbers are assigned? I'm asking because I have no idea.

P11: Find the numbers, and draw them at the first bin.

Computer: How can I draw them?

P11: What should I tell? Color?

(Task 2. Custom Filter)

P5: First, I scan the list with my eyes and exclude them. They clearly stand out.

Computer: How do they stand out?

P8: I'd order, "Exclude houses with one or two bedrooms."

Computer: How can I know the number of bedrooms?

(Task 3. Mash Up)

P11: I'd ask computer to show available colors of this Columbia shirt.

Computer: Where can I get available colors?

Natural language tends to be underspecified and ambiguous [67]. We frequently observed that our participants skipped mentioning essential information. For example, most participants did not specify how to iterate multiple elements in list. They instead demonstrated handling the first item, and expected the computer to automatically repeat the same process for the rest of the items. They did not refer to objects by names as programmers use variables. However, they referred to previously mentioned objects by their actual values (underlined in the following example), as P20 said, "*In this next column, we need items going 6, 7, and 8. So please find those 6, 7, 8, and draw bar in this column.*" They also used pronouns (e.g. "*Remove them*"), data type (e.g. "*Attach colors*"), and gestures (e.g. "*Paste them here*"). While loops and variable referencing

are core concepts of programming languages, our findings suggest that non-programmers would find them unnecessary or even unnatural. We will discuss the issue further with design implications for future EUP systems in the discussion section.

Through conversational dialogues, participants figured out what information the computer requires and how to explain. We found several characteristics of how non-programmers explain computational tasks as listed below.

Explaining with rules and examples was used by 9 of 13 participants. When participants explained rules first, the following examples usually provided information that the rules were potentially missing. For example, while drawing a histogram for Task 1, P4 stated a rule, “*Determine which bin each number is in*”, followed by an example, “*If the number is one (pointing the first item in the table), then count up this bin (pointing the first bin in the histogram).*” Participants also provided examples first, and then explained the rules. P10 doing Task 1 gave all the numbers (0, 1, and 2) for the first bin, “*For here (pointing the first column) we need 0, 1, and 2*”, and then explained the range of those numbers, “*Find numbers including zero, smaller than two.*” Traditional programming languages rarely allow example-based programming. Although EUP systems often support Programming-by-Example (PBE) techniques, they do not allow this pattern – combining rules and examples to describe individual functional elements.

Elaborating general statement through iteration was observed for every participant. Initial explanations of tasks were usually top-level actions (e.g. draw bars, remove houses, attach pictures) that contained a variety of ambiguities; but participants then iteratively elaborated the statements by adding more details. For example, P1

doing T3 described the top-level action, “*Attach pictures here.*” Then he elaborated where the pictures were taken from, “*Attach pictures from the pages.*” He kept on elaborating what the pages are and how to extract pictures from the pages. For T 1, as another example, P14 told the computer, “*Draw a graph.*” She then rephrased the statement with more details, “*Draw a graph to number 2.*” This pattern is far from traditional programming languages that support users to create statements in the order of their execution.

Multi-modal expressions including gestures and scribbles were frequently used by all participants. While verbal statements were still the central part of explanation, they used gestures along with pronouns (e.g. “*Count these*”, “*Put them here*”), and scribbles to supplement verbal statements like an example in Figure 2. While multi-modal expressions seem to be natural and effective for non-programmers, traditional programming environments rarely support them.

Rationales are not direct instructions for the computer. However, we consistently observed participants explaining rationales. For example, P6 doing T3 explained why she chose to attach small color chips rather than larger images, “*While we can show images, which would be quite complex, I'd want you to do use color boxes.*” P13 also explained rationale of her scribbles on the sheet of T1, “*We can also secretly write number here (center of each cell) to remember, so track for afterward so we didn't make any mistake.*”

3.3.6. Implications

This study provides characteristics of non-programmers explaining how they would solve computational tasks. Given that traditional programming environments do not

fully support the way these participants conceptualized their solutions, we discuss the implications for the design of multi-modal and mixed-initiative approaches for making end-user programming more natural and easy-to-use for these users. Our recommendations are to:

Allow end-users to express ideas with rules, examples, gesture, and rationales.

Traditional programming environments mostly support single programming styles: imperative, declarative, or example/demonstration-based. However, as seen in the user study, end-users express ideas via combinations of rules, example, and rationales.

Support iterative refinement of programs. End-users may not be able to provide complete information of the programs they want. Instead, they would start with quick and brief description of task outlines, goals, or solutions that handle only a subset of the potential scenarios. They then iteratively refine it by adding more rules and examples. In order to support this iterative refinement, future EUP tools should allow users to sketch programs with missing details, and guide them to fill in.

Support mixed-initiative interaction to disambiguate user intent. To guide non-programmers to explain essential information such as loops and variable referencing, our study employed conversational dialogue (as explained in Section 5.2) between participants and the computer. For example, when participants gave incomplete statements (e.g. demonstration for the first item), the computer asked them for additional information (“*What would you like to do for the rest items?*”) or confirmation (e.g. “*Do you want to do the same for the rest items?*”) Likewise, future EUP tools should incorporate mixed-initiative interaction to help end-users express unambiguous

statements; although it is an open-ended research question how the computer and end-users have mutual understanding.

3.3.7. Limitations

We made several simplifying assumptions that limit the scope of our findings. First, the computer followed three informal rules, which may not be specific enough to design working a system. A formal set of rules would make the Wizard-of-Oz study stronger. Second, participants could not review or test programs they built, which is uncommon for most programming environments. Third, the three tasks do not represent a full spectrum of computational tasks. However, we believe that even this narrow analysis provided useful insights for designing natural and intuitive EUP systems. To address these limitations, a follow-up study should employ an actual, interactive EUP system that presents and tests solutions that address the challenges and the implications of this study.

3.3.8. Conclusion

This chapter reports two formative studies that extend the understanding of end-users' needs and mental models. The first study, a semi-structured interview, explores challenges that end-users daily experience, and suggests seven functionalities of future EUP systems. The second, a Wizard-of-Oz study, demonstrates how non-programmers explain common computational tasks and provides design implications for more natural programming environments. Based on these findings, we designed VESPY, an interactive EUP system VESPY, which is presented in the following chapter.

Chapter 4: VESPY: A Visual Environment for Symbiotic Programming

4.1. Introduction

In the previous chapter, we reported end-users' needs for enhancing the web, and how they express programming intent. Based on that, we developed VESPY (Visual Environment for Symbiotic Programming), an end-user programming tool that enables amateur programmers to build interactive Web enhancements. This chapter presents VESPY's user interface, domain-specific language, and PBE engine. While most end-user programming systems for the Web (WebEUP) focus on specific application domains (e.g. extracting data from pages, automation, information mashup), VESPY covers much wider range of enhancements by letting users orchestrate common functionalities (e.g. extraction, transformation, integration, automation, customization, and interactivity). Our approach is to interleave visual programming techniques with programming-by-example (PBE) so that users decompose complex tasks into tractable modules (with the grid UI), and generate solutions for each module by providing input and output examples to the PBE engine. To demonstrate the versatility of VESPY, we present four example enhancements. Finally, a preliminary user study shows that PBE helps users do complex tasks with efficiency, but simple tasks are more suitable for direct specification. We also observed that participants experienced usability issues and made a variety of mistakes.

4.2. Design Iteration

The development of VESPY was a long iterative process, taking 1.5 years to explore various ways to accommodate visual programming and PBE. In this chapter we describe design prototypes of VESPY UI with our consideration of challenges and rationales.

4.2.1. Version 1: Spreadsheet

The first UI design (Figure 15) was matrix-based, motivated by the generality and understandability of standard spreadsheets. Each column represented a list of homogeneous values, calculated by the same operation (e.g. DOM elements extracted with a single query). When the “application” represented by the spreadsheet is executed, VESPY executes an operation assigned to the leftmost column to update its values, and then repeats the same process for columns to the right. The blue arrows between adjacent columns represent operations that calculate the next column from the previous columns on the left side. Note that PBE can suggest multiple operations for a single column, illustrated as multiple blue arrows next to the first column. The numbers in the blue arrows are number of values it calculates. For example, in Figure 15, the first column has a single value, which represent the current page, before executing the program. As the entire program is executed, the second column’s operation extracts 100 DOM elements of each row (p.row). The third column extracts 100 URLs from each row. The fourth column loads pages of the URLs, and then gets images in the fifth column.

We informally assessed strength and weakness of the spreadsheet design. We felt that the spreadsheet approach was an effective, intuitive representation of values, and




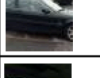


D original	100 100 15 1 1	D [p.row]	URL [http://washingtondc.craigslist.org/mld/cto/3600394199.html]	D [2004 navigator 4x4]			
		D [p.row]	URL [http://washingtondc.craigslist.org/mld/cto/3600394199.html]	D [1999 LINCOLN NAVIGATOR]			
		D [p.row]	URL [http://washingtondc.craigslist.org/mld/cto/3600394199.html]	D [Honda Accord LX]			
		D [p.row]	URL [http://washingtondc.craigslist.org/mld/cto/3600394199.html]	D [2004 navigator 4x4]			
		D [p.row]	URL [http://washingtondc.craigslist.org/mld/cto/3600394199.html]	D [05 AUDI A4 QUATTRO 1.8...]			
		D [p.row]	URL [http://washingtondc.craigslist.org/mld/cto/3600394199.html]	D [1996 HONDA ACCORD SW EX...]			

Figure 18. The 1st design of VESPY UI looks like a spreadsheet. Each column represents a list of values. The green arrows represent operations that calculate the next column.

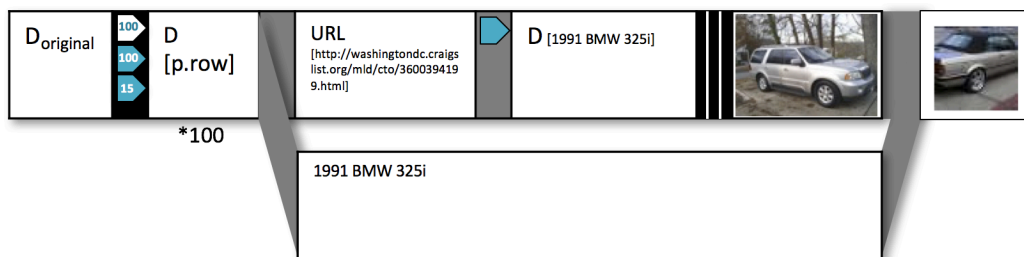


Figure 19. The 2nd design of VESPY UI. Widgets that contain small spreadsheets represent complex program structure such as branching and merging.

the linear flow of data was straight forward. However, we decided to try an alternative design, because the spreadsheet metaphor was too limiting. We were not able to find a way to adapt it to support complex control flow such as branches or nested loops.



Figure 20. The 3rd design of VESPY UI is optimized for showing the description of every operation.

4.2.2. Version 2: Graph of Multiple Spreadsheets

The 2nd version was designed to represent more complex, non-linear data flow such as branching, merging, and executing other modules. Although existing dataflow programming environments accommodate graph structure (as discussed in the Related work section), they do not consider how to support interaction, such as reviewing how individual values change along the control flow, or providing input and output examples. We wanted our UI to represent control flow and data flow at the same time. For example, Figure 19 illustrates a data flow with a branching and a merging for finding a specific item in the list. While the 2nd design is clearly more versatile than the 1st one, we felt the 2nd version is still weak at presenting what the entire program is about.

4.2.3. Version 3: List of Operations

While designing the 3rd revision, we changed our focus on describing operations rather than values. As illustrated in Figure 20, items in the vertical list (e.g. “Pick elements”,

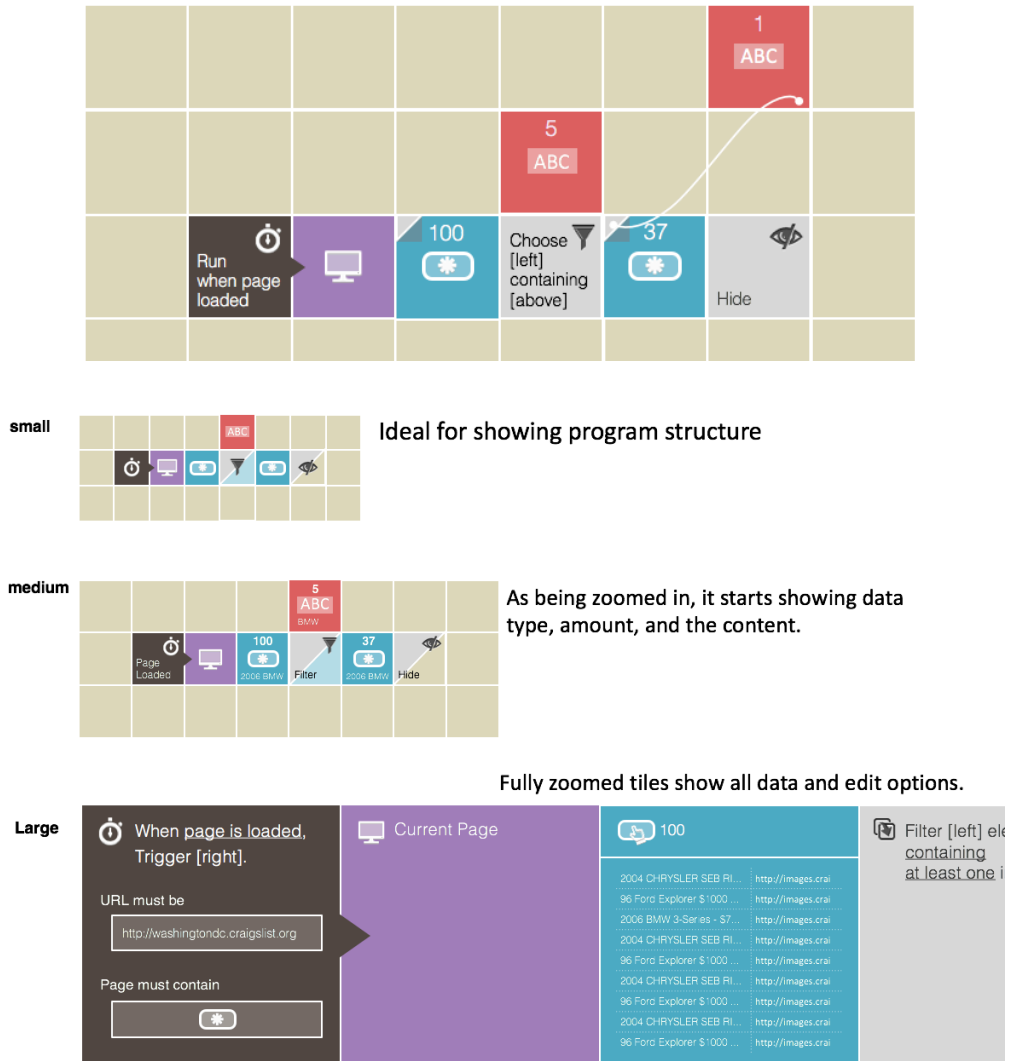


Figure 21. The 4th design of VESPY UI employs a 2D grid and a semantic zoom feature.

“Inspect links”) represent steps, which will be executed in sequence when users run the program. The vertical list works like an accordion, where users can fold / unfold items

to see details. To add an operation, users click at the bottom of the list. To insert an operation, users click between two items. To infer commands for a step, users can click the step to unfold its input (above the step) and output (below the step) value tables. As users type examples in the value tables, the PBE engine recommends corresponding commands that users can click to confirm. The design is relative compact – taking only small part of the entire screen, thus it would be comfortable to use alongside with the web pages. However, the design also has many limitations. For example, it would be extremely difficult to visualize complex non-linear programs that contain branching and merging. Moreover, the design was not flexible to accommodate large amount of input and output examples. In sum, we felt the 3rd design was too simplistic to support symbiotic interaction.

4.2.4. Version 4: Grid and Semantic Zoom

The 4th revision was designed to better visualize non-linear control flow and data at the same time. Each cell on the 2D grid represent an operation and calculated values. By default, each cell accepts input data from the left and the above, and thus data flows top-to-bottom, or left-to-right. Users could manually modify each node to take input from arbitrary node as well. For example, the program in Figure 21 starts from the leftmost cell, “Run when page loaded”, and triggers the cell on the right side. The cell in the middle, “Choose [left] containing [above]”, gets input from left and above. After filtering, 37 items that contains the keyword becomes invisible by the rightmost cell. We liked the grid UI, not only because it represents non-linear flow of execution, but also provides extra flexibility for program decomposition. For example, to solve

complex problems, users can create sets of connected operations, and connect them later.

The biggest design challenge was to find a balanced representation for both operations and values of each node. The 4th design addresses the requirement with semantic zoom. As users click a cell, the system would zoom into the selected nodes and show more details, such as full description of the operation, and current values. Although semantic zoom sounded brilliant, we soon realized that it also has a major

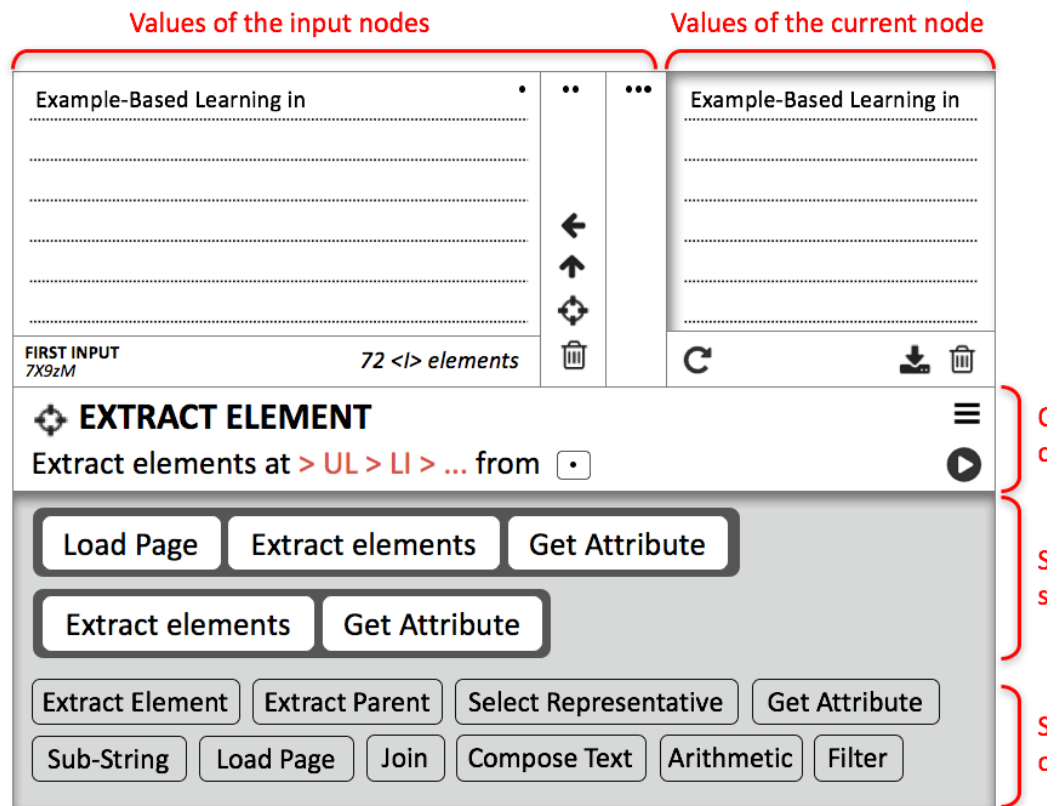


Figure 22. The 5th design of VESPY UI includes a pop-up panel that shows details of the currently selected node. The top row represents values of the input nodes and the current node. The middle row explains what operation is assigned to the current node. The bottom row shows a set of operations that users can click to assign to the current node.

limitation. When creating a new node or arranging multiple nodes, which are common

activities of visual programming, users need to see the overview (for organizing nodes) and node details (for providing input and output examples) at the same time. Since semantic zoom can only provide a single view at a time, we could not confirm that the small and the large level of semantic zoom provide much benefits.

4.2.5. Version 5: Grid and Pop-up Panel

For the 5th revision, we designed a pop-up panel that shows details of the current node and supports PBE interaction. As illustrated in Figure 22, the top of the panel shows values of the input nodes (left) and the current node (right) side by side so that users can easily compare corresponding input and output examples for PBE. The middle row gives title and description of the operation assigned to the node. The bottom row shows operations generated by the PBE engine based on the input and output examples. Users pick one of the operations to assign to the current node. I liked the panel design for its top, middle, and bottom structure can represent a wide range of situations. However, we soon realized that the bottom part can easily be overcrowded with a large number of generated operations. Moreover, in pilot studies we observed that inexperienced users could not easily grasp in what order they have to use the top, middle, and bottom parts of the panel. In the end, we decided to separate the bottom, so that the panel can focus on details of the current node.

4.2.6. Version 6: Grid, Pop-up Panel, and Side Panel

The 6th design was the last revision, as illustrated in Figure 23. It has a side panel on the left, which contains the information of the current program (called enhancement), and operations (called actions) that the PBE engine generates or filters based on the current node values. The pop-up panel shows information of the current node. Another pop-up panel, called *Inspector*, appears when users select DOM elements of the web page. More details will be discussed in the following sections.

4.3. Example Walkthrough

Here is a typical walkthrough of creating simple enhancements using VESPY. Jane is a knowledge worker who frequently calculate the sum of numbers in a HTML table.

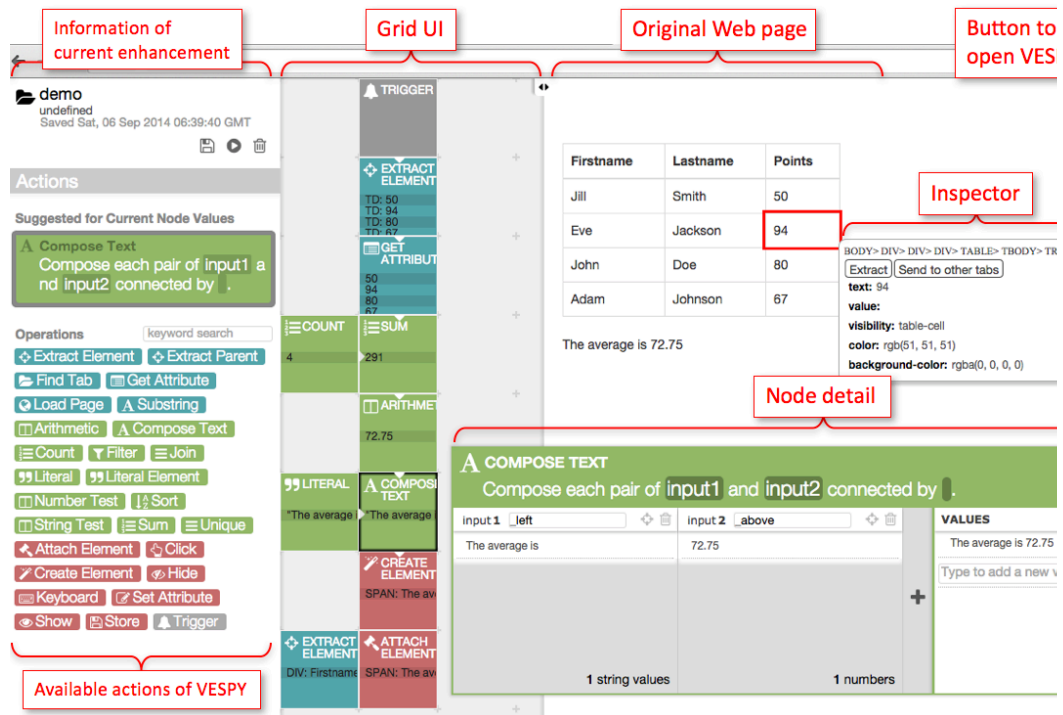


Figure 23. The VESPY user interface consists of the grid, info, actions, and node details. s can open the UI at any web page by pressing the button on the top right corner of web browsers.

Although she has been doing the task by manually importing the entire page to

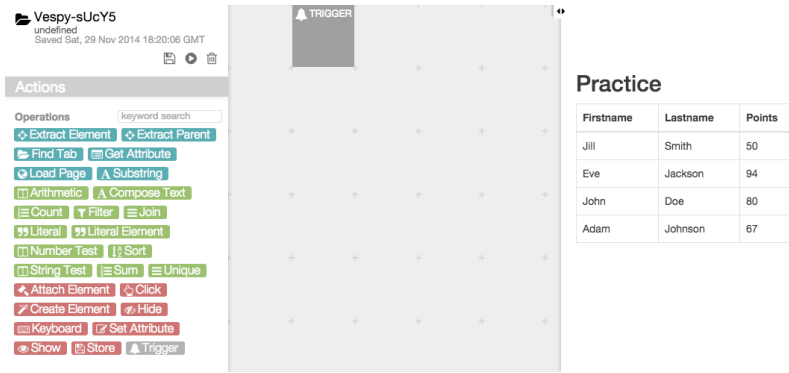


Figure 24. A new enhancement is created. The grid UI contains a Trigger node to begin with. The original web page is shown on the right side.

Microsoft Excel. She wants to add an interactive feature to the web page so that she can calculate the sum simply by clicking a button.

Jane first navigates to the page she visits, and clicks the button on top of her browser (Figure 23). Then the VESPY UI appears on the left side of the browser, pushing the original page to the right. In the middle, the grid UI shows a new empty enhancement with a Trigger node (Figure 24). The Trigger node will execute nodes below and right when the page is completely loaded.

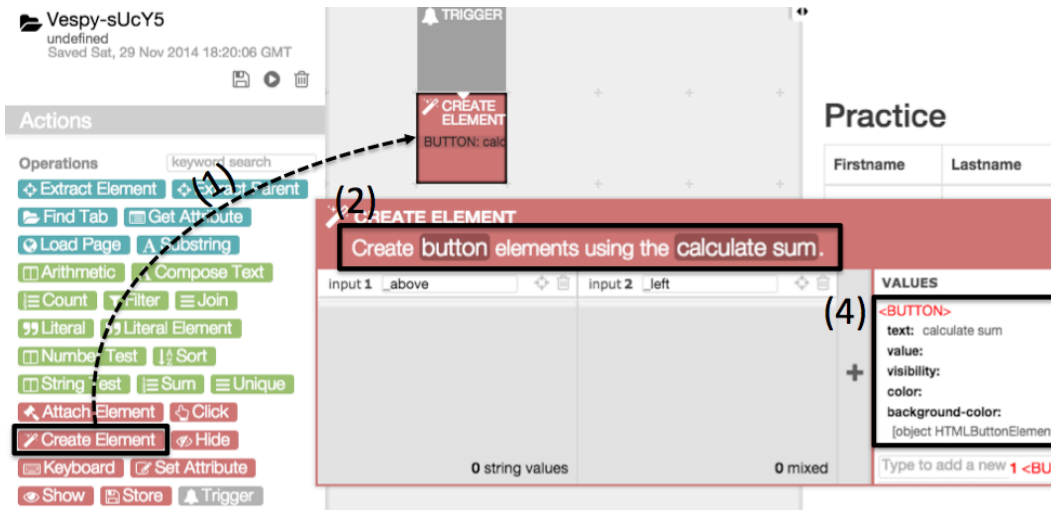


Figure 25. User can (1) drag an operation from Actions panel (left) to the grid (center), (2) directly change options of the operation (e.g. “button”, “calculate sum”) in the floating node detail window, and then (3) run the operation by clicking the play button on the right side of the window. Finally, (4) the values of the node will be updated.

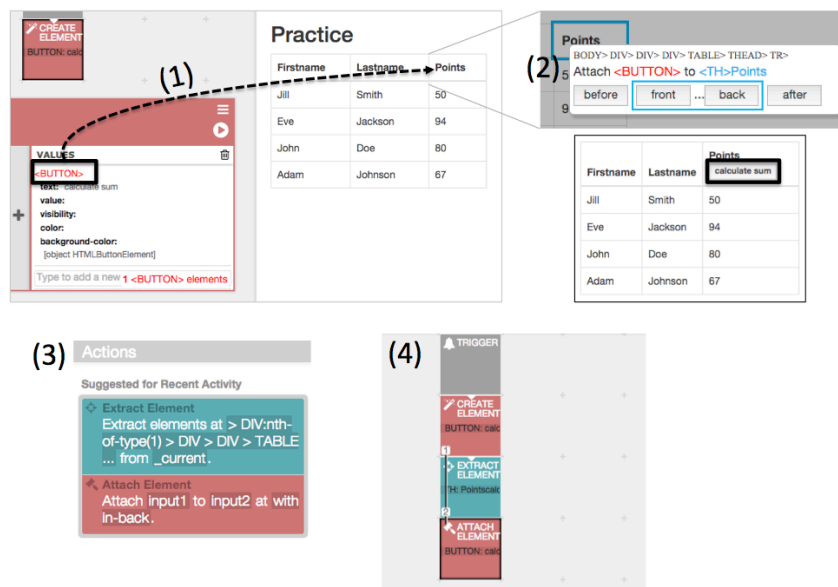


Figure 26. User can attach new elements to any place in the web page by (1) drag-and-drop an element to the target place, (2) choosing the relative position (before, front, back, after) to the target, and (3) clicking a suggested program in the Actions panel. Then (4) two nodes are added to the grid.

Since Jane wants to calculate the sum only when she needs it, the next step is to attach a button for executing the calculation. She drags Create Element operation from Actions panel to the below of the Trigger node. The pop-up panel shows the detail of Create Element operation (Figure 25), she directly specifies parameters from “Create

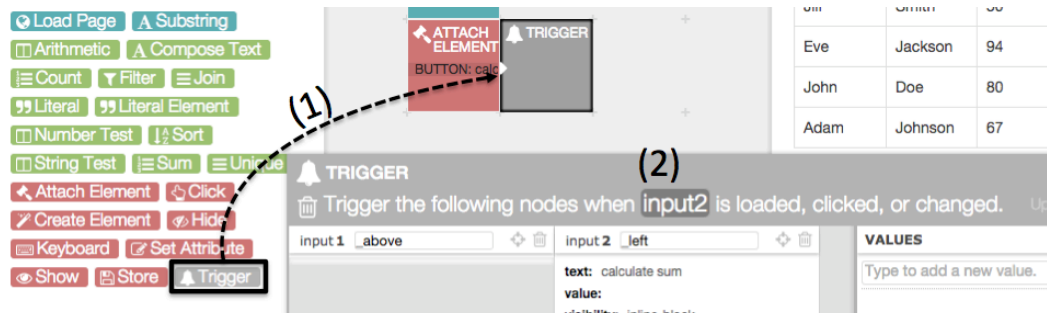


Figure 27. User can set an event handler by (1) dragging Trigger operation next to the node containing elements, and (2) setting the correct input channel.

[span] elements using the [input1]” to “Create [button] elements using the [calculate sum]”.

Now she tests the Create Element operation by clicking the play button on the right side or the panel (Step 3 in Figure 25), the node runs the operation and puts newly created button element to its values.

Then she attaches the “calculate sum” button to the page as Figure 26 illustrates. Users can drag a DOM element, which is a value of the current node, to any target DOM element of the current page. As she drops the button to the table header (Step 1 of Figure 26), another pop-up panel shows up and asks her to clarify its relative placement (before, front, back, after) to the target. If she chose back, and the button would be attached as the last element in the target. After attaching at least two elements, the PBE engine generates corresponding a 2-step operation (Extract Element, and Attach

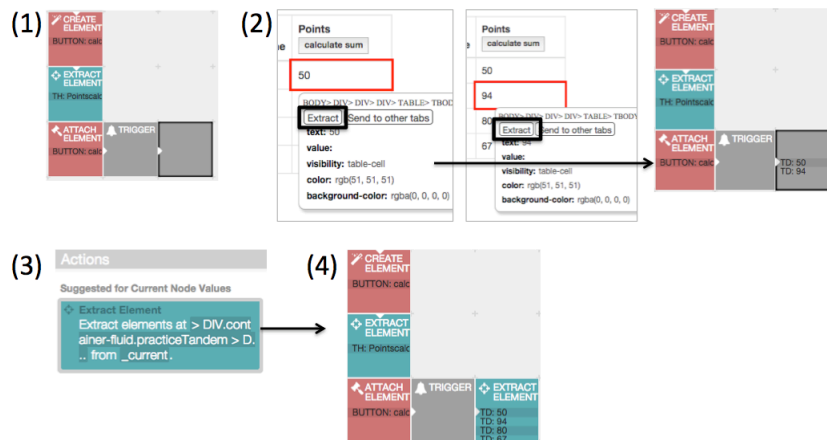


Figure 28. s can specify a node that extracts elements at a specific DOM position by (1) create an empty node, (2) click an element of interest and press extract button (repeat twice for extracting a set of elements), and (3) confirm the suggested Extract Element operation in the action panel. Then (4) the empty node is replaced with the node that can extract all the elements at the same position.

Element), and shows them in the Actions panel. As she clicks the 2-step operation, they are inserted in the grid UI.

Next step is to create an event handler, which specifies what will happen when the button is clicked in runtime. She drags Trigger operation at the right of the Attach Element node, which contains the button element (Step 1 in Figure 27). Now the Trigger node

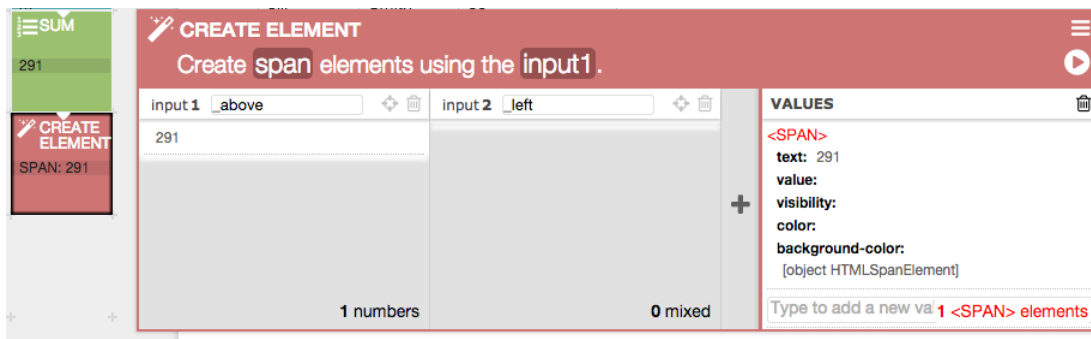


Figure 29. s can create new elements from values with Create Element node.

monitors the button element, and executes the following nodes when the button is clicked.

Thus far the enhancement creates a new button, attaches it to the table, and assigned an event-handler to the button. The next step is to define how to calculate the sum of the numbers. As illustrated in Figure 28, she creates an empty node next to the trigger so that the node will be executed when the button is clicked. She clicks the DOM elements of the numbers, and click Extract button in the inspector pop-up to add them to the current node. (Step 2 in Figure 28). As multiple (at least 2) elements are added to the current node, the PBE engine generates an Extract Element operation (or multiple operations), and show it in the Action panel. She validates the suggested operation, and clicks to assign to the current node (Step 4 in Figure 28).

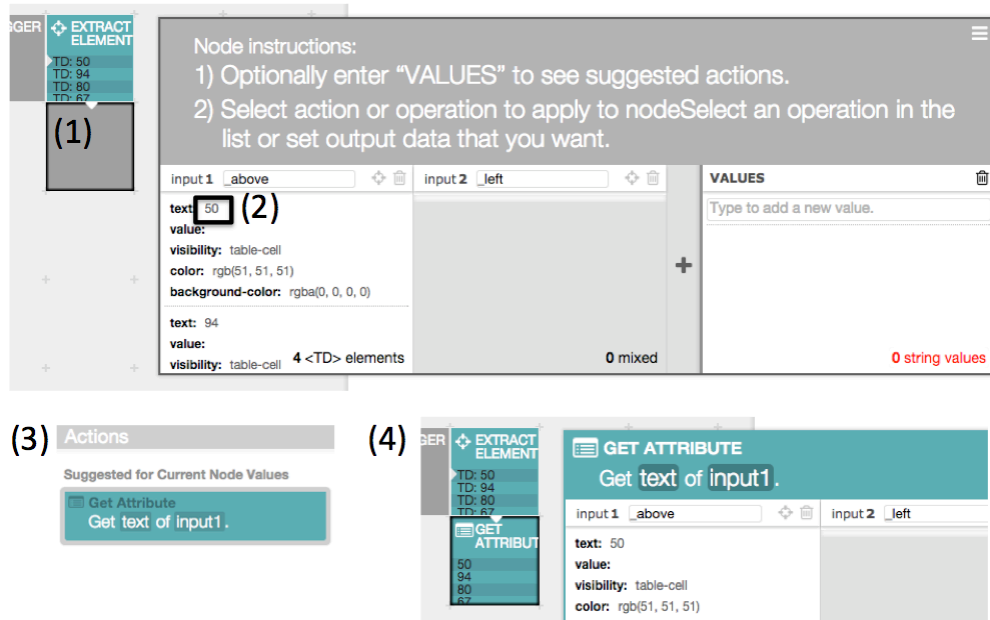


Figure 30. s can extract specific attributes from elements by (1) creating an empty node next to the elements, (2) clicking the attribute value in the detail window, and (3) confirming the suggested action.

The screenshot shows a PBE interface with an 'Actions' panel on the left and a workflow canvas on the right. The 'Actions' panel contains various operations like 'Extract Element', 'Find', 'Load', 'Arithmetic', 'Compose Text', 'Count', 'Filter', 'Join', 'Literal', 'Literal Element', 'Number Test', 'Sort', 'String Test', 'Sum', 'Unique', 'Attach Element', 'Click', 'Create Element', 'Hide', 'Keyboard', 'Set Attribute', 'Show', 'Store', and 'Trigger'. The workflow canvas consists of several nodes: a 'CREATE ELEMENT' node (BUTTON: calc), an 'ATTACH ELEMENT' node (BUTTON: calc), a 'TRIGGER' node, an 'EXTRACT ELEMENT' node (TH: Points), a 'GET ATTRIBUT' node (TD: 50, TD: 94, TD: 93, TD: 67), a 'SUM' node (291), another 'CREATE ELEMENT' node (SPAN: 291), and another 'ATTACH ELEMENT' node (SPAN: 291). A table titled 'Practice' is shown on the right with columns 'Firstname', 'Lastname', and 'Points'. The table contains the following data:

Firstname	Lastname	Points
		calculate sum 291
Jill	Smith	50
Eve	Jackson	94
John	Doe	80
	Johnson	67

Annotations in red text explain the workflow: 'Creates and Attaching the button - calculate sum' points to the first two nodes; 'Executes following nodes when the button is clicked' points to the trigger node; 'Extracting points from the table' points to the extract and get attribute nodes; and 'Creating and attaching the calculated sum.' points to the final two nodes.

Figure 31. A simple enhancement creates a button for calculating total points. Three nodes on the left side create and attach “calculate sum” button to the table. When the button is clicked in runtime, the trigger node executes the following nodes to extract all the points from the table, add them, and attach the result back to the page.

The Extract Element node contains DOM elements of numbers of which she wants to calculate the sum. Thus the next step is to get the attribute from the elements. She creates an empty node next to the extracted elements (Step 1 of Figure 30), and clicks the attribute of interest (“50”; Step 2 of Figure 30). Then the PBE generates “Get [test] of [Input1]” operation, and suggests it in the Actions panel (Step 3 of Figure 30). Lastly, she clicks the operation to assign to the current node.

She needs to specify how to calculate the numbers. There are two methods for specifying the Sum operation: first, she can drag and drop the Sum operation directly from the Action panel to the empty cell below the numbers. Second, she can create an empty node below the numbers, and type the correct value of the Sum operation so that the PBE engine will generate and suggest the Sum operation at the top of the Action panel. After specifying the Sum operation, she drag and drop the Create Element

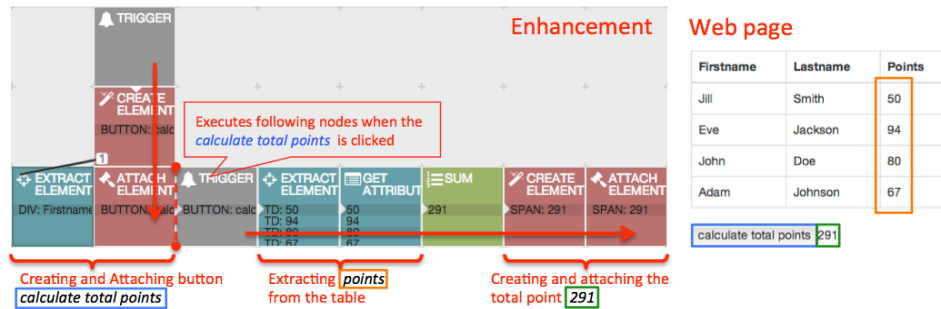


Figure 32. A simple enhancement creates a button for calculating total points. Four nodes on the left side attach “calculate total points” button to the Web page. When the button is clicked, the trigger node runs the following nodes to extract all the points from the table, add them, and attach the result back to the page.

operation, and change the parameter to “Create [span] elements using the [input1].” The step is important, as we can attach only DOM elements to the page.

Finally, she drags and drops the span element to the table, and clicks the Attach Element operation suggested in the Action panel – as she previously attached the button element to the table. The complete enhancement (Figure 31) attaches a “calculate sum” button to the HTML table, and when the button is clicked, the nodes on the right side of the trigger extract the points, sum them up, and then attach it back to the page.

4.4. VESPY System

Along with the iterative design process, we implemented VESPY as a Chrome browser extension. Users could activate VESPY for any HTML based web pages, to create an enhancement that would automatically customize all the pages in the same domain. This section describes design and technical details of how VESPY works.

4.4.1. The Grid UI

VESPY includes several UI components, illustrated in Figure 23. Users open the UI by clicking the button at the top of the browser window. The *Grid* panel shows all the nodes of the current enhancement. The *Info* panel contains the current enhancement’s

title and description. The *Actions* panel shows operations defined in the VESPY’s domain specific language and actions suggested by PBE and PBD. The *node detail* UI shows details of the currently selected node such as *operation*, *values*, and *input nodes*.

VESPY employs the data-flow programming paradigm [32] that represents a program as a directed graph of data flowing between connected operations. Edges between nodes pass not only data but also define which node should run next. As an example, Figure 32 illustrates the structure of a simple enhancement that attaches a “*Calculate total points*” button below a plain HTML table. When the button is clicked, the nodes on the right side of the trigger extract the points, sum them up, and then attach it back to the page.

Most real-world problems are complex enough that state-of-the-art PBE engines cannot solve in single steps. Thus it is crucial for users to deconstruct and reconstruct smaller modules. Our approach is to interleave visual programming and PBE techniques. With the grid UI, users create nodes and arrange them to compose large programs without necessarily following the order of execution. Although some existing PBE tools such as CoScripter [47], Wrangler [25] or Karma [79] allow users to build up multi-step programs, they support sequential lists only that users have to create steps in the exact order of execution. In contrast, VESPY’s grid UI provides more flexibility for users to arrange multiple groups of operations, where each group can be independently created and tested, and connect them later to compose large programs.

4.4.2. Direct Specification

VESPY allows users to directly specify a node by dragging an operation from the Action panel to the grid. Users then manually change the parameters in the node detail UI, and

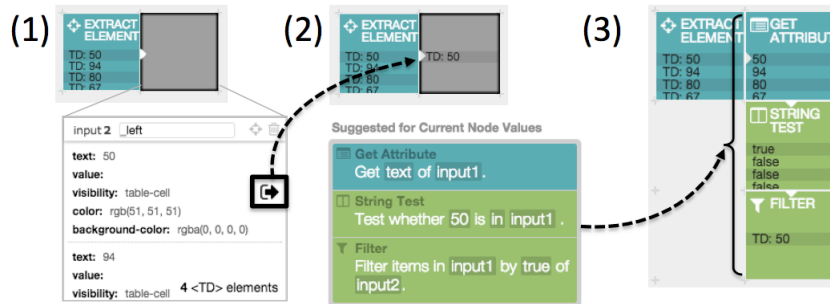


Figure 33. Users can bring elements in the input node by (1) clicking the arrow button in the node detail window. (2) When the current node contains elements of the input node, PBE suggests a three-step task that filters the input elements by their properties. (3) Clicking the task will add three new nodes for the filtering task.

test the completed operation. While direct specification is simple and efficient for simple operations, and applicable for most programming tasks, it has learnability and efficiency issues as previously noted by researchers [76]. First, direct specification requires users to know all the syntax and usage from documentation. Second, even if end-users had sufficient knowledge, directly specifying complex tasks requires a significant amount of time and effort. While direct specification is suitable for simple WebEUP tools, expressive programming tools like VESPY, which has more than 30 operations, require alternative methods.

4.4.3. Programming-by-Example (PBE) Engine

As reviewed in section 2.2.4, PBE is an approach to find programs that are consistent with a few input and output pairs given from the user. VESPY provides PBE techniques to generate single or multiple operations from user intent. VESPY offers provides six ways to express their intent.

First, users can type desired output values in a node that follows the input. This is suitable for data transform operations such as arithmetic, filter, sort, and substring. Figure 34 illustrates the example process of creating sort operation using PBE.

Second, users can extract couple examples of DOM elements from web pages, and ask the PBE to infer a consistent Extract Element operation for elements at the same position. Figure 28 illustrates a typical use case of this process.

Third, users can choose specific attributes (e.g. text) directly from input elements (e.g. paragraph). For example, Figure 33 illustrates the process of getting text attribute from TD element by clicking an example of desired attribute values in the following node's detail.

Fourth, users can directly choose a subset of input elements as intent of filtering task. Then the PBE infers tasks that consist of 2-4 operations (Get Attribute, Number / String Test, and Filter).

Fifth, users can drag and drop values in the current node to the web page. To attach elements to current page, a user opens up the node containing the elements, and drags

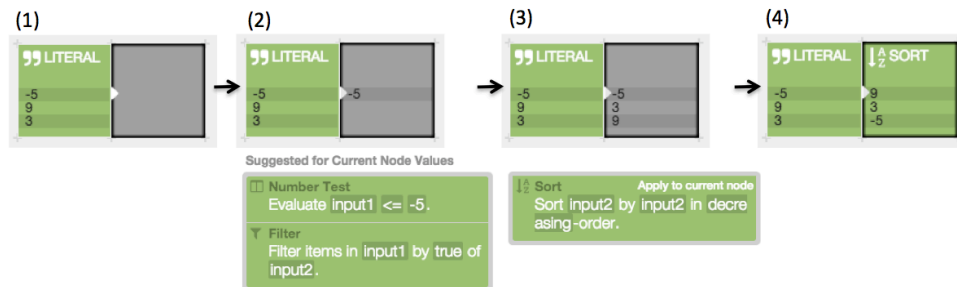


Figure 34. VESPY PBE suggests single / multiple operation tasks based on the values of the input nodes and the current node. To sort a list of numbers, an user (1) creates an empty node that follows the input node. (2) He starts typing desired output “-5”. However, at this point, PBE can only suggest a task with Number Test + Filter operations. (3) As he typed the sufficient output values, PBE suggests a correct Sort operation. (4) He clicks the suggestion to confirm it as the node’s operation.

the button tag to the target as shown in Figure 26. If the user wants to attach the element

to a set of targets, he/she repeats the steps once more. Then VESPY will suggest a 2-step task (Extract Element → Attach Element).

Lastly, users can express their intent with multiple input nodes. For example, as **Error! Reference source not found.** illustrates, if an user wants to filter elements with a complex predicate logic, he can prepare a few steps to get the key values, and then



Figure 35. Filtering a set of table rows by values of a specific column requires the filtered list [c] and the key values for predicate [b]. Users (1) extract key values from the original list, (2) use both the original elements and the key values as input nodes for the node of filtered elements.

4.4.4. Domain Specific Language (DSL)

The expressive power of VESPY enhancements is defined with the domain-specific language (DSL) written in JavaScript. The DSL enables users to build a wide range of enhancements by combining the five areas of common WebEUP functionalities. As summarized in Figure 36 and Figure 37, an enhancement consists of multiple nodes that are connected to other input nodes. Each node contains an operation (P) and a list

of values (V). VESPY currently supports only four value types (*DOM element, String, Number, and Boolean*), and each value list can have single data type, defined by the top element. An operation has a type (e.g. Load page, Extract Elements) and parameters.

The DSL's operations are shown in Table 2, covering five areas the common WebEUP functionalities plus event handling and data storages.

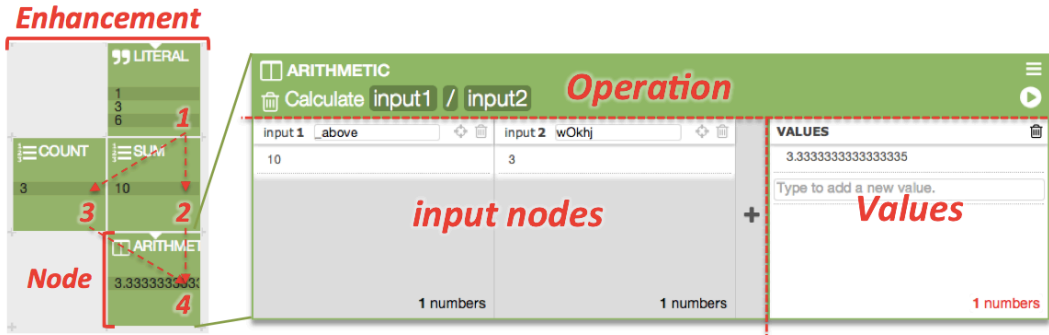


Figure 37. Representation of the VESPY program. An enhancement consists of multiple nodes. The enhancement in this figure calculates the average of numbers ([1,3,6]) by running the four nodes in the numbered order (1→2→3→4). Each node contains an operation, values, and input nodes. When its preceding node triggers a node, it executes its operation, updates its values, and then triggers its following nodes.

$$\begin{aligned}
 \text{Enhancement } E &:= (n_1, \dots, n_m) \\
 \text{Node } n &:= (V, I, P) \\
 \text{Data values } V &:= (v_1, \dots, v_k) \\
 \text{Value } v &:= \text{DOM element} | \text{String} | \text{Number} | \text{Boolean} \\
 \text{Input Nodes } I &:= (n_1, \dots, n_m) \\
 \text{Operation } P &:= (\text{type}, \text{Param})
 \end{aligned}$$

Figure 36. The syntax of VESPY enhancements.

<i>Operation</i>	<i>Input</i>	<i>Output</i>	<i>Param</i>	<i>Description</i>
DATA EXTRACTION				
Load page	I _{URL}	O _{DOM}	-	Load page DOM elements of I _{URL}
Extract Elements	I _{DOM}	O _{DOM}	<i>path</i>	Extracts elements using <i>path</i> from I _{DOM}
Extract Parent	I _{DOM}	O _{DOM}	<i>d</i>	Get enclosing elements of I _{DOM} , <i>d</i> steps above
Find Tab	I _{URL}			Find a currently open tab of I _{URL} , and executes the following nodes in the tab.
Get attribute	I _{DOM}	O _{VAL}	<i>k</i>	Get attribute <i>k</i> of elements of I _{DOM}
DATA TRANSFORMATION				
Literal	I	O		Directly set the current node data to I
Filter	I	O		Get a subset of I whose corresponding boolean value in I _{BOOL} is true
	I _{BOOL}			
Sort	I _{VAL}	O _{VAL}	<i>direction</i>	Sort values in I _{VAL} in ascending / descending <i>direction</i>
Unique	I	O		O ← I without repeating same values
Substring	I _{STR}	O _{STR}	<i>S₁, S₂, B₁, B₂</i>	Get substring of I _{STR} from string <i>S₁</i> to <i>S₂</i> , including <i>S₁</i> if <i>B₁</i> is true.
String Test	I _{STR1}	O _{BOOL}	<i>p</i>	O _{BOOL} will have <i>True</i> if I _{STR1} contains I _{STR2} , <i>False</i> otherwise
	I _{STR2}			
Number Test	I _{NUM1}	O _{BOOL}	<i>op</i>	Evaluate inequality (I _{NUM1} <i>op</i> I _{NUM2}) and update true / false in O _{BOOL} . <i>op</i> can be <, <=, >, >=, ==, !=
	I _{NUM2}			
Arithmetic	I _{NUM1}	O _{NUM}	<i>op</i>	Calculate two operands I _{NUM1} and I _{NUM2} with operator <i>op</i> , which can be +, -, *, %.
	I _{NUM2}			
Compose text	I _{STR1}	O _{STR}	<i>s</i>	Concatenate every pair of values in I _{STR1} and I _{STR2} with separator <i>s</i>
	I _{STR2}			
MODIFYING / CREATING DOM ELEMENTS				
Attach elements	I _{DOM1}	O _{DOM}		Attach DOM elements of I _{DOM1} to I _{DOM2} .
	I _{DOM2}			
Create elements	I _{VAL}	O _{DOM}	<i>t</i>	Create new DOM elements using Input values and tag name <i>t</i> , which can be <i>button</i> , <i>span</i> , or <i>img</i> .
Literal element	-	O _{DOM}	<i>t</i>	Create a single element from <i>t</i> , which is JSON string of an arbitrary element.
Hide / Show	I _{DOM}	O _{DOM}		Hide / Show I _{DOM} elements.
Set attribute	I _{DOM}	O _{DOM}	<i>k</i>	Updates I _{DOM} elements' attribute <i>k</i> with I _{VAL}
	I _{VAL}			
SIMULATING MOUSE AND KEYBOARD INTERACTION				
Click	I _{DOM}	-		Simulate mouse clicks on Input elements.
Type	I _{DOM}	-	<i>str</i>	Simulate keyboard input <i>str</i> on input field of V _{DOM} .
DATA STORAGE				
Store data	I _{VAL}	-	<i>k</i>	Store Input values into data storage with key <i>k</i> . (Not implemented yet)
EVENT HANDLING AND FLOW CONTROL				
Trigger	I _{DOM}	O _{DOM}	<i>e</i>	Trigger the node on the right when event <i>e</i> occurs.

Table 4. VESPY operations and their required parameters. Subscripted types (e.g. VAL of I_{VAL}) mean that the operation requires the type of the value. I_{DOM} must contain only DOM elements; I_{VAL} can be any type except DOM elements.

4.4.5. Single-step inference algorithms

Many operations of VESPY have inference algorithms that find parameters corresponding to given **Input** (input node values) and **Output** (current node values). When a node value has been changed, the PBE system asks each operation to try its inference algorithm. An inference algorithm would fail and return **false**, if there is no parameter that satisfy the given input and output. The following list briefly explains the inference algorithms.

Extract Element can infer a *path* for extracting a set of elements (**Output**) from a single element / multiple elements (**Input**). If the input node contains a single element, the algorithm tries to infer a *1-to-n* query for extracting all the output elements from the element. If multiple elements are in the input node, the algorithm will try to find a *1-to-1* query that extracts each output from matching input elements. VESPY uses a XPath-based algorithm similar with Sifter [31], Karma [79], and Vegemite [52].

Extract Parent can infer how many steps a set of elements (**Out**) is above another set of elements (**In**) as illustrated in Figure 38. It returns fail if every output does not enclose the corresponding input element.

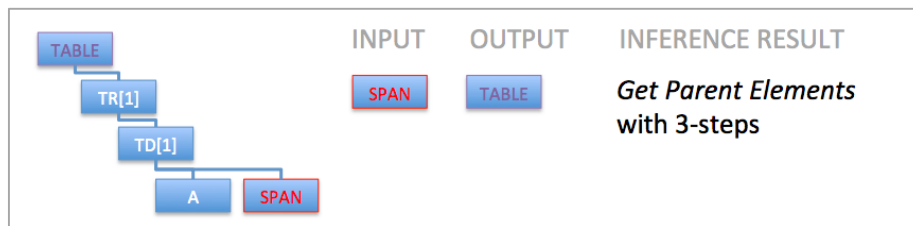


Figure 38. An example of Extract Parent operation inference.

Get Attribute can infer the attribute key for getting the output values from the input elements. For example, if the output elements are URL attributes of the input

elements, then the algorithm will return a **Get Attribute** operation having ‘URL’ as the key parameter.

Literal can infer the parameter for setting the current node values. For example, if the current node values are [1,2,3] then the algorithm simple suggest Literal operation with “[1,2,3]” as the parameter.

Sort can infer the right *direction* to get the output values by sorting the input values.

Unique checks whether the unique set of the input values is equivalent to the output values, and returns a **Unique** operation or *false*.

Substring can infer two tokens (S1 and S2) and boolean values (B1 and B2) for getting the output texts from the input texts. S1 and S2 indicate the starting and ending position of the substring, and B1 and B2 indicate whether S1 and S2 should be included or not. If it cannot find a consistent set of parameters, it returns *false*. Table 5 shows three examples of Substring inference.

String Test can infer a keyword that can determine true or false values of the output from the input strings. The inference algorithm also tries whether the keyword should be in or not in the input string as shown in Table 6.

IN	OUT	RESULT. S1 [B1] – S2 [B2]			
		S1	B1	S2	B2
["CSIC-1032", "MSC-33"]	["1032", "33"]	"_"	false	"_EOF_"	false
["(6/7)(4/5)", "49(28/11)"]	["(6/7)", "(28/11)"]	"{"	true	"")"	true
["323-708-7700", "510-333"]	["323", "510"]	""	false	"_"	false

Table 5. Examples of Substring inference.

IN	OUT	RESULT.	
		keyword	In / not in
["CSIC-1032", "MSC-33"]	[true, false]	"CSIC"	in
["a 1", "b 2", "a 2"]	[false, true, true]	"1"	not in
["tomato soup", "potato soup", "tomato salad"]	[true, false, true]	"tomato"	in

Table 6. Examples of String Test inference

IN	OUT	RESULT.	
		operator	Operand
[-5, 3, 9, 1, 2]	[true, false, false, false, false]	<=	-4
[1,2,3,4,5]	[false, true, false, true, false]	%= (divisible)	2
[3,1,2,0,5]	[false, true, true, true, true]	!=	3

Table 7. Examples of Number Test inference

IN1	IN2	OUT	RESULT.		
			Operand1	operator	Operand2
[-5, 1, 2]	[5]	[10,6,7]	IN1	+	IN2
[1, 2, 3]		[2, 4, 6]	IN1	*	2
[3]	[6,2,-3]	[0,2,0]	IN2	%	IN1

Table 8. Examples of Arithmetic inference

IN1	IN2	OUT	RESULT.		
			Text1	connector	Text2
["CSIC", "MSC"]	[1032, 33]	["CSIC-1032", "MSC-33"]	IN1	"_"	IN2
["a", "b"]		["a is good", "b is good"]	IN1	""	" is good"
["soup", "salad"]	["potato"]	["potato soup", "potato salad"]	IN2	" "	IN1

Table 9. Examples of Compose Text inference

Number Test can infer an operator (e.g. <, >, ==, <=, >=, !=, %=, !% =) and an operand node or number that can determine true or false values of the output numbers from the input numbers as shown in Table 7. In order to get an accurate result, Number Test requires around many examples.

Arithmetic can infer an operator (e.g. +, -, *, /, ^) and two operands (numbers or input nodes) for getting the output numbers. Table 8 shows examples of Arithmetic inference.

Compose Text can infer a connector (text or an input node) and two text (or input nodes) for getting the output text (Table 9).

The rest operations (e.g. Create element, Literal element, Hide, Set Attribute) are not suitable for input and output examples.

<i>Recipe</i>	<i>Condition / Decomposition</i>
Extract Attribute	Condition: Every Output Text exists in Input Element.
Find Path	Condition: Target elements are not within or enclosing the Input elements.
Filter Element	Condition: Filtered Elements is a subset of Original Elements.

Table 10. The core set of task recipes in VESPY. If input and output satisfies the condition, the recipe will create temporary nodes (in orange color) and will try to find sub-solution.

4.4.6. Multi-step PBE with task recipes

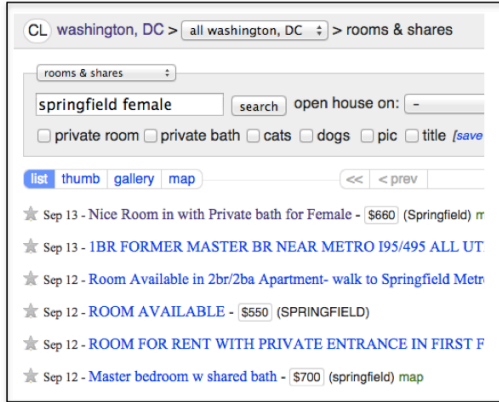
PBE provides a larger benefit when it generates programs for complex tasks. For example, filtering elements by attribute task requires at least four nodes. If PBE can generate the four nodes in a single step, it will save a significant amount of user's time and effort. The problem is that solving large tasks mostly require multiple PBE algorithms.

We thus developed a *planner* that decomposes a large problem into sub-problems, and assigns them to different PBE algorithm. Each plan is called a *task recipe*. Table 10 shows a few of them. The search algorithm was inspired by HTN (Hierarchical Task Network) planning [64]. The algorithm detail is beyond the scope of this paper. In short, when the provided input-output examples match with a recipe's condition, the recipe creates several intermediate nodes (orange color in Table 10) and requests corresponding PBE algorithms to solve them. If they could find matching solutions for every intermediate node, the *planner* combines them and suggests to the user.

4.5. Example Enhancements

To demonstrate the versatility of VESPY, this section presents four enhancements designed to exemplify the kinds of problems we know that users have based on the studies of Chapter 3.

Original page



Enhanced page

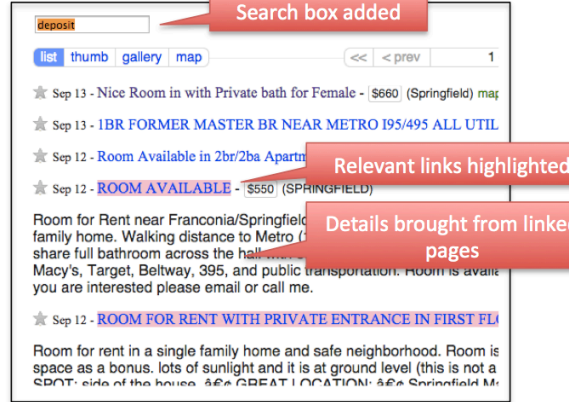


Figure 39. The *deep search* enhancement adds a text input box to the original page. When user types a keyword in the input box, it searches all the linked pages and highlights links whose pages contain the keyword. The main content of the links are attached to the links as well.

4.5.1. Example #1: Deep search

A fictitious user regularly visits Craigslist.com to buy second-hand items. For every item he found interesting, He must visit the detail page, check information, and move back to the listing page. To make it more efficient, He wants to look into linked pages and search keywords without opening them. Let's call it *deep search*. Using VESPY, he built an enhancement consists of 17 operations. *Deep search* attaches a text input box above the links (see Figure 39). When users type a search keyword, it automatically loads every page of the links, and highlights some of the links that contain the keyword. Also, for further preview, it extracts the key content from the pages and attaches below.

Original page

DBLP

2014

- *Example-Based Learning in Computer-Aided STEM Educ* [abstract] [pdf] [bibtex] [ppt slides]
- *Feedback Generation for Performance Problems in Intro* [abstract] [pdf]
- *LaSEWeb: Automating Search Strategies Over Semi-struct* [abstract] [pdf]
- *Applications of Program Synthesis to End-User Programm* [abstract] [pdf]
- *NLyze: Interactive Programming by Natural Language for* [abstract] [pdf] [Video]
- *Programming by Example using Least General Generaliz* [abstract] [pdf] [Video]
- *Synthesis of Geometry Proof Problems, AAAI 2014, Chris* [abstract] [pdf]

Enhanced page

DBLP | WCW | WAMBSE | VMCAI | VLDB | UIST | SYNASC | SIGMOD | SAS | PhI

PPoPP | PPOF | POPL | PLDI | OOPSLA | MobiSys | LPAR | KDD | Journal | IJCI

UICAI | ICSE | ICML | ICCPS | GECCO | FSTTCS | FSE | ESOP | CHI | CAV | CA

CACM | AAAI

Buttons for filtering publications by venue

2014

- *Example-Based Learning in Computer-Aided STEM Education, CACM 2014, Sumit Gulwani* [abstract] [pdf] [bibtex] [ppt slides]

2013

2012

- *Spreadsheet Data Manipulation using Examples, CACM 2012, Resea Highlights Paper, Sumit Gulwani, William Harris, Rishabh Singh* [abstract] [pdf] [Technical Perspective by Martin Rinard]
- *Continuity and Robustness of Programs, CACM 2012, Research Highlights Paper, Swarat Chaudhuri, Sumit Gulwani, Roberto*

Filtered publications

Figure 40. The custom filter enhancement extracts all the venues from the publication list, and attaches a list of unique buttons. When a button is clicked, it shows only the articles published to the selected venue.

Enhanced event listing site

11:00 am - 12:00 pm CS Department
Second Languages as
 Marine Carput - National Reser... Council Canada
 Room 4172 A.V. Williams Building (AVW)
 Add to Google calendar

4:00 pm - 5:00 pm Bioinformatics happenings on campus
Bring Your Own Bioinformatics: The NCGAS I
 Will Gammerding - BIS/BEEES
 MCB 1123
 Add to Google calendar

A button is added to each event

Click

Google Calendar site

Google

Search

← SAVE Discard

Bring Your Own Bioinformatics: The NCGAS

9/16/2014 4:00pm to 5:00pm 9/16/2014

The clicked event information is injected.

Figure 41. The event parser enhancement attaches button to every event in the list. When a button is clicked, it finds an open tab of Google Calendar and fills the input form with the event information.

Original page

Institution	Country	Academic Reputation	Employer Reputation	Faculty Reputation	Citations per Faculty	International Faculty	International Students	Total
MASSACHUSETTS INSTITUTE OF TECHNOLOGY (MIT)	US	100.0	100.0	100.0	99.7	97.6	96.3	
HARVARD UNIVERSITY	US	100.0	100.0	99.3	100	94.1	85.3	
UNIVERSITY OF CAMBRIDGE	GB	100.0	100.0	99.6	95.8	95.5	96.0	
UCL (UNIVERSITY COLLEGE LONDON)	GB	99.9	98.7	98.9	95.6	96.5	100.0	
IMPERIAL COLLEGE LONDON	GB	99.9	100.0	99.8	92.5	99.9	99.9	
UNIVERSITY OF OXFORD	GB	100.0	100.0	100.0	93.1	97.7	96.7	
STANFORD UNIVERSITY	US	100.0	100.0	94.4	100	75.6	76.0	
YALE UNIVERSITY	US	100.0	100.0	100.0	88.8	94.4	72.7	
UNIVERSITY OF CHICAGO	US	99.9	94.3	96.2	97.8	78.8	74.9	

Enhanced page

custom weights

Institution	Country	Academic Reputation	Employer Reputation	Faculty Reputation	Citations per Faculty	International Faculty	International Students	Total	Update
MASSACHUSETTS INSTITUTE OF TECHNOLOGY (MIT)		100.0	100.0	99.7	97.6	96.3	1860.30		
HARVARD UNIVERSITY	US	100.0	100.0	99.3	100	94.1	85.3	1746.4	
UNIVERSITY OF CAMBRIDGE	GB	100.0	100.0	99.6	95.8	95.5	96.0	1850.9	
UCL (UNIVERSITY COLLEGE LONDON)	GB	99.9	98.7	98.9	95.6	96.5	100.0	1889.2	
IMPERIAL COLLEGE LONDON	GB	99.9	100.0	99.8	92.5	99.9	99.9	1890.69	
UNIVERSITY OF OXFORD	GB	100.0	100.0	100.0	93.1	97.7	96.7	1857.80	
STANFORD UNIVERSITY	US	100.0	100.0	94.4	100	75.6	76.0	1630	

Sort button

Weighted total scores with color coding

Figure 42. The multi-attribute ranking enhancement adds text boxes to each column header that users can type in their own weight factors. When a factor is changed, it updates weighted total scores and color codes on the right end of the table. It also attaches the Sort button that reorders the table rows by weighted total scores.

4.5.2. Example #2: Custom Filter

While reading a publication site, a user wants to filter the publications by their venues. However, personal Websites rarely provide filtering functionality. She saw a *custom filter* enhancement in the VESPY repository. While the enhancement had been built for other sites, she could adapt it by modifying two **Extract Element** nodes that extract the items to be filtered, and a target position for buttons. As shown in Figure 40, her *custom filter* extracts all the venues from the page, creates buttons that are alphabetically sorted without duplication. Clicking a button hides articles published to other domains.

4.5.3. Example #3: Event Parser for Google Calendar

A user regularly visits the event-listing site. When he finds an interesting event he has to manually type essential information (when, where, description) to his calendar application. *Event parser* enhancement can help him by adding a button next to each event (see Figure 41). When he clicks the button, it extracts the essential information and looks for a tab of Google Calendar app. If the calendar app is found, it injects the information to corresponding input boxes so that he can check and confirm to create the event.

4.5.4. Example #4: Multi-Attribute Ranking

A user is a prospective student deciding on a university. On the Web, he found a data table containing multiple attributes of universities. He wants to compare them with his own ranking formula. The *multi-attribute ranking* enhancement (shown in Figure 42) attaches input boxes to columns of a plain HTML table so that he can change weight factors, calculate the weighted total scores, and sort the rows. When he tweaks his

weight factors, the weighted total score column updates the weighted scores and colors. After setting the best weight factors for him, he sorts the universities by the updated score. This scenario includes a wide range of tasks: (1) creating and attaching new DOM elements (input boxes and buttons) to the page; (2) extracting information from the page; (3) performing complex arithmetic; (4) modifying attributes (background-color) of elements; (5) sorting elements by custom scores; (6) adding event handlers to the input boxes and the buttons; and finally (7) orchestrating the above tasks with an execution flow. To our knowledge, no existing WebEUP tool can support all of these tasks.

4.6. Preliminary User Study

To answer the research question, “(R3) Is PBE better than direct specification?”, we conducted a preliminary user study using two interaction modes that provide limited functionalities of VESPY. The Direct Specification (DS) mode only allows users to directly drag and drop operations from the Action panel. In the Programming By Example (PBE) mode, the participants were not allowed to drag operations in the actions panel to the grid, but can use the PBE features (e.g. Typing in node values, suggestion of actions).

We recruited 16 amateur programmers through a university mailing list. Three subjects were female and thirteen were male. Their average age was 29.25 years (SD=8.1). We defined the eligibility criteria for amateur programmers that they must be familiar with at least one programming language, and understand basic programming concepts such as loop, conditionals, and data objects. We excluded

applicant who had coded a program longer than 500 lines or for commercial purpose. We offered qualified participants ten dollars per hour.

4.6.1. Method

We conducted a within-subject experiment that compared two modes (DS and PBE) of VESPY. The study began by learning the basics of common part of the UI using Web-based tutorial. Throughout the study, one of the authors was sitting next to the participants answering questions.

First, the participants learned the basic concepts using a web-based tutorial that includes short demonstration clips and exercises. The tutorial took 10-20 minutes. After completing the tutorial, they tried to accomplish four tasks, as shown in Table 11. For each task, they first read the instructions about a randomly-selected version (e.g. DS), tried a practice task, and then completed the actual task. The same process repeated for the other version (e.g. PBE). To minimize learning effects, half of the participants used the DS mode first, while another half used the PBE mode first. Also the practice and the actual tasks used variations of the same problems with different numbers and parameters. At the end of each task, the participants answered to a survey question about the problems' difficulty. After finishing all the tasks, the participants took a general survey and participated in a semi-structured wrap-up interview.

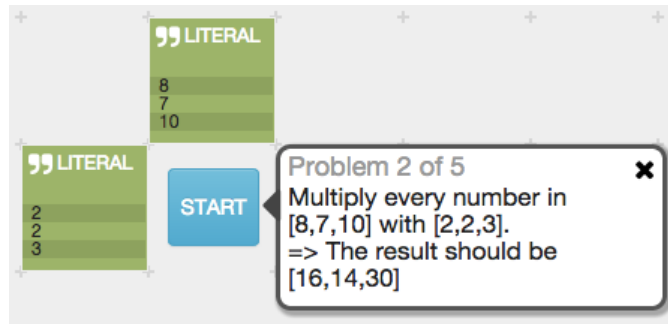


Figure 43. An example of the second problem of the Calculating numbers task. Given the two input nodes, the participants need to create an Arithmetic node that multiplies the two node values.

We measured the completion time for the tasks. However, as VESPY requires training, and is not intended to be a walk-up-and-use system, the level of understandings about the system had large variance across users. During the pilot study, we observed that the participant’s understanding had a dominating impact on their performance. For example, if a participant got lost for several minutes, all the other aspects would make little difference to the total completion time. To avoid the situation, when the participants got stuck for longer than 20 seconds we reminded them high-level hints (e.g. “If you want to extract DOM elements, click them”, “You need to confirm one of these suggestions.”), which were also instructed during the tutorial.

4.6.2. Tasks

We designed four tasks that are commonly used in most enhancement scenarios and solvable in both modes. Each task has 1-5 problems depending on their difficulty. As illustrated in Figure 43, the participants were requested to add new nodes to get the desired result from given input nodes. They pressed the START and DONE button at the

Inputs	Problem Description => Solution
<i>Task 1. Calculating numbers</i>	
[1,2,3], [2,0,2]	Add every number in [1,2,3] with [2,0,2] => The result should be [3,2,5]
[8,7,10], [2,2,3]	Multiple every number in [8,7,10] with [2,2,3] => result: [16,14,30]
[3,6,9]	Divide numbers [3,6,9] with 3 => result: [1,2,3]
[1,9,-5]	Arrange [1,9,-5] in increasing order => result: [-5,1,9]
[4,1,1]	How many numbers are in [4,1,1]? => result: [3]
<i>Task 2. Extracting information</i>	
4 elements	Get the text attributes of the elements
4 elements	Get the URL attributes of the links
4 elements	Get the text of sub-elements within the input.
2 elements	Find a path from a set of elements to another.
<i>Task 3. Filtering</i>	
[apple juice, banana, apple, peach]	Find text values that contains "apple" => The result should be ["apple juice", "apple"]
[1,2,3,4]	Find even numbers => [2,4]
6 elements	Find elements that contains a specific keyword
<i>Task 4. Attaching Elements</i>	
1 input element	Attach an element to a set of items in the page

Table 11. The four tasks for the controlled experiment consist of thirteen problems.

beginning and end of each problem. The time gap was the metric of the system's performance.

4.6.3. Tasks

We designed four tasks that are commonly used in most enhancement scenarios and solvable in both modes. Each task has 1-5 problems depending on their difficulty. As illustrated in Figure 43, the participants were requested to add new nodes to get the desired result from given input nodes. They pressed the START and DONE button at the beginning and end of each problem. The time gap was the metric of the system's performance.

Task Code		Task 1					Task 2				Task 3			T4	
Problem Code		P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	
Required nodes		1	1	1	1	1	1	1	2	3	2	2	4	2	
Completion Time	Direct Spec.	Mean	13.1	14.1	19.4	22.4	13.4	17.8	29.3	63.9	108.8	53.6	53.6	111.5	63.7
		Std	4.68	6.17	6.30	10.61	8.51	13.76	35.56	53.90	69.23	29.69	17.81	75.65	27.85
	PBE	Mean	17	21	18	14	14	15	14	28	57	19	18	55	25
		Std	6.26	14.68	7.43	3.81	7.21	5.81	12.91	17.17	38.50	7.12	11.15	38.81	12.44
	Wilcoxon signed rank	Z-score	-2.8114	-1.5254	-0.8519	2.1344	-0.6592	-0.2841	-0.1675	-3.2577	-2.7406	-3.4645	-3.4645	-2.5854	-3.5162
		p-value	0.005	0.126	0.395	0.033	0.509	0.779	0.093	0.001	0.006	0.0005	0.0005	0.01	0.0004

Table 12. Wilcoxon signed rank test result of the completion times for each problem. For simple problems that require single steps (P1-P7), the Direct Specification condition equivalent or better performance. However, for complex problems requiring multiple-steps (P8-13), the PBE condition was significantly more efficient ($p < 0.03$)

4.6.4. Results

We tested whether completion times of each group follow normal distribution. It turned out that 9 out of 26 (P1-P13 for two conditions) data groups are non-normally distributed ($p > 0.03$). Therefore, we compared two conditions for the entire study using the Wilcoxon Signed-ranks Test. The result indicates that the participants could finish problems requiring multiple steps (i.e. P8-P13) significantly faster under the PBE condition than the Direct Specification condition (see Table 12 for Z-scores and p-values). These results suggest that none of PBE and direct specification outperforms the other in all time, and thus, EUP systems should support both approaches for different circumstances.

4.7. Discussion

4.7.1. PBE vs. Direct Specification

Before running the study, we expected to see either PBE or direct specification outperforms the other. To the contrary, the usefulness of PBE appears to be affected by three factors: (1) user's knowledge of the domain-specific language, the PBE engine, and the task; (2) the amount of work for creating sufficient examples vs. directly specifying parameters; (3) credibility of programs.

First, to use PBE effectively, users must know what program the system can generate. Otherwise, users must figure out the system's capability through trials-and-errors, which can be tedious and frustrating. For instance, while learning the Extract Element operation, s4 wondered, "*What if I want to extract only these two elements not the entire set?*" System knowledge is important to feel confidence of the programs they make, and to apply the same approach for similar problems, users need to understand how the PBE engine extracts DOM elements from a few examples. Existing PBE systems teach its capability and limitations with samples and feedback, we believe there is plenty of room for improvement.

Second, if a user can create a program using both PBE and direct specification, he/she will choose a more efficient approach. For simple problems (P1-P7), participants preferred to use direct specification, which requires less time and effort, as s7 said, "*why should I type correct outcomes when I can program it easily?*" For complex problems (P8-P13), participants could easily perceive the benefits of PBE, which is not just easier but also more efficient than direct specification. Future PBE systems should consider how to make example creation more efficient.

Lastly, lack of credibility is another important issue of PBE. Even after learning the usage of PBE, some participants were still reluctant to completely rely on it. s8 told us, "*For larger data set, I would prefer direct specification, because PBE may generate incorrect solutions.*" How to quickly build up credibility between user and system is an interesting research question of the longitudinal study in our future work.

Designing PBE involves issues very different from conventional direct manipulation UI, which are still open research questions in HCI. In the next chapter we

will conduct a user study that investigates to what extent inexperienced users can use PBE, and what mistakes they make.

4.7.2. Limitations

The study design has a few simplifying assumptions that limit the scope of its findings. First, participants performed relatively simple, abstract programming tasks (e.g. arithmetic, filtering), which can be solved in 1-3 steps. As abstract tasks are widely used in many EUP domains (e.g. data wrangling, text transformation), our findings are generalizable for PBE tasks beyond web customizations. However, if participants were asked to build practical solutions for real-world problems such as the four enhancements in 4.5, findings of the study would be different.

Second, participants did the tasks with the both approaches (PBE and DS) while learning the usage of VESPY. We expect that significant ordering effects exist. For instance, participants could be fixated to the approach they learned first. Or, they could perform better with the approach they learned later. The ordering effect was counter balanced by letting half of the participants learn PBE first, while the other half learn DS first.

Third, VESPY is a practical EUP tool, which eventually requires users to learn computational thinking skills and develop programs for their own problems. The preliminary study in this Chapter reports only the first few hours of user experience. To assess the efficacy of the tool, a multi-dimensional in-depth long-term case study (MILCs [77]) would be appropriate.

4.8. CONCLUSION

This chapter presented VESPY, an end-user programming environment for creating web enhancements. VESPY enables amateur programmers to deconstruct complex tasks into smaller sub-tasks, and to find programs for sub-tasks with examples. Four scenarios of sample enhancements demonstrate unique capability and versatility of VESPY's approach. In the preliminary user study, we observed that PBE significantly increased user's performance for multi-step tasks. However, direct specification is as good as PBE for single-step tasks. We believe VESPY can help Web end-users improve their productivity by creating, sharing, and customizing interactive Web enhancements.

Chapter 5: Understanding Human Mistakes when Programming by Example

5.1. Abstract

In the previous chapters, we examined how inexperienced users would describe computational tasks (Chapter 3), and introduced VESPY, a WebEUP system that employs visual programming and PBE techniques (Chapter 4). Findings from the preliminary user study of VESPY indicate that PBE systems can be much harder for inexperienced users than PBE researcher's expectation. Unfortunately, there is little research on people's ability to accomplish complex tasks by providing examples. This chapter presents an online user study, reporting how well people decompose complex tasks, and disambiguate sub-tasks. The findings suggest that disambiguation and decomposition are difficult for even highly-motivated workers from Amazon Mechanical Turk. We identify seven types of mistakes made, and suggest new opportunities for actionable feedback based on unsuccessful examples, with design implications for future PBE systems.

5.2. Introduction

As described in Chapter 2, the goal of PBE is to enable ordinary people to automate complex and repetitive tasks, and it has even made its way into commercial products such as Microsoft Excel's FlashFill [23]. However, guiding inexperienced users on how to provide high-quality examples is still an open-ended research question. To create high-quality examples, users need to consider two requirements: (1) **disambiguation**, and (2) **decomposition**. First, users must be able to provide diverse cases to

disambiguate the operation they want to create from other operations the PBE engine could infer. Second, to create operations for complex tasks, users need to decompose those tasks into small sub-tasks that the PBE engine can (more easily) infer. Both disambiguation and problem decomposition are challenging computational thinking skills and are often part of required training for computer science and engineering students.

To answer the research questions, “(R4) Can inexperienced users perform problem decomposition and disambiguation?” and “(R4a) What mistakes do users make when using PBE?”, we conducted an online user study with participants recruited from Amazon Mechanical Turk (AMT) who were asked to complete 6 tutorials and 5 main tasks using our PBE system. Our research focuses on examining the behavior of ordinary people providing input and output examples, managing steps and cases for decomposition and disambiguation, and making and fixing mistakes. To provide recommendations for PBE tool designers, we also designed two feedback mechanisms, and compared their impact on the main task success rate. A total of 161 users participated in the study, and 30 of them successfully completed all five main tasks.

Our findings suggest that disambiguation and decomposition are difficult for even highly-motivated AMT workers, and for those that had practiced all required subtasks during the tutorials. We report seven types of mistakes identified from unsuccessful trials. We also determined that those unsuccessful trials contain meaningful information about users’ intent and misunderstandings about PBE. Under the actionable feedback condition, participants received context-aware suggestions based on the information from unsuccessful trials, and outperformed other participants.

5.3. METHODS

We conducted an online user study that began with a brief introduction to PBE. Then six tutorials on the user interface and basic PBE tasks (Table 13) were given. After finishing the tutorials, participants were asked to complete five main tasks, that are advanced variations of the tutorials. Finally, the tasks were followed by a demographic survey. The study took around 26 minutes ($M = 25.97$, $STD = 11.54$), and participants who finished the entire study were paid \$3.00. The study was posted on Amazon Mechanical Turk for two days, during which 161 workers started the first tutorial, 137 workers finished the tutorials and proceeded to the tasks, but only 30 finished the entire study. Summary demographics of the 30 participants who finished the entire study indicate the majority age range was 25-34 (60%, $M = 36.43$, $STD = 7.56$), male (60%), with bachelor (50%) or high school degrees (37%). The majority (84%) of participants reported that they had no programming knowledge (57%) or only basic concepts (27%). However, many of them had various IT experience, such as using spreadsheets (70%), creating web pages using HTML (30%) or content-management systems (20%), database (23%), and scripting languages such as Python or Ruby (20%).

	Description	Default examples	Solution examples
Tutorials	T1,2 Input + 1	IN 1 OUT 2	IN 1 5 OUT 2 6
	T3 (Input + 1) * 2	IN 1 OUT 4	IN 1 2 STEP 2 3 OUT 4 6
	T4 Get the sum of all numbers	IN 1,1 OUT 2	IN 1,1 3,2 OUT 2 5
	T5 Get length of a text value (including spaces).	IN yes OUT 3	IN yes no OUT 3 2
	T6 Find numbers that are greater than 9	IN 11,8,9,10 OUT 11,10	IN 11,8,9,10 STEP T,F,F,T OUT 11,10
	Main tasks	T7 (Input + 1) * (Input - 1)	IN 1 OUT 0
T8 Sort numbers in ascending order		IN 1,-1 OUT -1,1	IN 1,-1 5,2,3 OUT -1,1 2,3,5
T9 Find words that are longer than two letters		IN be, are, I, some OUT are, some	IN be, are, I, some STEP 2,3,1,4 STEP F,T,F,T OUT are, some
T10 Find numbers that are not divisible by 4 without remainder		IN 1,4,5 OUT 1,5	IN 1,4,5 2,4 STEP T,F,T F,T OUT 1,5 4
T11 Extract prices of cars that are manufactured in 2014 or later.		IN Civic(2014)-\$12000, Elantra(2012)-\$9500, Corolla(2015)-\$14000, Corolla(2013)-\$10000 OUT 12000,14000	IN Civic(2014)-\$12000, Elantra(2012)-\$9500, Corolla(2015)-\$14000, Corolla(2013)-\$10000 STEP 2014, 2012, 2015, 2013 STEP 12000, 9500, 14000, 10000 STEP T, F, T, F OUT 12000,14000

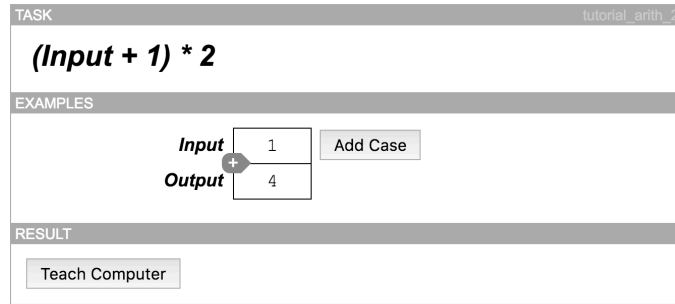
Table 13. With the given description and default examples for each task, participants were asked to add more examples, such as the solution examples shown.

5.4. Experimental System

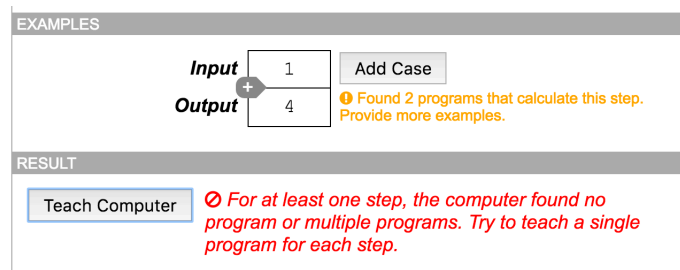
We developed an experimental PBE system that allows non-technical participants to quickly learn and perform decomposition and disambiguation as illustrated in Figure 1.

The system can generate simple programs for standard PBE tasks (e.g. arithmetic, text

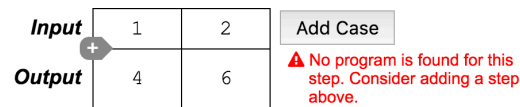
1. Initial state of the task UI that contains default Input and Output values (1 and 4), buttons for adding case (“Add Case”), adding step (“+”), and inferring operations from current examples (“Teach Computer”).



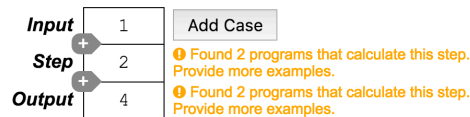
2. As the user clicks the “Teach Computer” button, the UI shows feedback messages for every step and the entire program.



3a. As the user clicks “Add Case”, an empty column is added to the right of the table in which he/she types an example (2 and 6).



3b. Alternatively, the user could click [+] between two rows, and an extra step would be inserted between the rows.



4. By adding a case and a step, the user makes every step find a single operation, and teaches the correct operation.

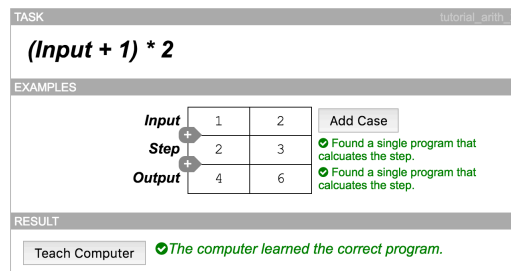


Figure 44. The study UI and basic walkthrough

processing, filtering). In the system’s UI, table rows represent sequential steps from input to output, and table columns represent independent cases. Participants can type

examples values in table cells, insert steps by pressing “+” buttons between rows, and add cases by pressing the “Add Case” button. Pressing “Teach Computer” runs the PBE inference engine, to generate operations that calculate each step. When the engine fails to determine operations from the provided examples, feedback messages are shown to the right rows. If participants spent at least three minutes, and tried (unsuccessfully) to “Teach Computer” at least eight times for a given task, a button was shown to allow them to give up and move on to the next task. Through internal pilot tests, we decided on a reasonable, high number of minutes and trials to give people the chance to try a number of answers in order to study example-providing.

We designed two types of feedback (*simple* and *actionable*) to see whether actionable feedback effects user’s behavior. The **simple feedback** provides only the number of programs that the system generated. We designed the simple feedback as the baseline condition, since most existing PBE systems [23,35,54,88] provide a similar level of feedback for generated programs. In contrast, the **actionable feedback** detects user’s intentions from the examples, and explains details why it failed to generate any program and how to resolve the issue. To our knowledge, no prior PBE systems provide actionable feedback.

When the PBE system finds a **single operation** for the step, both types of feedback show the same message, "*Found a single program that calculates the step.*"

When the system finds **multiple operations**, the *simple* feedback is "*Found N programs that calculate this step*", where *N* is the number of generated operations. The *actionable* feedback is same, but adds "*Provide more examples.*" to the end.

When the system finds **no operation** for the step, the *simple* feedback is "*Found no program that calculates this step.*" In contrast, the *actionable* feedback includes the following messages:

- If there is an empty cell in the current row, the *actionable* feedback is, "*There is an empty case. Did you miss filling it?*"
- If the current row contains values of multiple types (e.g. number and string), the actionable feedback is, "There are number and string examples in this case. This might have caused the computer to fail in finding a program."
- If there is any row above the current row that contains all the values of the current row, the actionable feedback is, "*If you are trying to filter values from steps above, you need an additional step containing T or F.*"
- If the current row is a substring of a filtered subset of any row above, the actionable feedback is, "Are you trying to filter and extract part of string at the same time? If that's the case, you have to do them in two steps."

Task	Success rate			Average # Trials (per participant)			
	Base.	Exp.	χ^2 p-value	Base.	Exp.	Mann Whitney U-test	
Tutorials	T1	1.00	1.00	>.5	1.67	1.07	p > .5
	T2	0.93	1.00	>.5	3.00	1.20	Z = 0.91, p < .30
	T3	0.80	0.87	>.5	6.80	3.47	Z = 2.13, p < .30
	T4	1.00	1.00	>.5	3.40	1.87	Z = 0.76, p < .30
	T5	1.00	1.00	>.5	1.20	1.07	p > .5
	T6	0.67	0.67	>.5	7.40	6.47	p > .5
Main tasks	T7	0.53	0.87	<.05	10.33	3.13	Z = 2.32, p < .01
	T8	0.67	1.00	<.03	8.73	2.73	Z = 5.56, p < .3
	T9	0.27	0.93	<.001	18.27	5.27	Z = 2.90, p < .001
	T10	0.53	0.93	<.03	13.00	4.13	Z = 1.60, p < .05
	T11	0.27	0.67	<.03	28.73	6.87	Z = 3.17, p < .001

Table 14. Success rates (proportion of participants who passed the task) and average numbers of trials for the baseline (Base.) and the experimental (Exp.) conditions. Highlighted cells are significant (p<.05).

5.5. SUCCESS RATE

As mentioned, 30 of the 161 participants finished the entire study. They successfully finished most tutorials (*average success rate* = 91.1%, *# trials* = 3.22) as shown in Table 2. The main tasks were successfully completed less often than the tutorials (*success rate* = 66.7%, *# trials* = 10.12) To understand the effect of feedback on successful task completion, we conducted a non-parametric repeated measure ANOVA test [84]. The result yielded an F ratio of $F(1, 150) = 26.01, p < .001$, indicating that the success rate was significantly greater with the actionable feedback than with the baseline feedback. We also conducted factorial ANOVA to check the effect of demographic factors on success rate, but found no significant impact ($p > .03$).

5.6. Types of Mistakes

We counted mistakes as participant errors in user-provided examples that prevent the PBE engine from generating a single program for each step. The first author reviewed 150 task results (5 main tasks done by 30 participants), and identified 246 mistakes. 25.6% of the mistakes were *critical*, meaning that they remained until participants gave up the task. We grouped mistakes into the categories below.

5.6.1. Missing steps (found 92 times; 30 were critical)

The PBE engine failed to generate programs when participants did not provide crucial steps as illustrated below: (a) missing steps of key values above predicates (35 times; 15 critical), (b) missing steps of predicates values above a list filtering step (31 times; 7 critical), (c) subtasks of a combination of filtering and text extraction (22 times; 15 critical), and multi-step arithmetic (T3 and T7; 4 times) as illustrated in Table 15.

5.6.2. Ambiguous cases (29 times; 11 critical)

Participants often could not provide sufficient examples for the engine to find the right program. For example, participants stuck with single-case examples (18 times; 8 critical). (a) To generate a “not divisible by 4” condition for T10, the input requires “2”, but eight participants had to try multiple times, and three of them gave up. (b) Similarly, T8 (sorting numbers) requires an additional case containing at least three numbers, whose

(a)	IN	be, are, I, some	For T9, A predicate step (ST1) needs a step of key values (“2,3,1,4”) above.
	ST1	F,T,F,T	
	OUT	are, some	
(b)	IN	1,4,5	For T10, filtered result (OUT) requires a step containing predicate values (“T” for including, “F” for excluding values).
	OUT	1,5	
(c)	IN	Civic(2014)-\$12000, Elantra(2012)-\$9500, Corolla(2015)-\$14000, Corolla(2013)-\$10000	For T11, the output (“12000, 14000”) is a substring of the filtered list. It requires either a substring of the original list or the filtered list above.
	STEP	2014, 2012, 2015, 2013	
	STEP	T, F, T, F	
	OUT	12000,14000	

Table 15. Examples of missing steps

output is not the input in reverse-order. See examples in Table 16.

(a)	IN	1,4,5	2,4	For T10, to disambiguate “divisible by 4” from “divisible by 2”, IN requires a value “2”.
	STEP	T,F,T	F,T	
	OUT	1,5	4	
(b)	IN	1,-1	5,2,1	For T8, examples for sorting must contain three numbers that are not in reverse order.
	OUT	-1,1	1,2,5	

Table 16. Examples of ambiguous cases

5.6.3. Inconsistent or unsupported values (28 times; 8 critical)

Participants provided a variety of values that the PBE engine could not find a matching program, such as inconsistent values for arithmetic tasks (9 times; 2 critical), incorrect predicates for filtering (5 times; 1 critical), and incorrectly sorted list (2 times). Participants also provided steps with single Boolean values, when the correct program requires multiple values (7 times; 3 critical). Participants often made formatting mistakes such as (a) Boolean values next to numbers (e.g. "T11, T10, F8, F9": 2 times), Boolean values without a separator (e.g. "FTFT"; 3 times) and using "Yes" and "No" instead of "T" and "F" (1 time).

5.6.4. Unnecessary steps (15 times; 5 critical)

Participants often added unnecessary steps. For example, (a) they often provided steps

(a)	IN	11,8,9,10	“T11” probably means that the value “11” is marked with “T”
	STEP	T11, T10,F8, F9	
	OUT	11,10	

Table 17. Examples of inconsistent or unsupported values

of unnecessary Boolean values for filtering tasks (7 times; 2 critical), numbers for arithmetic (4 times; 2 critical), or completely empty steps (2 times; 1 critical). For T10,

(a)	IN	1	“Input+1” and “*2” are formulas for arithmetic tasks.
	STEP	Input+1	
	STEP	*2	
	OUT	4	
(b)	IN	be, are, I, some	“are>2” and “some>2” are conditional formulas for predicates.
	STEP	T	
	STEP	are>2	
	STEP	some>2	
	OUT	are, some	

Table 19. Examples of describing with formula

(b) two participants provided a step that contains "4", which is the operand of the *number-predicate* program they need (2 times). For examples, see Table 18

5.6.5. Describing with formula (11 times; 7 critical)

Five participants described steps with formulas instead of example values. For instance, (a) they provided "Input+1", "*2", "(2)*(0)", and "+1" for arithmetic tasks (3 times; 3 critical). For the filtering tasks, they tried (b) "<2014", "1/4", "1<2<3<4", "-1<1", "are>2", and "some>2" (6 times; 3 critical). For the sorting tasks, two participants tried to describe the direction with "increasing order" and "reverse input" (2 times; 1 critical). For examples, see Table 19.

(a)	IN	11,8,9,10	The third row (“T,T”) is unnecessary.
	STEP	T,F,F,T	
	STEP	T,T	
	OUT	11,10	
(b)	IN	1,4,5 3,8,15	To express a conditional “not divisible by 4”, a participant created steps of “4” and “F”.
	STEP	4 4	
	STEP	F F	
	OUT	1,5 3,15	

Table 18. Examples of unnecessary steps

5.6.6. Inconsistent program (3 times; 2 critical)

Even when the PBE engine generated a single program for every step, the entire program could be inconsistent with the task. For instance, participants often created wrong arithmetic (2 times; 2 critical), or filtering programs (1 time).

5.6.7. Empty cases (2 times; 0 critical)

Participants sometimes left the right most case empty.

5.7. LIMITATIONS

We made several simplifying assumptions that limit the scope of our findings. First, to allow non-expert users to quickly learn, the study introduces only a few standard tasks (e.g. arithmetic, string processing, and filtering). While the general patterns of findings will likely apply to other tasks, it will be important to confirm the extent to which this is true.

Second, our experimental system does not show generated programs, while a few PBE systems [35,54] support interactive disambiguation where users read program descriptions and disambiguate by directly choosing a desired program. Further work is needed to explore the opportunity and effectiveness of interactive disambiguation.

Third, we did not collect log data of dropouts, which could explain why they gave up the study. The high dropout rate suggests that an attrition bias might exist between the baseline and the actionable feedback settings.

The study also leads us to a wide research area. For example, how to construct and train a knowledge model of a PBE user is an open-ended research question. How various design factors effect a user's motivation and understanding of the PBE system is the goal of the next chapter.

5.8. CONCLUSION

This chapter presents a user study that examines how inexperienced users learn and use our PBE system. Findings include seven types of common mistakes, and an evidence confirming that we can automatically detect a user's programming intent, and generate actionable feedback that helps the user quickly fix mistakes.

Chapter 6: Experiments on Feedback and Human Mistakes in PBE Systems

6.1. Motivation and Introduction

Human-centered design is an essential factor for the success of any interactive system, but is often overlooked for PBE systems [44]. As reported in Chapter 5, inexperienced PBE users make a wide range of mistakes while decomposing complex tasks and providing unambiguous examples. However, our preliminary user study (Chapter 5) suggests that even unsuccessful examples contain enough clues for detecting a user's programming intent and misunderstanding of the system, and PBE systems can provide useful feedback based on those clues. The result reaffirms a widely known principle - human-readable, informative feedback is crucial for designing usable interfaces [75]. However, there is little prior research about detailed feedback design particularly for PBE users. The goal of our study in this chapter is to address R5 - exploring the design space of feedback.

- R5. What is the impact of feedback design on user's experience of PBE?
- a. Is showing either *system information*, *instruction*, or *both* helpful for completing tasks, understanding the system, and fixing human mistakes?
 - b. Does feedback design affect user's behavior of using PBE features?
 - c. Does feedback design affect user's credibility of the programs they make?
 - d. Does demographic information affect user's performance and behavior of using PBE features?

- e. Is the history of previous trials helpful for users to understand and fix their mistakes?

To answer the above questions, we conducted an online experiment with 133 participants, who were recruited from Amazon Mechanical Turk. The experiment is based on the preliminary study in Chapter 5, with a few modifications and extended features. First, we collected log data from not only those who finished the entire study but also who dropped out in the middle of the study. We compared different feedback design in terms of user's dropout rates (i.e. how far participants proceeded in the tutorials and main tasks), the success rate (i.e. how likely each participant would accomplish each tutorial and task), behavioral metrics (e.g. click rates of the feedback messages), and subjective assessments (e.g. perceived usefulness the system, credibility of programs that participants created). Second, we developed 12 rules for detecting mistakes and generating feedback messages. We believe these findings provide valuable implications for designers of future PBE systems.

6.2. Experimental System UI

To conduct the study, we extended the experimental system used in Chapter 5. The new system has the same goal – to enable non-technical participants to quickly learn and perform decomposition and disambiguation. There are a few extended features. First, if a command is assigned to a step, the description of the command is provided next to the step, as “Calculate Input*2” next in Figure 45.

Second, when the PBE engine cannot find any command from user-provided

TASK tutorial_arith_2

(Input + 1) * 2

EXAMPLES

Input	1	4	+ ADD CASE
Step1	2	8	
Step2	5	3	
Step3	1	0	
Output	4	abc	

✓ Calculate Input * 2 is found.
 ✗ No command is found. What is your intent for the step?
 • "Calculating a multi-step arithmetic from above numbers" The system can only learn a single arithmetic step. Insert a step above that contains intermediate values of the arithmetic that you want.
 ⚠ 4 commands found. Add another case, or CHOOSE AMONG THEM
 ✗ No command is found.

RESULT

Teach Computer Make sure every step has a single command.

Are you having troubles with this task? You can skip this task and move on.

Give up the current task

HISTORY (The latest trial is shown first)

IN 1 4	IN 1 4	IN 1 4	IN 1 4	IN 1 4	IN 1 4	IN 1 4
ST1 2 8 ✓	ST1 2 8 ✓	ST1 5 3 ✗	ST1 5 3 ✗	ST1 3 6 ✓	ST1 3 6 ✓	ST1 3 6 ✓
ST2 5 3 ✗	ST2 5 3 ✗	ST2 1 4 ⚠	ST2 1 4 ⚠	ST2 4 abc ✗	ST2 4 abc ✗	ST2 4 abc ✗
ST3 1 0 ⚠	ST3 1 4 ⚠	OUT 4 abc ✗	OUT 4 abc ✗	OUT 4 abc ✗	OUT 4 abc ✗	OUT 4 abc ✗

Figure 45. The experimental system UI. The TASK section describes the program participants should build. The EXAMPLES contains a table of user-provided examples and feedback from the PBE engine. In the RESULT panel, users press the Teach Computer button to let the PBE engine generate programs based on provided examples, and get feedback. Finally, the HISTORY panel shows all the trials provided for the current task.

examples, it shows a feedback message, “No command is found. What is your intent for

1. To solve a task, users need to make every step has a single command. If users provided examples that are ambiguous, the PBE engine provides feedback as below, “N commands found. Add another case, or CHOOSE AMONG THEM” where N is the number of commands consistent with the examples.

TASKtutorial_arith_1_2

Input + 1

EXAMPLES

Input	1
Output	2

🔍 2 commands found. Add another case, or
CHOOSE AMONG THEM

RESULT

Teach Computer
Make sure every step has a single command.

2. Clicking the “CHOOSE AMONG THEM” button will open a popup that contains the list of generated commands. Users can select a command to lock, or close the popup.

🔍 2 commands found. Add another case, or

All the commands below can calculate the current step. You can click a command to lock.

Calculate Input + 1
Calculate Input * 2

3. A selected command will be locked for the step. Locked commands will not be updated by teaching computer again.

Input	1
Output	2

🔒
Calculate Input + 1 is locked for the step.
UNLOCK

4. Clicking the “UNLOCK” button will remove the locked command. To get a command for the step, users need to teach again.

Input	1
Output	2

🔓
Unlocked. To find matching programs again, press "Teach Computer" button below.

Figure 46. The mechanism of choosing and locking commands for a step. When the computer generates multiple commands users can choose one among them. Chosen commands are locked to the step, and stay until they got unlocked.

the step?”, and a list of potential user intents that it extracted from the examples. For instance, Step2 in Figure 45 shows a potential intent of a user, “*Calculating a multi-step arithmetic from above numbers*”. If the user thinks the intent is correct, he/she will click it to see the relevant information about the system (e.g. “*The system can only learn a single arithmetic step*”), and/or instruction for fixing the examples (e.g. “*Insert a step above that contains intermediate values of the arithmetic that you want.*”)

Lastly, if the PBE engine generates multiple commands for a step, it shows a message “*n commands found. Add another case, or [CHOOSE AMONG THEM].*” Users have two options for fixing it: (1) providing additional cases, (2) choosing among the list of generated programs as illustrated in Figure 46. Fourth, if a user makes more than five unsuccessful trials, the system shows a button for giving up in the RESULT panel. Lastly, the HISTORY panel shows the list of unsuccessful trials that the user has made so far for the current task.

6.3. Feedback rules

In Chapter 5, I identified the seven types of mistakes that inexperienced users make. To detect types of mistakes in the current example, and to provide adequate feedback messages, I developed 12 rules. It has to be noted that the rules are applied only when the PBE engine found no command for the step - not multiple commands. Therefore, the rules do not cover *Ambiguous cases* and *Inconsistent program*, for which the PBE engine generated multiple commands or a single command respectively.

6.3.1. Missing steps

When participants cannot decompose a complex task (i.e. did not create essential steps), the PBE engine would fail to generate any command for the provided examples. To help users understand and decompose tasks by adding essential steps, we developed three feedback rules.

F1. FILTER WITHOUT PREDICATE

The PBE engine requires a predicate step between source and target steps of a filtering task. However, inexperienced users often forget to create a step of predicates. The first

feedback rule detects the mistake using two conditions, in addition to the failure of generating any command. First, the current step must be a target of the filtering task. To check this, values of the current step (e.g. “a, d”) are a subset of values of any source step above (e.g. “a, b, c, d”). The second condition is non-existence of a valid predicate step between the current step and the source. A valid predicate step must have the same shape as the original step. In order for two steps to have the same shape, they must have the same number of cases, and every case must have the same number of values. For instance, the three examples below have the same shape, because all of them have three columns, and all the matching columns have the same number of values.

a, b	c	d, e	T, F	F	T,F	0,0	0	0,0
------	---	------	------	---	-----	-----	---	-----

When the rule is satisfied, the system generates the feedback components below, and show users according to their experimental conditions.

- Intent: *“Trying to filter {source step}.”*
- System Information: *“Filtering requires a predicate step containing T or F for each value.”*
- Instruction: *“Insert a step above and type predicate values. For instance, F,T,F will keep the second values, and filter out the first and the third values.”*

F2. PREDICATE WITHOUT NUMBERS

The PBE engine can evaluate numbers. Thus, any predicate step requires a step that contains key numbers. Our system detects this mistake if two conditions are satisfied. First, the current step must hold predicates only. Second, there is no above step that contains numbers in the same shape as the current step. Two steps having the same shape means that they have the same number of columns (i.e. cases), and corresponding

columns always have the same number of values. When the rule is satisfied, the system generates the feedback as below.

- Intent: *“Predicates for filtering {source step}”* where the *source step* is the closest step above that has the same shape.
- System Information: *“To calculate predicates, it requires numbers above.”*
- Instruction: *“Insert a step above and type key numbers for determining predicates.”*

F3. MULTISTEP ARITHMETIC

T3 and T7 require that participants decompose complex arithmetic tasks such as $(\text{Input}+1)*2$ or $(\text{Input}+1)*(\text{Input}-1)$. However, Inexperienced users often realize they need to add additional steps between the input and output steps. Our system detects this type of mistake if the current step contains only numbers, and there exist a step containing numbers with the same shape. When the rule is satisfied, the system generates the feedback below.

- Intent: *“Calculating a multi-step arithmetic from above numbers”*
- System Information: *“The system can only learn a single arithmetic step.”*
- Instruction: *“Insert a step above that contains intermediate values of the arithmetic that you want.”*

6.3.2. Ambiguous cases

To accomplish a task, users need to specify a single command for every step. However, the PBE engine often generates multiple commands that are consistent with provided examples. In order to fix it, users either provide additional cases (i.e. additional columns in the EXAMPLE table) or manually choosing from the list of generated commands. Since these two solutions are well-explained within their own UIs, we did not create a feedback rule for this type of mistake.

6.3.3. Inconsistent or unsupported values

All the values across columns must be consistent with a command, which is supported by the PBE engine. However, we observed that inexperienced users make a wide range of mistakes. We created five rules (F4-F8) for detecting and generating feedback for inconsistent or unsupported values.

F4. INCONSISTENT CASES

When our participants provided multiple cases for disambiguation, they often gave a value inconsistent with the others. It took many trials until they noticed the mistakes. Our system detects inconsistent cases using the *leave-one-out* cross-validation technique [41]. To begin with, the step must contain at least three cases. Second, the PBE engine tries to generate commands multiple times leaving one case out of the examples. For instance, if the current step contains three cases, the PBE engine tries to generate commands three times using (1,2), (1,3), and (2,3). If it generates an alternative command from (2,3), which left out the first case, the system will create a feedback message as follows.

- Intent: *“Trying to teach {alternative command}”*
- System Information: *“It cannot learn when cases are inconsistent.”*
- Instruction: *“Consider fixing or removing the 1st case.”*

F5. NON MATCHING SIZE OF PREDICATE

A predicate step must have exactly the same shape with the step to be filtered. However, inexperienced users often use their creativity to give examples that the PBE engine cannot comprehend. For instance, in the previous user study (Chapter 5), we observed 7 (out of 150) cases in which participants provided single predicate values. To detect this mistake, the system checks whether the current step contains predicate values, and there is no step above has values of the same shape. If the rules are satisfied, the system generates feedback as follows.

- Intent: *“Making predicates for filtering”*
- System Information:
“Predicates and items to be filtered must have the same length.”
- Instruction: *“Modify this step to have predicates (T or F) for every value in the step to be filtered.”*

F6. PREDICATE AND VALUE COMBINED

PBE users often create values in their own format. For example, we observed two (out of 150) cases in which participants provided predicates and values to be filtered combined (e.g. “T11, T10, F8, F9”; indicating that 11 and 10 are true, 8 and 9 are false cases). Our system detects this mistake by testing that the current step begins with a predicate value (e.g. “T”, “F”, “t”, or “f”) but also contains non-predicate values. If the rule is satisfied, the system generates feedback as follows.

- Intent: *“These are predicates with values combined”*
- System Information: *“However, predicates must be T and F separated by commas.”*
- Instruction: *“Modify this step to have predicates (T or F) for every value to be filtered.”*

F7. LIST WITHOUT SEPARATOR

PBE users often provide predicate values without a separator, such as “FTFT”. Detecting this mistake is simple: the current step data must consist of predicate values (e.g. “T”, “F”, “t”, or “f”) only. If the rule is satisfied, the system creates the *{correct list}* by adding separators between every pair of adjacent characters, and generates feedback as follows.

- Intent: *“Predicates for filtering”*
- System Information: *“Values in a list must be separated by a comma (,).”*
- Instruction: *“Modify the value to {correct list}.”*

F8. YES NO PREDICATE

Users can create a predicate step with “yes” and “no” – instead of “T” and “F”. This is a very rare mistake, which happened only once. To detect this mistake, our system uses a regular expression that checks whether the current step contains “yes” or “no” separated by commas. When the rule is satisfied, it can automatically generate *{corrected predicates}* by replacing “yes” to “T”, and “no” to “F”, and generate feedback as follows.

- Intent: *“Predicates for filtering”*
- System Information: *“But predicates must be a list of T and F separated by commas.”*
- Instruction: *“Modify the value to {corrected predicates}.”*

6.3.4. Unnecessary steps

F9. UNUSED STEP

Users often create steps that are unnecessary for calculating the output. Although unused steps do not fail the task, it is better to remove them for clarity. To evaluate whether the current step is unused, the system checks two conditions. First, the output must have a single command. Second, the current step is not in the ancestors of the output. When the rule is satisfied, the system generates feedback as follows.

- Intent: *“I have no intent for this step”*
- System Information: *“This step is unnecessary for getting the output.”*
- Instruction: *“You may remove the step to simplify your program.”*

F10. EMPTY STEP

Users often leave a step completely empty, especially when they are exploring to the solution. However, since empty steps will fail to generate any command, the system provides the following feedback for empty steps.

- Intent: *“I left this step empty”*
- System Information: *“The system cannot learn any program from an empty step.”*
- Instruction: *“Consider adding relevant values or removing the step.”*

6.3.5. Describing with formula

F11. FORMULA

Although the PBE engine accepts example values only, PBE users often provide formulas. To detect formulas, the system uses a regular expression of common operators (\backslash , $<$, $>$, $=$, $+$, $-$, $*$, STEP, INPUT). If the current step contains any of those common operators, it generates feedback messages as follows.

- Intent: *“I described the step using formula”*
- System Information: *“The system cannot understand formula.”*
- Instruction: *“Give values that your formula calculates.”*

6.3.6. Inconsistent program

This is the case when the PBE engine can generate single programs for every step, but the entire program is wrong. We do not have any feedback rule for this type of mistake.

6.3.7. Empty cases

F12. EMPTY CASES

During the preliminary study we observed that participants often provided incomplete tables having a few empty cells. Since the PBE engine requires consistency across columns, a step with empty cases might fail to generate any command. Detecting this mistake is straightforward; the system checks whether a column is empty. However, this rule can make false positive errors, because cases are often empty for good reasons especially when they are the results of filtering. The system generates feedback message as follow.

- Intent: *“I left some columns empty on purpose”*
- System Information: *“To teach a program, all the cases must be consistent.”*
- Instruction: *“Type consistent values in every case, or remove unnecessary columns.”*

6.4. Methods

6.4.1. Procedure

To participate, participants clicked the hyperlink to our system posted on Amazon Mechanical Turk (AMT). In the landing page, they read the consent form and clicked the “I agree” button to proceed. In the second page, they filled in the demographic survey form, which asked their age, gender, and the highest level of education, major, current occupation, programming expertise, and technical experience. After the survey, participants learned basic usage and the concept of example-based programming through the six tutorials. They then proceeded to the five main tasks. Both tutorials and tasks are the same as the preliminary study in Chapter 5. Finally, participants were asked to fill in a closing survey about perceived usability and effectiveness of the system.

6.4.2. Closing survey

After finishing the entire study, we gave participants several questions about their experience and opinions. The five questions were about how much they agreed with the following statements.

- *The system was easy to understand.*
- *The interface was effective to accomplish the tasks.*
- *The feedback next to each row was helpful.*
- *The programs I taught will work correctly for wider ranges of inputs.*

Lastly we asked them to give us general comments.

- *Do you have any other comments on what worked or didn't work about the system?*

6.4.3. Compensation

During the preliminary study we observed that a lot of participants from AMT dropped out. Through a few rounds of pilot studies, we figured out a reasonable multi-stage compensation policy. Participants received a \$1 basic reward as they finish the tutorials. Those who finished the entire study, no matter how many tasks they gave up, received a \$2 bonus. In addition, we gave \$1 extra bonus to the best performing one among every 10 participants.

6.4.4. Experimental design

To explore the design space of the feedback mechanism, we chose two factors: feedback components and the history panel. For the first factor, our feedback rules generate feedback messages as three components: *intent*, *system information*, and *instruction*, as described in section 6.3. Although it is possible to make a maximum of eight combinations from three items, *detected intent* is an essential component, which must be included in all experimental conditions, in order to enable users to choose the

right feedback. Therefore, we created four conditions of feedback components as follows.

- (BASELINE): The baseline setting shows no feedback component.
 - **⊗ No command is found.**
- (SYSTEM INFO): Shows an *intent* first. As users click the intent, it reveals the relevant *system information*.
 - **⊗ No command is found. What is your intent for the step?**
 - **"Trying to filter Input"** Filtering requires a predicate step containing T or F for each value.
- (INSTRUCTION): Shows an *intent* first. As users click the intent, it reveals the *instruction* for fixing the example.
 - **⊗ No command is found. What is your intent for the step?**
 - **"Trying to filter Input"** Insert a step above and type predicate values. For instance, F,T,F will keep the second values. and filter out the first and the third values.
- (BOTH): Shows an *intent* first. As users clicked the intent, it reveals the both *system information* and *instruction*.
 - **⊗ No command is found. What is your intent for the step?**
 - **"Trying to filter Input"** Filtering requires a predicate step containing T or F for each value. Insert a step above and type predicate values. For instance, F,T,F will keep the second values, and filter out the first and the third values.

The second factor is whether the UI shows the history panel or not. A participant can see his/her previous trials for the current task in the history panel.

The study uses the between-subject design. When a new participant visits, our system randomly assigns one of the eight conditions (4 feedback components * 2 history panel settings).

6.4.5. Measurements

The research questions focused on user's performance in relation with various feedback components. We measured each participant's *progress* - how many tutorials and tasks

he/she finished the entire study or stopped participating at a specific task. The system also collected the *success rate* - whether participants *passed* or *gave up* each task they finished. In addition, the system collected various information of each feedback rule, such as *frequency* (i.e. how many times the rule is activated) and *click rates* (i.e. % of feedback messages clicked by participants). The system also measured user's behavior of using UI components such as "add a case", "add a step", "choose among them", etc.

6.4.6. Participants

We recruited participants from AMT. The only constraint was that participants must reside in North America. We kept posting batches of our HIT until we reached around 35 participants for each feedback setting.

6.5. Result

6.5.1. The insignificant impact of feedback messages on completion and success rates

133 participants started the study. 61.5% finished the tutorials, and 30.7% finished the entire study. The biggest portion (26.5%) of participants dropped out while doing T3, which is the first tutorial they learned decomposition of complex arithmetic "(Input+1)*2". The second most difficult task was T7, the first main task, where 29.5% among participants who reached T7 dropped out.

We expected dropout rates indirectly indicate the performance of each feedback setting. If a condition is better than the others, participants using the condition would be more likely to finish the entire study. It seems that Figure 47 supports the hypothesis - the BOTH condition (shown as a green line) was above the other conditions for the main tasks T8-T11. However, Pearson's Chi-squared tests do not support a significant

difference between pairs of conditions. For example, even the best (BOTH) and the worst (BASELINE) performing conditions do not have significantly different impacts on the probability of the participants to reach the final task, $\chi^2(2, N = 67) = 2.5519, p > .1$. Mann-Whitney U test also do not tell significant difference between conditions, $Z = 1.0858, p > .1$. We also performed Kruskal-Wallis test on the number of tasks that participants reached, but could not find a significant difference between conditions, $\chi^2(10, N = 67) = 14.891, p > .1$. Similarly, feedback conditions do not have significant impact on the number of tasks that participants successfully finished, $\chi^2(10, N = 67) = 2.393, p = .495 > .1$

We compared the two conditions about the history panel in Figure 48. It seems that the history panel helped users keep going after T3, but through T6 and T7, they converged into one.

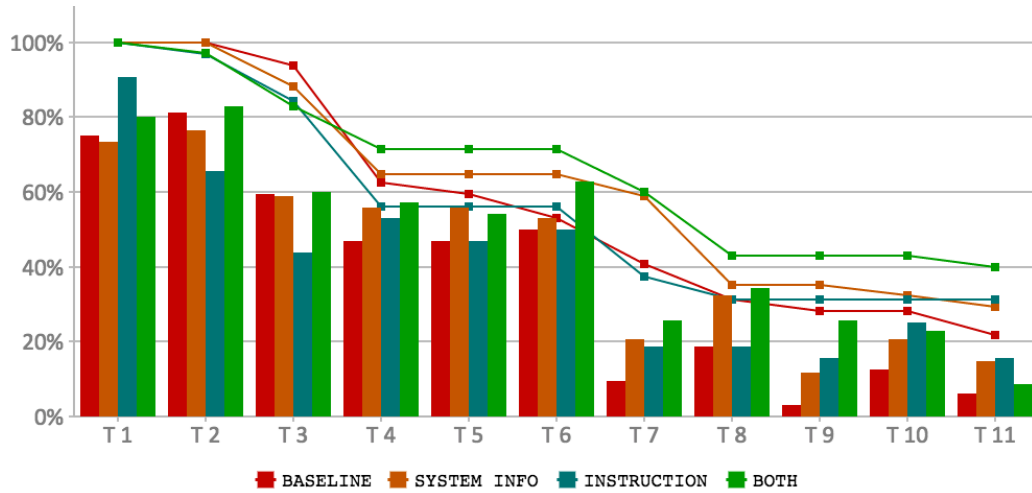


Figure 47. Probabilities of participants reaching and completing tasks compared across different feedback compositions. Lines indicate the portion of participants who reached specific tasks. Bars indicate the portions of participants who accomplished tasks without giving up. The green line above the other lines suggests that the 'BOTH' setting, which shows both system info and instruction, outperformed the other settings.

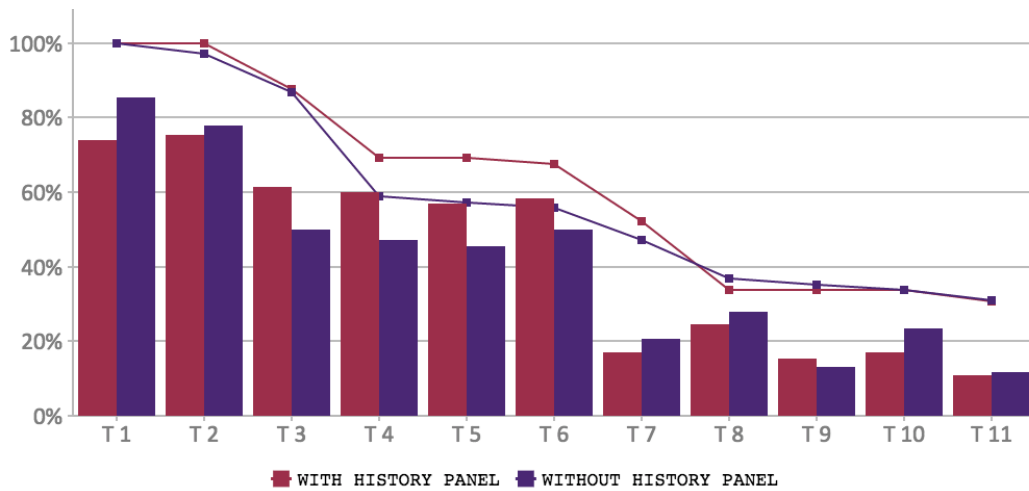


Figure 48. Probabilities of participants reaching and completing tasks compared to whether the history panel is given or not. Lines indicate the portion of participants who reached specific tasks. Bars indicate the portions of participants who accomplished tasks without giving up. The two lines go along with each other, suggesting that the history panel does not have a strong impact on how many users reached and completed tasks.

6.5.2. Frequency and click rates of feedback

The system shows feedback messages that 12 feedback rules generate based on user-provided examples. This section analyzes the log data of 876 tasks focusing on how frequently each feedback rule was activated (i.e. shown), and how frequently participants clicked them to read system information and instruction components¹⁵. In total, feedback rules were activated for 314 times, and clicked 115 (36.6% click rate) times. As shown in Figure 50, **R1. Filter without predicate** is the most frequently activated (60 times) and clicked (25 times). **R2. Predicate without number** was activated less frequently (35 times), but clicked as many times as R1. The three rules (**R1-R3**) for the *Missing Step* mistakes were clicked 49.3% of the tasks they were shown, which is higher than the average click rate (36.6%).

R5-R8 were activated much less (<10 times) than other rules. The rules were created for the **Inconsistent and unsupported values** type of mistakes, which occurred less frequently than we saw in the preliminary study. It does not necessarily mean that R5-R8 are less useful than the other rules.

R9-R12 were shown 25-40 times, and clicked 7-12 times. The 27.3% click rate is lower than the average click rate (36.6%).

¹⁵ Note that no matter how many times a rule is activated or clicked within one task, we counted it as one activation or click. This is because users often try a task repeatedly (>50 times).

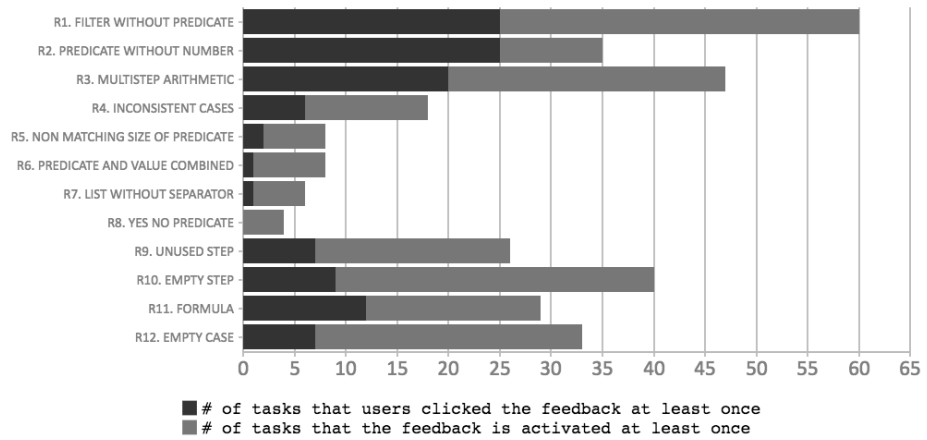


Figure 49. # of tasks (and tutorials) that a specific feedback rule was activated and clicked by participants.

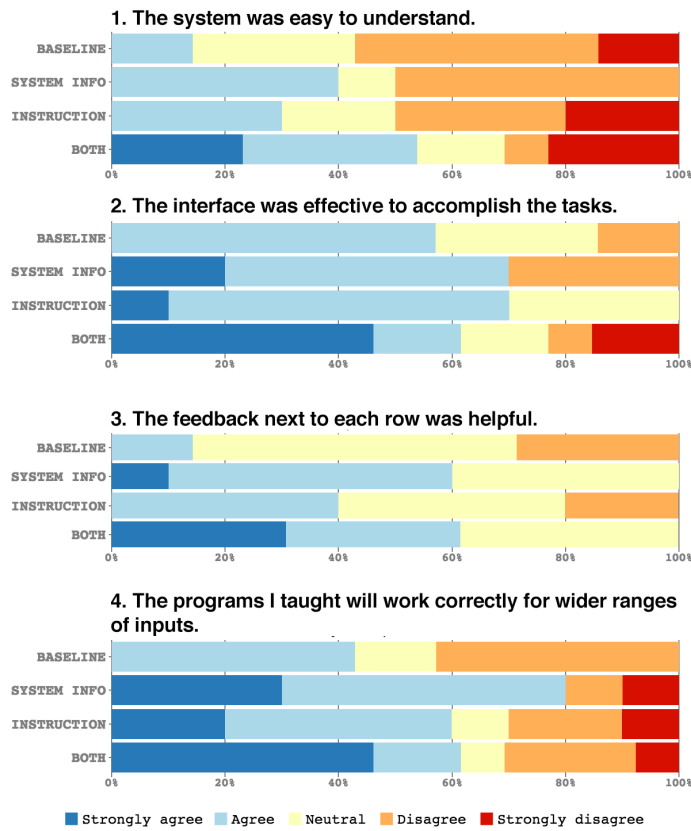


Figure 50. The closing survey result. The Likert scale ratings generally suggest that the BOTH condition is perceived to be intuitive, effective, and useful to increase the credibility of outcome. However, a few participants perceived the BOTH condition to be hard to understand and ineffective.

6.5.3. Perceived quality of the system and the outcomes

After finishing the main tasks, participants filled a closing survey about the effectiveness of the system, and the generalizability of the programs they created, as illustrated in Figure 50. We compared raw frequency of answers, and then conducted Kruskal-Wallis test to see whether the differences are significant. It has to be noted that there is an attrition bias across conditions because of the high dropout rate. For example, only seven participants in the BASELINE condition answered the closing survey, while 10, 10, and 13 participants answered the survey for SYSTEM INFO, INSTRUCTION, and BOTH conditions respectively.

The first question was about how they perceived the usability of the system. The majority of participants using the first three conditions (BASELINE, SYSTEM INFO, and INSTRUCTION) gave negative (“disagree” or “strongly disagree”) answers. In contrast, participants who used the BOTH condition gave either strongly positive or strongly negative ratings. However, the difference is not statistically significant by feedback conditions, $\chi^2 = 2.203, p = .5313$, at the $\alpha = 0.05$ significance level.

The second question was about the effectiveness of the UI. Participants gave similar but a bit more positive ratings than the first question. Ratings on the BOTH condition were again polarized into positive and negative opinions. However, the difference is not statistically significant by feedback conditions, $\chi^2 = 1.003, p = .8005$, at the $\alpha = 0.05$ significance level.

The third question was about the effectiveness of the feedback. More than 60% of participants who used the BOTH and the SYSTEM INFO conditions rated positively, and no participant gave the conditions a negative rating. The difference is statistically

significant, $\chi^2 = 8.266, p = .0408$, at the $\alpha = 0.05$ significance level. A post-hoc analysis using Dunn's test adjusted by the Benjamini-Hochberg FDR shows that the BASELINE and BOTH are significantly different ($p = .0593$).

The last question was about how confident participants felt about the programs they created. More than 60% of the participants who used three conditions except the BASELINE rated their programs positively. 45% of the participants who used the BOTH condition strongly agreed that they trusted their programs. Around 10% of the participants who used the three non-BASELINE conditions were quite negative as well. The difference is not statistically significant by feedback conditions, $\chi^2 = 2.461, p = .4824$, at the $\alpha = 0.1$ significance level.

To sum up, we found a statistically significant difference between the BASELINE and the BOTH conditions.

6.5.4. Participant background and behavior

133 participants (80 males, 53 females) were recruited via AMT. They were on average 34.6 years old ($SD = 10.33$, range 20-70) and all currently live in the United States or Canada. The majority (73) of participants have bachelor degrees, 41 graduate high school, 7 have master degrees, 11 have professional degrees, and 1 has a doctoral degree. In terms of programming experience, 48 participants have no programming knowledge, while 53 know basic concepts. Participants also include 23 amateur programmers and 9 reported that they are professional programmers.

We conducted rank-order correlation tests to check whether participants' progress and demographic information are positively or negatively correlated. First, Participants' gender and progress were not significantly correlated, $Z = 0.5053, p = .6101 < .05$.

Age did not affect their progress, $r_s = -0.1235, p = .1566 > .05$. Participants' education level have almost significant positive correlation, $r_s = 0.1695, p = .0510 > .05$. Lastly, their programming expertise has a significant positive correlation with progress, $r_s = 0.1844, p = .0333 < .05$.

6.6. Discussion

6.6.1. The insignificant impact of feedback messages

We observed that drop out rates in the experiment are statistical indifferent across feedback conditions, although the difference between conditions are visible in Figure 47. The finding is contradictory to another finding from the online user study in Chapter 5.5. There are potential reasons of the contradiction. First, it is possible that the experimental system has many features (e.g. choosing among generated programs), added or extended to the online study, which can weaken the impact of feedback conditions on both completion and success rates. It is possible that the UI get overloaded with too much information that participants ignored feedback messages.

To confirm whether feedback settings have significant impacts on user's performance, we have a few options of follow-up studies. First, we can collect more data to see whether the feedback settings gain the statistical power or not. Second, we can get rid of a few irrelevant information from the system so that participants are not overloaded with too much information. Third, we can analyze detailed log data to investigate what event occurred before participant dropping out.

6.6.2. Potential reasons and remedies for the high dropout rate

Among 131 participants recruited from AMT, only 69.3% dropped out before finishing the entire study. For most user studies, a low attrition rate (i.e. high dropout rate) is an

alarming signal that indicates serious issues might exist in the study design. In my study, there are several factors that possibly contributed to the high dropout rate.

First, participants might have dropped the study because our tutorials and tasks were too challenging. I believe this is highly likely because the two biggest dropouts happened when participants first learned the concept of decomposition (T3 and T7), and 15.5% of participants dropped after finishing the tutorials, which suggests that they were demotivated not to do the main tasks. Although an immediate fix of this issue is to use shorter and simpler tasks and tutorials, we need to consider that the aim of the experiment is to give users sufficiently hard tasks so that they need additional supports.

Second, the experimental UI and instruction might have room for improvements. This reason is also likely because participants made mistakes even though they had already read the relevant instruction. Why did they miss the relevant information? Information overload can be a potential reason, as discussed in 6.6.1. To improve the design quality of UI and instruction, we could have conducted more pilot studies in the lab.

Third, AMT may not be the best platform to recruit participants for the experiment, which requires them to learn a lot of new concepts such as example-based programming, and managing steps and cases. Many experimental studies using online samples (e.g. Amazon Mechanical Turk) often do not report attrition rates, which can range from 30% to 50% and vary across experimental conditions [94]. If this is the dominant reason for the high dropout rate, improving UI and instruction will have only limited effectiveness. The best way to fix it is to conduct a lab study - which is also the best way to find out the actual problem. However, conducting a lab study requires much more resources

than running an online experiment. A more economic way is to add a screening test, which estimates user's ability to finish the experiment, and to allow only participants who pass the screening.

6.6.3. Plan for a follow-up experiment: addressing the high dropout rate

This section describes plan for a follow-up experiment addressing the high dropout rate.

Simplify the study through an iterative design process

To identify and fix potential usability issues in the experimental system, I will go through a few rounds of iterative design process that consists of in-person pilot studies and redesigning the tasks, UI, and instruction. While redesigning the system, I will consider two options for simplifying the study. First, I will observe participants to identify redundant or unnecessary part of the UI, and remove them to prevent information overloads. Second, I will consider breaking the current tasks (and tutorials) into a few groups so that each participant can finish the experiment with less time and effort.

Diversify the population and the study setting

Who the participants are, and where the experiment is conducted may have a significant impact on the result. To control the impact, I will conduct the study with two populations: (P1) Amazon Mechanical Turk, and (P2) campus mailing list. Another factor is the study setting. I will conduct the study with P2 in two environments: (E1) in-person lab study, and (E2) remotely. P1 will always participate remotely. Cross validation of the results will give insights of how population and study setting affect the dropout rate.

Screening questions

I found out that programming expertise is not perfect but still a good indicator of participants' completion. To lower the dropout rate, I will carefully add a few screening questions about their ability to understand the basic concept of PBE tasks and usage of the system. The first question will be about the participant's programming experience. It will require participants to know at least basic concepts of programming. Second, I will give five tests about PBE. Each test will show four examples of input and output values, and participants must pick a matching program that calculates all the input and output examples among four programs, as illustrated below. Participants who answered all the questions correctly will be able to proceed to the tutorials.

Input => Output
1 => 3
2 => 4
3 => 5
4 => 6
Choose the right program that matches the examples above
(a) Input * 3 (b) Input + 2 (c) Input +1 (d) Input * 4 - 2

Closing survey when a participant dropout

The closing survey is an important source of information, but the current system does not collect from dropouts. I will redesign the system so that participants are asked to fill in the closing survey for the HIT completion code, even when they want to stop participating.

Chapter 7: Conclusion

The goal of this dissertation was to improve the human-centered design of PBE systems by studying users' needs and mental models, identifying usability issues and human mistakes, and developing and testing novel features. In this chapter, we summarize what we have learned in response to the research questions, thesis contributions, and directions for future work.

7.1. Answers to the research questions

7.1.1. R1. What do end-user programmer need to improve the Web?

To answer the question, we conducted a semi-structured interview study with 35 end-users of the Web, as presented in section 3.2. The interview study explored the space of challenges that end-users regularly experience on the Web, and the functionalities of enhancements that they envisioned. We proposed seven categories of enhancements (*Modify, Compute, Interact, Gather, Automate, Store and Notify*), which provide guidance to website designers in the first place to be aware of the unique needs of many users.

7.1.2. R2. How do non-programmers express their programming intent?

To answer the question, we conducted a Wizard of Oz study (section 3.3) that asked non-programmers to express computational tasks. We found interesting characteristics of them. First, non-programmers would express their intent effectively using multiple channels such as rules, examples, and rationales. Although they may not be able to provide complete information at first, they can iteratively refine their intent with additional information. To enable non-programmers to express high-quality intent,

future EUP tools should incorporate mixed-initiative interaction to help end-users express unambiguous statements.

7.1.3. R3. Is PBE better than direct specification?

To answer the question, we conducted a preliminary user study with two versions of VESPY (Chapter 4). We could not find a clear answer, such as “PBE is always better than direct specification” or the opposite. Instead we observed that PBE is effective for complex tasks where users can skip multiple steps of interaction. Thus the alternative answer to the question is that the usefulness of PBE is affected by many factors: (1) user’s knowledge of the domain-specific language, the PBE engine, and the task; (2) the amount of work for creating sufficient examples vs. directly specifying parameters; (3) credibility of programs.

7.1.4. R4. Can inexperienced users perform problem decomposition and disambiguation?

The answer was “No”. As reported in Chapter 5, we observed that only 30 out of the 161 participants finished the entire study. We also identified seven types of common mistakes: *Missing steps*, *Ambiguous cases*, *Inconsistent or unsupported values*, *Unnecessary steps*, *Describing with formula*, *Inconsistent programs*, and *Empty cases*. However, we also found empirical evidence that the PBE system can automatically detect a user’s programming intent, and generate actionable feedback that helps the user quickly fix mistakes.

7.1.5. R5. What is the best feedback design for PBE users?

To answer the last question, we conducted an online experiment with 133 participants. We developed 12 rules for detecting mistakes and generating feedback messages, three

components of feedback messages, and the history panel showing the previous trials. According to Figure 47, the BOTH condition seems to outperform the others, and the BASELINE was the worst setting. However, statistical tests could not confirm that the numbers of completed tasks are significantly different across conditions. We also compared the closing survey result across feedback conditions, and found out that only the third question was significant - i.e. participants rated the perceived effectiveness of feedback in the BOTH condition higher than the others. Participants did not give significantly different ratings for the system's intuitiveness and their credibility of the programs. We discussed about a few potential reasons of the insignificant differences, in relation with the high dropout rate.

To investigate whether personal background affect user's performance, we conducted rank-order correlation tests on demographic information and the number of completed tasks. While age, gender, and education level did not have significant impacts on user's performance, programming expertise is helpful to complete more tasks.

7.2. Thesis contributions

7.2.1. Identification of unmet needs of end-users of the Web

End-user programming (EUP) is a common approach for helping ordinary people create small programs for their professional or daily tasks. However, it is often hard to address these needs, especially for fast-evolving domains such as the Web. We conducted a semi-structured interview study (Chapter 3.2) with 35 end-users of the Web. The interview study explored the space of challenges that end-users regularly experience on the Web, and the functionalities of enhancements that they envisioned.

We identified seven categories of enhancements that can provide guidance to future EUP developers.

7.2.2. Characterization of non-programmers' mental model

Programming is difficult to learn since its fundamental structure (e.g. looping, if-then conditional, and variable referencing) is not familiar or natural for non-programmers [67]. Understanding a non-programmer's mindset is an important step to develop an easy-to-learn programming environment. We conducted a Wizard of Oz study (Chapter 3.3), which provided characteristics of non-programmers explaining how they would express their intent of computational tasks. Given that traditional programming environments do not fully support them, we discussed the implications for the design of multi-modal and mixed-initiative approaches for making end-user programming more natural and easy-to-use for these users.

7.2.3. Design process of interleaving visual programming and PBE

Researchers and companies have developed many PBE systems, but how to design UI to support users to decompose and disambiguate complex tasks is still an open-ended research question. Through a 1.5 year-long iterative process, we developed VESPY UI (Chapter 4) in which users decompose complex tasks into tractable modules (using visual / dataflow programming techniques), and generate solutions for each module (using PBE techniques). We believe the design process and the final outcome would be valuable resources for future PBE system designers.

7.2.4. Identification of human mistakes of PBE

PBE systems can be challenging for inexperienced users. Unfortunately, there is little research on people's ability to accomplish complex tasks by providing examples. We conducted an online user study that investigates how well people decompose complex tasks, and disambiguate sub-tasks. We also identified seven types of mistakes made, and suggested new opportunities for actionable feedback based on unsuccessful examples.

7.2.5. Design and assessment of feedback for PBE users

While human-readable, informative feedback is crucial for designing usable interfaces [75], there is little prior research about feedback design for PBE users. To explore the design space of feedback, in Chapter 6, we designed three components of feedback messages: *user intent*, *system information*, and *instruction*. We also proposed a *history* panel that shows previous trials of the user. To assess their impacts on user's performance, we conducted an online experiment. The findings suggest that the feedback messages do not significantly affect participants' performance, but providing both system information and instruction increases the perceived effectiveness of feedback messages. The result also suggests that the high dropout rates and information overloads lowered the validity of the study, we will conduct a follow-up experiment with a revised system and study design.

7.3. Future work

With the investigations and designs presented in this dissertation, I have demonstrated that human-centered aspects of PBE can be improved with mixed-initiative, actionable

feedback for human mistakes. From here I present several directions for continued research.

7.3.1. Crowdsourcing feedback rules to users

We have shown that actionable feedback messages are essential to inexperienced users of PBE systems. However, since the current set of rules are manually created by the author, they may not be scalable or generalizable to other PBE systems. To overcome this limitation, designers of future PBE systems can consider crowdsourcing feedback rules. For example, if a lot of users make similar mistakes that the current set of rules cannot detect, the system can ask users provide structured hints. Based on multiple hints, the PBE can automatically create a new feedback rule.

7.3.2. Balancing between too much or too little feedback to users

Although a main benefit of PBE is that it requires users to learn little additional knowledge, we observed that inexperienced users could not provide high-quality examples without proper feedback. In contrast, most features proposed in this dissertation (e.g. adding / removing steps and cases, feedback messages) add a significant amount of information to the system. We observed in the last study that overloading users with too much information can result in negative results. Providing a right amount of information is an important decision for designing usable PBE systems. There are a few directions of future work for the issue. First, we need a metric to monitor whether the current feedback gives too much or too little information. Second, we need a metric to assess the importance of different information so that we can stress the most important point. Third, we need a new interaction model that

initially reveals a small portion of information in-situ, and users can gradually learn the system without getting overloaded with irrelevant or too much information.

7.3.3. Long-term user study of practical EUP systems

Although the last two studies (Chapter 5 and 6) were motivated by the usability issues of VESPY (Chapter 5), we did not have a chance to apply our findings to VESPY. I would like to improve the usability of VESPY with actionable feedback components, and conduct a long-term user study of how users gradually learn the capability of PBE based on what feedback they get. A long-term users study will give an opportunity of crowdsourcing feedback rules, which is explained in 7.3.1

7.4. Final remarks

We are in the early stage of a widespread adoption of automated systems including PBE engines, statistical models, intelligent agents, and more. As we interact with automated systems more frequently, the importance of symbiotic interaction between human minds and automated systems will only increase. Without symbiotic interaction, humans would risk blindly accepting what automated systems suggest, or rejecting them without reasoning. In this dissertation, we have provided a variety of insights into users' needs, mental model, and mistakes. We also have proposed several ways toward symbiotic interaction including VESPY's interleaved UI, the feedback rules, and the history panel. I plan to continue this research with the goal of further exploring the design space of symbiotic interaction between human and AI.

Bibliography

1. Daniel L Ashbrook, James R Clawson, Kent Lyons, Thad E. Starner, and Nirmal Patel. 2008. Quickdraw: The Impact of Mobility and On-Body Placement on Device Access Time. *Proceeding of the twenty-sixth annual CHI conference on Human factors in computing systems - CHI '08*: 219–222. <https://doi.org/10.1145/1357054.1357092>
2. A. Begel and Mitchel Resnick. 1996. LogoBlocks: A Graphical Programming Language for Interacting with the World. In *MIT Media Lab*.
3. Alan W Biermann, Bruce W Ballard, and Anne H Sigmon. 1983. An experimental study of natural language programming. *International journal of man-machine studies* 18, 1: 71–87.
4. A. Blackwell and M. Burnett. 2002. Applying attention investment to end-user programming. In *IEEE 2002 Symposia on Human Centric Computing Languages and Environments, 2002. Proceedings,* 28–30. <https://doi.org/10.1109/HCC.2002.1046337>
5. C. Bogart, M. Burnett, A. Cypher, and C. Scaffidi. 2008. End-user programming in the wild: A field study of CoScripter scripts. In *IEEE Symposium on Visual Languages and Human-Centric Computing, 2008. VL/HCC 2008,* 39–46. <https://doi.org/10.1109/VLHCC.2008.4639056>
6. Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. 2005. Automation and customization of rendered web pages. In *Proceedings of the 18th annual ACM symposium on User interface software and technology (UIST '05),* 163–172. <https://doi.org/10.1145/1095034.1095062>
7. Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. 2005. Automation and customization of rendered web pages. In (UIST '05), 163–172. <https://doi.org/10.1145/1095034.1095062>
8. Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology* 3, 2: 77–101. <https://doi.org/10.1191/1478088706qp063oa>
9. Margaret M. Burnett and Christopher Scaffidi. 2014. End-User Development. *The Encyclopedia of Human-Computer Interaction, 2nd Ed.* Retrieved June 7, 2015 from /encyclopedia/end-user_development.html
10. Amedeo Cesta. 1998. Mixed-Initiative Issues in an Agent-Based Meeting Scheduler. In *No.1-2, pp 45 – 78,* 45–78.
11. Pern Hui Chia, Yusuke Yamamoto, and N. Asokan. 2012. Is This App Safe?: A Large Scale Study on Application Permissions and Risk Signals. In *Proceedings of the 21st International Conference on World Wide Web (WWW '12),* 311–320. <https://doi.org/10.1145/2187836.2187879>
12. Cynthia L. Corritore, Beverly Kracher, and Susan Wiedenbeck. 2003. On-line trust: concepts, evolving themes, a model. *International Journal of Human-Computer Studies* 58, 6: 737–758. [https://doi.org/10.1016/S1071-5819\(03\)00041-7](https://doi.org/10.1016/S1071-5819(03)00041-7)
13. Allen Cypher, Mira Dontcheva, Tessa Lau, and Jeffrey Nichols. 2010. *No Code Required: Giving Users Tools to Transform the Web.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

14. Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky (eds.). 1993. *Watch what I do: programming by demonstration*. MIT Press, Cambridge, MA, USA.
15. Rob Ennals, Eric Brewer, Minos Garofalakis, Michael Shadle, and Prashant Gandhi. 2007. Intel Mash Maker: join the web. *SIGMOD Rec.* 36, 4: 27–33. <https://doi.org/10.1145/1361348.1361355>
16. Gerhard Fischer and Elisa Giaccardi. 2006. Meta-design: A Framework for the Future of End-User Development. In *End User Development*, Henry Lieberman, Fabio Paternò and Volker Wulf (eds.). Springer Netherlands, 427–457. Retrieved September 5, 2014 from http://link.springer.com/chapter/10.1007/1-4020-5386-X_19
17. Gerhard Fischer, Andreas C. Lemke, Thomas Mastaglio, and Anders I. Morch. 1990. Using Critics to Empower Users. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '90)*, 337–347. <https://doi.org/10.1145/97243.97305>
18. Gerhard Fischer, Andreas C. Lemke, Thomas Mastaglio, and Andres I. Morch. 1991. The Role of Critiquing in Cooperative Problem Solving. *ACM Trans. Inf. Syst.* 9, 2: 123–151. <https://doi.org/10.1145/123078.128727>
19. Marc Fisher, II, Mingming Cao, Gregg Rothermel, Darren Brown, Curtis R. Cook, and Margaret M. Burnett. 2006. Integrating automated test generation into the WYSIWYT spreadsheet testing methodology. *Acm Trans. Softw. Eng. Methodol* 15: 2006.
20. Mihai Boicu Gheorghe Tecuci. 2007. Seven Aspects of Mixed-Initiative Reasoning: An Introduction to this Special Issue on Mixed-Initiative Assistants. *AI Magazine* 28: 11–12.
21. J. Gindling, A. Ioannidou, J. Loh, O. Lokkebo, and A. Repenning. 1995. LEGOsheets: A Rule-based Programming, Simulation and Manipulation Environment for the LEGO Programmable Brick. In *Proceedings of the 11th International IEEE Symposium on Visual Languages (VL '95)*, 172–. Retrieved October 27, 2014 from <http://dl.acm.org/citation.cfm?id=832276.834311>
22. Daniel G. Goldstein, R. Preston McAfee, and Siddharth Suri. 2013. The Cost of Annoying Ads. In *Proceedings of the 22Nd International Conference on World Wide Web (WWW '13)*, 459–470. Retrieved April 13, 2015 from <http://dl.acm.org/citation.cfm?id=2488388.2488429>
23. Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *SIGPLAN Not.* 46, 1: 317–330. <https://doi.org/10.1145/1925844.1926423>
24. Sumit Gulwani. 2016. Programming by Examples (and its applications in Data Wrangling). In *Verification and Synthesis of Correct and Secure Systems*, Javier Esparza, Orna Grumberg and Salomon Sickert (eds.). IOS Press.
25. Philip J. Guo, Sean Kandel, Joseph M. Hellerstein, and Jeffrey Heer. 2011. Proactive wrangling: mixed-initiative end-user programming of data transformation scripts. In (UIST '11), 65–74. <https://doi.org/10.1145/2047196.2047205>
26. Marti A. Hearst. 1999. Mixed-initiative interaction. *IEEE Intelligent Systems* 14: 14–23.

27. Eric Horvitz. 1999. Principles of Mixed-initiative User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '99)*, 159–166. <https://doi.org/10.1145/302979.303030>
28. Eric Horvitz. 1999. Principles of mixed-initiative user interfaces. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems (CHI '99)*, 159–166. <https://doi.org/10.1145/302979.303030>
29. Eric J. Horvitz. 2007. Reflections on Challenges and Promises of Mixed-Initiative Interaction. *AI Magazine* 28, 2: 3. <https://doi.org/10.1609/aimag.v28i2.2036>
30. Daniel J. Hruschka, Deborah Schwartz, Daphne Cobb St.John, Erin Picone-Decaro, Richard A. Jenkins, and James W. Carey. 2004. Reliability in Coding Open-Ended Data: Lessons Learned from HIV Behavioral Research. *Field Methods* 16, 3: 307–331. <https://doi.org/10.1177/1525822X04266540>
31. David F. Huynh, Robert C. Miller, and David R. Karger. 2006. Enabling Web Browsers to Augment Web Sites' Filtering and Sorting Functionalities. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology (UIST '06)*, 125–134. <https://doi.org/10.1145/1166253.1166274>
32. Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. 2004. Advances in Dataflow Programming Languages. *ACM Comput. Surv.* 36, 1: 1–34. <https://doi.org/10.1145/1013208.1013209>
33. Simon Peyton Jones, Alan Blackwell, and Margaret Burnett. 2003. A User-centred Approach to Functions in Excel. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP '03)*, 165–176. <https://doi.org/10.1145/944705.944721>
34. Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*, 3363–3372. <https://doi.org/10.1145/1978942.1979444>
35. Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In (CHI '11), 3363–3372. <https://doi.org/10.1145/1978942.1979444>
36. Caitlin Kelleher and Randy Pausch. 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.* 37, 2: 83–137. <https://doi.org/10.1145/1089733.1089734>
37. Caitlin Kelleher, Randy Pausch, and Sara Kiesler. 2007. Storytelling Alice Motivates Middle School Girls to Learn Computer Programming. In (CHI '07), 1455–1464. <https://doi.org/10.1145/1240624.1240844>
38. D. V. Keyson, M. P. A. J. de Hoogh, A. Freudenthal, and A. P. O. S. Vermeeren. 2000. The Intelligent Thermostat: A Mixed-initiative User Interface. In *CHI '00 Extended Abstracts on Human Factors in Computing Systems (CHI EA '00)*, 59–60. <https://doi.org/10.1145/633292.633329>
39. Andrew J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. 2011. The state of the art in end-user software engineering. *ACM Comput. Surv.* 43, 3: 21:1–21:44. <https://doi.org/10.1145/1922649.1922658>

40. Andrew J. Ko and Brad A. Myers. 2004. Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '04)*, 151–158. <https://doi.org/10.1145/985692.985712>
41. Ron Kohavi. 1995. A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2 (IJCAI'95)*, 1137–1143. Retrieved April 11, 2017 from <http://dl.acm.org/citation.cfm?id=1643031.1643047>
42. Alberto H. F. Laender, Berthier A. Ribeiro-Neto, Altigran S. da Silva, and Juliana S. Teixeira. 2002. A Brief Survey of Web Data Extraction Tools. *SIGMOD Rec.* 31, 2: 84–93. <https://doi.org/10.1145/565117.565137>
43. Tessa Lau. 2001. *Programming by Demonstration: a Machine Learning Approach*.
44. Tessa Lau. 2009. Why PBD systems fail: Lessons learned for usable AI. *AI Magazine* 30.4, 65.
45. Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. 2003. Programming by Demonstration Using Version Space Algebra. *Mach. Learn.* 53, 1–2: 111–156. <https://doi.org/10.1023/A:1025671410623>
46. Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In (PLDI '14), 542–553. <https://doi.org/10.1145/2594291.2594333>
47. Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: automating & sharing how-to knowledge in the enterprise. In (CHI '08), 1719–1728. <https://doi.org/10.1145/1357054.1357323>
48. Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: automating & sharing how-to knowledge in the enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*, 1719–1728. <https://doi.org/10.1145/1357054.1357323>
49. J. C R Licklider. 1960. Man-Computer Symbiosis. *IRE Transactions on Human Factors in Electronics* HFE-1, 1: 4–11. <https://doi.org/10.1109/THFE2.1960.4503259>
50. Henry Lieberman. 2001. *Your Wish is My Command: Programming By Example*. Morgan Kaufmann, San Francisco.
51. Henry Lieberman, Fabio Paternò, Markus Klann, and Volker Wulf. 2006. End-User Development: An Emerging Paradigm. In *End User Development*, Henry Lieberman, Fabio Paternò and Volker Wulf (eds.). Springer Netherlands, 1–8. Retrieved April 16, 2014 from http://link.springer.com/chapter/10.1007/1-4020-5386-X_1
52. James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A. Lau. 2008. End-user programming of mashups with vegemite. 106.
53. Greg Little, Tessa A. Lau, Allen Cypher, James Lin, Eben M. Haber, and Eser Kandogan. 2007. Koala: Capture, Share, Automate, Personalize Business Processes on the Web. In (CHI '07), 943–946. <https://doi.org/10.1145/1240624.1240767>
54. Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. 2015. User Interaction Models for Disambiguation in Programming by Example. In

- Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15)*, 291–301. <https://doi.org/10.1145/2807442.2807459>
55. Richard G. McDaniel and Brad A. Myers. 1999. Getting More out of Programming-by-demonstration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '99)*, 442–449. <https://doi.org/10.1145/302979.303127>
 56. L. A. Miller. 1981. Natural language programming: styles, strategies, and contrasts. *IBM Syst. J.* 20, 2: 184–215. <https://doi.org/10.1147/sj.202.0184>
 57. Lance A. Miller. 1974. Programming by non-programmers. *International Journal of Man-Machine Studies* 6, 2: 237–260. [https://doi.org/10.1016/S0020-7373\(74\)80004-0](https://doi.org/10.1016/S0020-7373(74)80004-0)
 58. Robert C. Miller, Victoria H. Chou, Michael Bernstein, Greg Little, Max Van Kleek, David Karger, and Mc Schraefel. *Inky: A Sloppy Command Line for the Web with Rich Visual Feedback*.
 59. Robert C. Miller, Victoria H. Chou, Michael Bernstein, Greg Little, Max Van Kleek, David Karger, and Mc Schraefel. *Inky: A Sloppy Command Line for the Web with Rich Visual Feedback*.
 60. Robert C Miller, Victoria H Chou, Michael Bernstein, Greg Little, Max Van Kleek, David Karger, and others. 2008. Inky: a sloppy command line for the web with rich visual feedback. In *Proceedings of the 21st annual ACM symposium on User interface software and technology*, 131–140.
 61. S. Münch, J. Kreuziger, M. Kaiser, and R. Dillmann. 1994. Robot Programming by Demonstration (RPD) - Using Machine Learning and User Interaction Methods for the Development of Easy and Comfortable Robot Programming Systems. In *In Proceedings of the 24th International Symposium on Industrial Robots*, 685–693.
 62. Brad Myers, Sun Young Park, Yoko Nakano, Greg Mueller, and Andrew Ko. 2008. How Designers Design and Program Interactive Behaviors. In *Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC '08)*, 177–184. <https://doi.org/10.1109/VLHCC.2008.4639081>
 63. Bonnie A. Nardi. 1993. *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, Cambridge, MA, USA.
 64. Dana S. Nau, Stephen J. J. Smith, and Kutluhan Erol. 1998. Control Strategies in HTN Planning: Theory Versus Practice. In *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence (AAAI '98/IAAI '98)*, 1127–1133. Retrieved April 14, 2014 from <http://dl.acm.org/citation.cfm?id=295240.296264>
 65. Myle Ott, Claire Cardie, and Jeff Hancock. 2012. Estimating the Prevalence of Deception in Online Review Communities. In (WWW '12), 201–210. <https://doi.org/10.1145/2187836.2187864>
 66. Myle Ott, Yejin Choi, Claire Cardie, and Jeffrey T. Hancock. 2011. Finding Deceptive Opinion Spam by Any Stretch of the Imagination. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1 (HLT '11)*, 309–319. Retrieved April 13, 2015 from <http://dl.acm.org/citation.cfm?id=2002472.2002512>
 67. John F. Pane, Brad A. Myers, and Chotirat Ann Ratanamahatana. 2001. Studying the language and structure in non-programmers' solutions to programming

- problems. *Int. J. Hum.-Comput. Stud.* 54, 2: 237–264. <https://doi.org/10.1006/ijhc.2000.0410>
68. Marian Petre and Alan F. Blackwell. 2007. Children As Unwitting End-User Programmers. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC '07)*, 239–242. <https://doi.org/10.1109/VLHCC.2007.13>
 69. Panko Ray. 1995. Finding spreadsheet errors; most spreadsheet models have design flaws that may lead to long-term miscalculations. *Information Week*.
 70. Alexander Repenning and Corrina Perrone. 2000. Programming by Example: Programming by Analogous Examples. *Commun. ACM* 43, 3: 90–97. <https://doi.org/10.1145/330534.330546>
 71. Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11: 60–67. <https://doi.org/10.1145/1592761.1592779>
 72. M. B. Rosson, H. Sinha, and T. Edor. 2010. Design Planning in End-User Web Development: Gender, Feature Exploration and Feelings of Success. In *2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, 141–148. <https://doi.org/10.1109/VLHCC.2010.28>
 73. M.B. Rosson, J. Ballin, and J. Rode. 2005. Who, what, and how: a survey of informal and professional Web developers. 199–206. <https://doi.org/10.1109/VLHCC.2005.73>
 74. Ben Shneiderman. 1984. The Future of Interactive Systems and the Emergence of Direct Manipulation. In *Proc. Of the NYU Symposium on User Interfaces on Human Factors and Interactive Computer Systems*, 1–28. Retrieved October 10, 2014 from <http://dl.acm.org/citation.cfm?id=2092.2093>
 75. Ben Shneiderman. 1997. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
 76. Ben Shneiderman and Pattie Maes. 1997. Direct Manipulation vs. Interface Agents. *interactions* 4, 6: 42–61. <https://doi.org/10.1145/267505.267514>
 77. Ben Shneiderman and Catherine Plaisant. 2006. Strategies for Evaluating Information Visualization Tools: Multi-dimensional In-depth Long-term Case Studies. In *Proceedings of the 2006 AVI Workshop on BEyond Time and Errors: Novel Evaluation Methods for Information Visualization (BELIV '06)*, 1–7. <https://doi.org/10.1145/1168149.1168158>
 78. Michael Toomim, Steven M. Drucker, Mira Dontcheva, Ali Rahimi, Blake Thomson, and James A. Landay. 2009. Attaching UI enhancements to websites with end users. In *(CHI '09)*, 1859–1868. <https://doi.org/10.1145/1518701.1518987>
 79. Rattapoom Tuchinda, Pedro Szekely, and Craig A. Knoblock. 2007. Building data integration queries by demonstration. In *Proceedings of the 12th international conference on Intelligent user interfaces (IUI '07)*, 170–179. <https://doi.org/10.1145/1216295.1216328>

80. Rattapoom Tuchinda, Pedro Szekely, and Craig A. Knoblock. 2007. Building data integration queries by demonstration. In (IUI '07), 170–179. <https://doi.org/10.1145/1216295.1216328>
81. Rattapoom Tuchinda, Pedro Szekely, and Craig A. Knoblock. 2008. Building Mashups by example. In *Proceedings of the 13th international conference on Intelligent user interfaces* (IUI '08), 139–148. <https://doi.org/10.1145/1378773.1378792>
82. Rattapoom Tuchinda, Pedro Szekely, and Craig A. Knoblock. 2008. Building Mashups by example. In (IUI '08), 139–148. <https://doi.org/10.1145/1378773.1378792>
83. US Bureau of Labor Statistics. 2017. United States Labor Force Statistics - Seasonally Adjusted. *Labor Market Information. Rhode Island Department of Labor and Training*. Retrieved March 8, 2017 from <http://www.dlt.ri.gov/lmi/laus/us/usadj.htm>
84. Jacob O. Wobbrock, Leah Findlater, Darren Gergle, and James J. Higgins. 2011. The Aligned Rank Transform for Nonparametric Factorial Analyses Using Only Anova Procedures. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '11), 143–146. <https://doi.org/10.1145/1978942.1978963>
85. Jeffrey Wong and Jason Hong. 2008. What Do We “Mashup” when We Make Mashups? In *Proceedings of the 4th International Workshop on End-user Software Engineering* (WEUSE '08), 35–39. <https://doi.org/10.1145/1370847.1370855>
86. Jeffrey Wong and Jason I. Hong. 2007. Making mashups with marmite: towards end-user programming for the web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '07), 1435–1444. <https://doi.org/10.1145/1240624.1240842>
87. Jeffrey Wong and Jason I. Hong. 2007. Making mashups with marmite: towards end-user programming for the web. In (CHI '07), 1435–1444. <https://doi.org/10.1145/1240624.1240842>
88. Kuat Yessenov, Shubham Tulsiani, Aditya Menon, Robert C. Miller, Sumit Gulwani, Butler Lampson, and Adam Kalai. 2013. A Colorful Approach to Text Processing by Example. In (UIST '13), 495–504. <https://doi.org/10.1145/2501988.2502040>
89. Kuat Yessenov, Shubham Tulsiani, Aditya Menon, Robert C. Miller, Sumit Gulwani, Butler Lampson, and Adam Kalai. 2013. A Colorful Approach to Text Processing by Example. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology* (UIST '13), 495–504. <https://doi.org/10.1145/2501988.2502040>
90. Nan Zang and Mary Beth Rosson. 2009. Web-active Users Working with Data. In *CHI '09 Extended Abstracts on Human Factors in Computing Systems* (CHI EA '09), 4687–4692. <https://doi.org/10.1145/1520340.1520721>
91. Nan Zang, Mary Beth Rosson, and Vincent Nasser. 2008. Mashups: who? what? why? In (CHI EA '08), 3171–3176. <https://doi.org/10.1145/1358628.1358826>
92. Nan Zang and M.B. Rosson. 2008. What's in a mashup? And why? Studying the perceptions of web-active end users. In *IEEE Symposium on Visual Languages and*

- Human-Centric Computing*, 2008. *VL/HCC* 2008, 31–38.
<https://doi.org/10.1109/VLHCC.2008.4639055>
93. Nan Zang and M.B. Rosson. 2009. Playing with information: How end users think about and integrate dynamic data. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2009. *VL/HCC* 2009, 85–92.
<https://doi.org/10.1109/VLHCC.2009.5295293>
 94. Haotian Zhou and Ayelet Fishbach. 2016. The pitfall of experimenting on the web: How unattended selective attrition leads to surprising (yet false) research conclusions. *Journal of Personality and Social Psychology* 111, 4: 493–504.
<https://doi.org/10.1037/pspa0000056>
 95. John Zimmerman, Kathryn Rivard, Ian Hargraves, Anthony Tomasic, and Ken Mohnkern. 2009. User-created forms as an effective method of human-agent communication. In (CHI '09), 1869–1878.
<https://doi.org/10.1145/1518701.1518988>
 96. John Zimmerman, Anthony Tomasic, Isaac Simmons, Ian Hargraves, Ken Mohnkern, Jason Cornwell, and Robert Martin McGuire. 2007. Vio: A Mixed-initiative Approach to Learning and Automating Procedural Update Tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '07), 1445–1454. <https://doi.org/10.1145/1240624.1240843>
 97. 2013. Yahoo! Pipes. Retrieved September 12, 2013 from <http://pipes.yahoo.com/pipes/>
 98. 2013. Greasemonkey. Retrieved September 12, 2013 from <https://addons.mozilla.org/en-US/firefox/addon/greasemonkey/>
 99. 2014. Quartz Composer. *Wikipedia, the free encyclopedia*. Retrieved October 27, 2014 from http://en.wikipedia.org/w/index.php?title=Quartz_Composer&oldid=598066763
 100. Greasemonkey. Retrieved September 12, 2013 from <https://addons.mozilla.org/en-US/firefox/addon/greasemonkey/>