Making End User Development More Natural

Brad A. Myers, Amy J. Ko, Chris Scaffidi, Stephen Oney, YoungSeok Yoon, Kerry Chang, Mary Beth Kery, and Toby Jia-Jun Li

Abstract When end users approach a development task, they bring with them a set of techniques, expressions, and knowledge, which can be leveraged in order to make the process easier. The *Natural Programming Project* has been working for over twenty years to better understand how end users think about their tasks, and to develop new ways for users to express those tasks that will be more "natural," by which we mean closer to the way they think. Our chapter in the previous book covered the first 10 years of this research; and here we summarize the most recent 10 years. This includes studies on barriers that impede EUD, and a new tool that helps with the understanding and debugging barriers by showing developers *why* their program has its current behavior. We also describe a tool that we created to

B.A. Myers (✉) · M.B. Kery · T.J.-J. Li
Carnegie Mellon University, Pittsburgh, PA, United States
e-mail: bam@cs.cmu.edu

M.B. Kery
e-mail: mkery@andrew.cmu.edu

T.J.-J. Li
e-mail: tobyli@cs.cmu.edu

A.J. Ko
University of Washington, Seattle, WA, United States
e-mail: ajko@uw.edu

C. Scaffidi
Oregon State University, Corvallis, OR, United States
e-mail: scaffidc@eecs.oregonstate.edu

S. Oney
University of Michigan, Ann Arbor, MI, United States
e-mail: soney@umich.edu

Y. Yoon
Google, Mountain View, CA, United States
e-mail: youngseokyoon@google.com

K. Chang
IBM, Armonk, NY, United States
e-mail: kerry.chang@ibm.com

F. Paternò, V. Wulf (eds.), *New Perspectives in End-User Development*,
DOI 10.1007/978-3-319-60291-2_1

help EUDs input, process, and transform data in the context of spreadsheets and web pages. Interaction designers are a class of EUDs that may need to program interactive behaviors, so we studied how they naturally express those behaviors, and then built a spreadsheet-like tool to allow them to author new behaviors. Another spreadsheet tool we created helps EUDs access web service data without writing code, and extends the familiar spreadsheet to support analyzing the acquired web-based hierarchical data and programming data-driven GUI applications. Finally, EUDs often need to engage in *exploratory programming*, where the goals and tasks are not well-formed in advance. We describe new tools to help users selectively undo past actions, along with on-going research to help EUDs create more efficient behaviors on smartphones and facilitate variations when performing data analysis.

Keywords Spreadsheets · exploratory programming · data analysis · the Natural Programming Group

1 Introduction

The Natural Programming group at Carnegie Mellon University (CMU) has been working for nearly 20 years on applying methods from Human-Computer Interaction (HCI) in order to make programming easier. We have applied this research to professional developers (Myers, Ko, LaToza, & Yoon, 2016), to learners who are trying to become professional developers, and to end-user developers (EUDs). Our key strategy is to study the target developers to understand what their current problems are, and then try to design new languages and tools that will address those problems. We try to make programming be a more "natural" process for the developers, by which we mean *closer to the way the developers think about their tasks*. The goal is to reduce the size of the gulfs of execution and evaluation as articulated by Don Norman (1988) – to make it easier for developers to implement what they have in mind and to understand the state of their program. This is also motivated by the *cognitive dimension* of "Closeness of Mapping," which says: "The closer the programming world is to the problem world, the easier the problem-solving ought to be" (Green & Petre, 1996). The Natural Programming methodology helps us understand how to bring those worlds closer together.

In the early days of the Natural Programming project, as reported in our chapter for the previous version of the *End User Development* book, we studied how non-programmers think about programming tasks, and used that knowledge to develop a more usable programming language for children (Pane & Myers, 2006). This chapter summarizes our work since then that has been focused on EUDs:

- We studied learners and identified debugging as a key stumbling block, which has been surprisingly ignored in many previous tools for EUDs (Ko, Myers, & Aung, 2004). We developed a new tool called the "Whyline" which helps EUDs answer a key question – *why* did or didn't something happen with a program (Ko & Myers, 2004).
- Most of the data on which EUDs' code operates are richly structured, yet mostly must be operated on as strings. The "Topes" system allows EUDs to express the constraints and structure of their data (Scaffidi, Myers, & Shaw, 2008).

- One class of EUDs that our group has addressed are interaction designers, who often must now program in HTML/CSS/JavaScript in order to achieve their desired behaviors. We first studied how interaction designers think about their tasks (Myers, Park, Nakano, Mueller, & Ko, 2008; Ozenc, Kim, Zimmerman, Oney, & Myers, 2010), and then designed a new tool, called **InterState** that tries to enable a more natural way for interaction designers to express those behaviors (Oney, Myers, & Brandt, 2014).
- Spreadsheets remain a key tool for EUDs to do data analyses, but much modern data now comes from web services in hierarchical XML or JSON formats. We developed the "Gneiss" tool to enable EUDs to create their own data analysis and web applications using hierarchical data from web services using familiar spreadsheet languages and interaction techniques (Chang & Myers, 2014a, 2014b, 2016).
- Much programming by EUDs and professionals is *exploratory*, in that the developer does not necessarily know the correct code to write before starting, and therefore must try out different code, often by *backtracking* or reverting old code (Yoon & Myers, 2014). However, there is surprisingly little support for this exploration in programming environments. We are studying this problem as part of a large, multi-institution project called "Variations to Support Exploratory Programming" (http://www.exploratoryprogramming.org/). One approach is to facilitate *undoing* of the unwanted edits. The "Azurite" tool supports selective undo, to allow developers to go back and undo edits while retaining desired edits that happened afterwards (Yoon, Koo, & Myers, 2013; Yoon & Myers, 2015). One application of this is *regional undo*, where all the edits for a selected section of code can be undone without affecting any other code.
- A current project is looking at better support for data scientists in their exploratory programming. Many data scientists are EUDs, using languages such as Excel, R, or Python, and often need to try out different algorithms, libraries and parameter values, for which there is little support. The "Variolite" tool provides many features, including light-weight variants, to support EUD explorations (Kery, Horvath, & Myers, 2017).
- Finally, another new project, called "Sugilite," supports EUD on mobile phones, especially to help with complex and repetitive multi-app tasks. This multimodal system can learn how to perform arbitrary tasks using third-party Android apps from the user's demonstration, and generalizes the automation by finding parameters and their possible alternative values from the users' verbal commands and the third-party apps' UI structures (Li, Azaria, & Myers, 2017).

The following sections discuss these projects in more detail.

2 Whyline

One of the most difficult tasks in end user development is *debugging*, or trying to find the code in a program that is causing an unwanted behavior. In our lab, we wanted to discover novel ways of making debugging easier, faster, and more successful. To begin, we asked *how do EUDs think about debugging?*

To find out, we observed many EUDs trying to fix bugs, and discovered many slow, unproductive strategies (Ko & Myers, 2005; Ko, Myers, Coblenz, & Aung, 2006). Less experienced EUDs would just read their code and change things they thought *might* be wrong. This often introduced *new* defects, rather than resolving the original ones. More experienced EUDs used breakpoint debuggers to step through a program's execution, looking for where it deviated from the expected behavior. For non-trivial programs, this involved inspecting thousands of lines of code, which required so much vigilance that many EUDs skipped right over the bug. The most experienced EUDs *guessed* what the defect might be and set breakpoints to see if their guess was right. If it was, or if it was close, this was effective – unfortunately, the space of possible defects was often so large, most guesses were wrong, and these EUDs had to spend minutes, if not hours discovering that their hypothesis was incorrect. When we compared these strategies to those of novices and professionals, we found that even experienced professional developers guessed wrong the first time (but were faster at investigating their hypotheses).

In all of these observations, we noticed one recurring trend: every search for a defect began with a question about *program output* such as "Why didn't that animation start?," "Why did this error dialog appear?," or "Why is this button disabled?" We realized that EUDs were starting their search with something they were certain about – the faulty output – and trying to retrieve information about its causes. This led to a compelling idea: what if EUDs could ask these "why" and "why not" questions directly and a tool could simply answer them by showing the *causes* of the faulty output?

Our breakthrough insight was that programs *specify* the output they produce in the form of API calls: programs have print statements, they call graphics rendering libraries, they call audio libraries, and so on. What our tool had to do was identify these output statements and then present a user interface for EUDs to select which output they wanted to ask "why" and "why not" about.

We built our prototype for the Alice programming environment, which enables EUDs to create interactive 3D virtual worlds (Ko & Myers, 2004). As Fig. 1 shows, our interface, which is called the Whyline, lets EUDs pause the program, click on a "why" menu that contained all of the possible program's output, and then select a question. To answer "why" questions, the Whyline keeps a detailed execution history that stores the *data* and *control* dependencies of every instruction executed in the program, allowing the Whyline to identify every upstream cause of a selected program output, and display those causes to help EUDs find the source of the unwanted output. To answer "why not" questions, the Whyline analyzes the static control dependencies that prevented the desired output statement from executing, showing all of the conditions that were not satisfied that would have enabled the output to execute. Fig. 1 shows an example of a "why not" explanation.

Did the Whyline actually help? Over the course of several studies (Ko & Myers, 2004, 2009), the answer was clearly yes, showing that EUDs using the Whyline could localize defects anywhere from 2 to 8 times faster than EUDs using conventional breakpoint debuggers. Our results showed these increases in

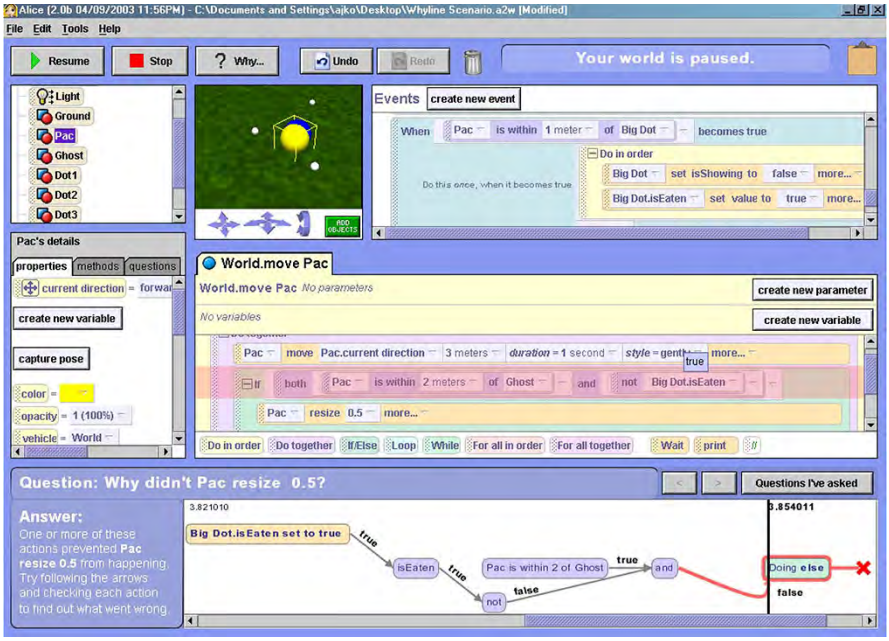

Fig. 1 The Whyline for Alice, showing an EUD asking why a Pac Man character did not resize, as expected. The Whyline explains that that it would have resized, but the condition that guarded the behavior was false

debugging speed were due to a change in the structure of an EUD's debugging task: rather than having to iteratively guess the defect and check if they were correct, EUDs could work backwards from something they were *certain* was wrong directly to the source code that caused it. The Whyline prevented speculation, and instead encouraged EUDs to focus on facts. Further, we found that the Whyline's benefits generalize beyond EUDs and can also be effective in helping more experienced developers debug Java programs (Ko & Myers, 2010). Thus, our initial observations of EUDs' bug fixing strategies informed the design of a technique that is useful for both EUDs and for traditional developers.

3 Topes

Much of the data that EUDs deal with must be represented in programming systems as strings, including names, job titles, part numbers, ID numbers, locations, etc. In fact, according to one study, 40% of spreadsheet cells contained non-numeric, non-formula textual data (Fisher & Rothermel, 2004). Software applications offer poor support for operating on these data, so EUDs must write their own code for working with them. Parsing, categorizing, validating, and

reformatting these data can be difficult for several reasons. First, each category can be multi-format in that each of its instances can be written several different ways. Second, many useful categories are probabilistic rather than binary – each category can include questionable values that are unusual yet still valid. During user tasks, such unusual strings often are worthy of double-checking, as they are neither obviously valid nor obviously invalid. Third, each category is application-agnostic in that its rules for validating and reformatting strings are not specific to one software application – rather, its rules are agreed upon implicitly or explicitly by members of an organization or society. For example, a web form might have a field for entering Carnegie Mellon office phone numbers like "8-5150" or "412-268-5150." EUD tools offer no convenient way to create code for putting strings into a consistent format, nor do they help users create code to detect inputs that are unusual but possibly valid, such as "7-5150" (since CMU office phone numbers rarely start with "7"). The result is that end-users must often manually clean up their data, or leave the data unchecked.

In order to help users with their tasks, we created a new kind of abstraction called a "tope" and a supporting development environment (Scaffidi et al., 2008). Each tope describes how to validate and reformat instances of a data category. Topes are sufficiently expressive for creating useful, accurate rules for validating and reformatting a wide range of data categories commonly encountered by EUDs. By creating and applying topes, EUDs can validate and reformat strings more quickly and effectively than they can with other techniques. Tope implementations are reusable across applications and by different people, highlighting the leverage provided by EUD research aimed at developing new kinds of application-agnostic abstractions. The topes model demonstrates that such abstractions can be successful if they model a shallow level of semantics, thereby retaining usability without sacrificing usefulness for supporting users' real-world goals.

The Topes system includes tools that allow EUDs to define their own categories, including checking whether a string is of the desired format, and ways to convert strings into various valid formats. For example, in Fig. 2, the user is defining two variations of a "person name," which share the same parts. The system will use this definition to generate code for use in spreadsheets and web pages for validating and transforming strings representing person names. For constraints that are "almost always" true, the system will generate warnings instead of errors for violations. In a small user study, EUDs were able to create such definitions and the system proved highly effective at helping EUDs to create abstractions for validating strings.

4 InterState

Creating a good user interface requires more than carefully arranging the graphical elements that define its appearance. It also requires defining the interface's *behavior* – how it reacts to user input and other events. Although sketches and drawing software

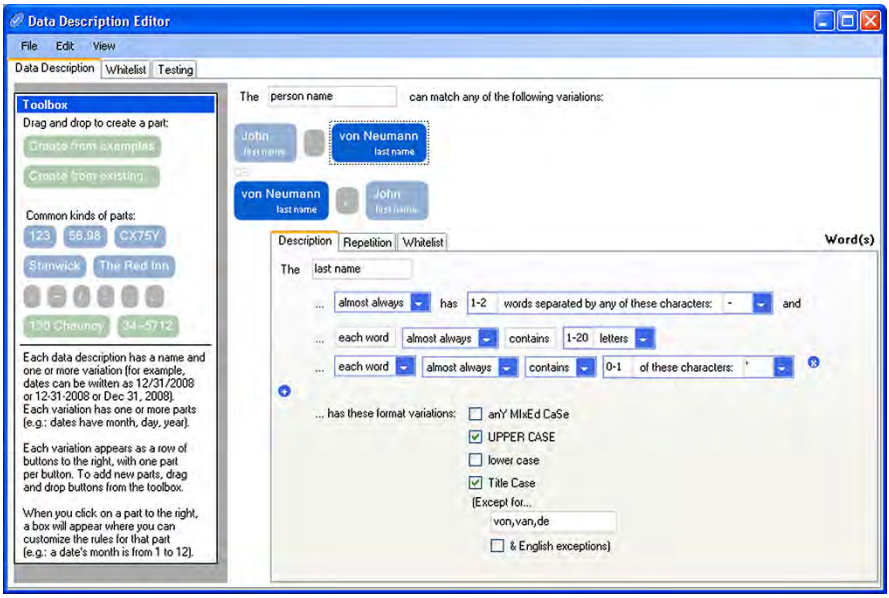

Fig. 2 Dragging and dropping a prototype's icon from the Toolbox creates a new part, and the editor also supports drag/drop rearrangement of parts as well as copy/paste. Users can click the example in a part's icon to edit it, while clicking other parts of the icon displays widgets for editing its constraints, which are shared by every instance of the part. Clicking the "+" icon adds a constraint while clicking the "x" icon deletes the constraint

make it relatively straightforward to define an interface's appearance, correctly implementing its behavior requires programming skill. The event-callback model, which most user interface frameworks rely on to define interface behaviors, has several drawbacks that make it inappropriate for EUDs (Meyerovich et al., 2009; Myers, 1991; Oney, Myers, & Brandt, 2012). We explore how to enable interaction designers who are EUDs to program behaviors themselves by extending the spreadsheet model of programming.

In order to explore a more usable way for EUDs to define interactive behaviors, we started with studies on how non-programmers naturally describe interactive behaviors (Park, Myers, & Ko, 2008). We also conducted workshops to better understand communication barriers between interaction designers and developers (Ozenc et al., 2010). Based on the results of those studies, we iteratively designed a new framework for letting interaction designers define GUI behaviors, called the *state-constraint framework*. This framework combines *constraints* – which allow developers to define relationships among elements that are maintained by the system – and *state machines* – which track the status of an interface. In the state-constraint framework, developers write interactive behaviors by defining constraints that are enforced when the interface is in specific states (Oney et al., 2012, 2014). We implemented the state-constraint framework

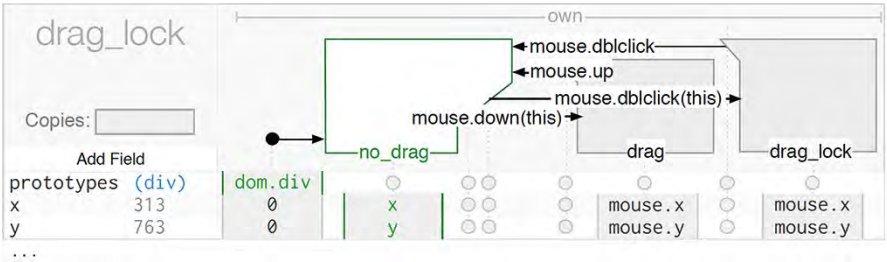

Fig. 3 An illustration of a basic InterState object, named drag_lock. Properties, which control drag_lock's display, are represented as rows (e.g. x, and y). States and transitions are represented as columns (e.g. no_drag, drag, and drag_lock). An entry in a property's row for a particular state specifies a constraint that controls that property's value in that state; while drag_lock is in the drag state, x and y will be constrained to mouse.x and mouse.y respectively, meaning drag_lock will follow the mouse while dragging. Note that in this example, when the user performs a double click to initiate drag lock, the drag_lock object does enter and then leave the drag state intermittently as a result of the mouse.down and mouse.up events that are fired during a double click

in InterState (Oney et al., 2014), an interactive spreadsheet-like graphical environment for EUDs.

InterState reduces the number of control structures that new developers need to learn in order to write UI behaviors. Developers can express UI behaviors using simple expressions – which are like spreadsheet equations – that define constraints and transitions among states. InterState's visual notation concisely represents interactive behaviors as a table whose rows are properties and whose columns are states. This visual notation allows developers to see which events affect a property by scanning the property's row and which properties an event affects by looking at that event's column, as Fig. 3 illustrates.

InterState also includes a *live editor* that helps reduce the "gulf of evaluation" in determining the effects of a change, which has been shown to be a significant barrier for both EUDs and experienced developers. In InterState, edits are immediately reflected in the running application and changes in runtime state and property values are highlighted in the editor, which enables quick experimentation and parameter tuning. The live editor also allows the developer to always have a running application by "localizing" errors. This means that only the parts of the program that depend on problematic expressions are not executed, which avoids confronting EUDs with dozens of syntax and runtime errors.

A comparative laboratory study indicated that InterState can be effective in helping users who do not have prior UI programming experience understand and modify code. Even developers with JavaScript experience were significantly faster at understanding and modifying UI code in InterState compared to using JavaScript (Oney et al., 2014). Further, in order to test InterState's scalability, we implemented several complex user interfaces, finding that by many metrics (such as number of control structures and amount of space), InterState's implementation is more concise than the alternative JavaScript implementation.

5 Gneiss

Today, more and more data are moving to the cloud, and many companies provide *web services* that let people access web data programmatically. Web services allow developers to make custom use of various kinds of online data. Many web services also provide computational services that can analyze or transform the user's data. While web services are powerful tools that make the data and computing ability of the cloud available to people, using these web services currently requires significant programming expertise and effort.

The creation of Gneiss[1] was motivated by prior literature that shows that even professional developers found it difficult to use web services and often required learning new language features or libraries to complete their tasks (Zang, Rosson, & Nasser, 2008). Prior literature also showed that some EUDs want efficient ways to do custom data analysis that use multiple online data sources (Lin, Wong, Nichols, Cypher, & Lau, 2009). In Gneiss, we explored extending the spreadsheet model to support using web service data, since spreadsheet programming is popular among users of all programming levels from EUDs to professional developers and data analysts.

Gneiss makes contributions in extending spreadsheets to support new programming tasks that help EUDs work with online data. First, Gneiss introduces new UIs and interaction techniques to the familiar spreadsheet environment to enable users to send data to and retrieve data from web services without writing conventional code. Gneiss has a left pane (Fig. 4 at 1) that lets users load JSON data from REST web services. The user can send data from arbitrary spreadsheet cells by replacing any part of the web API with spreadsheet cell name (see Fig. 4 for an example), and extract data from the returned document in the left pane to the spreadsheet by selecting a field and dragging it to a spreadsheet column. Using the returned document's structure and the user's selection as an example, Gneiss will extract other similar fields in the document for the users, eliminating the need to write queries in languages such as XPath to select the desired data. Leveraging the spreadsheet's live programming, changes in spreadsheet cells used in a web service call will trigger Gneiss to immediately send a new API request using the cell's new value and in turn update the spreadsheet data. This makes a spreadsheet into an interactive platform for querying cloud data (Chang & Myers, 2014b).

Gneiss further enables spreadsheet users to create interactive web applications that can use and modify spreadsheet data (Chang & Myers, 2014a). Gneiss's right pane (Fig. 4 at 3) is a web interface builder where the user can create web pages by dragging-and-dropping GUI elements from the right bar and editing the properties of an element in a table (Fig. 4 at 4 and 5). In Gneiss, each GUI element property in the web application is treated as a spreadsheet cell, so it can reference other

[1]Gneiss is a type of rock, pronounced like "nice." Here it stands for **G**athering **N**ovel **E**nd-user **I**nternet **S**ervices using **S**preadsheets.

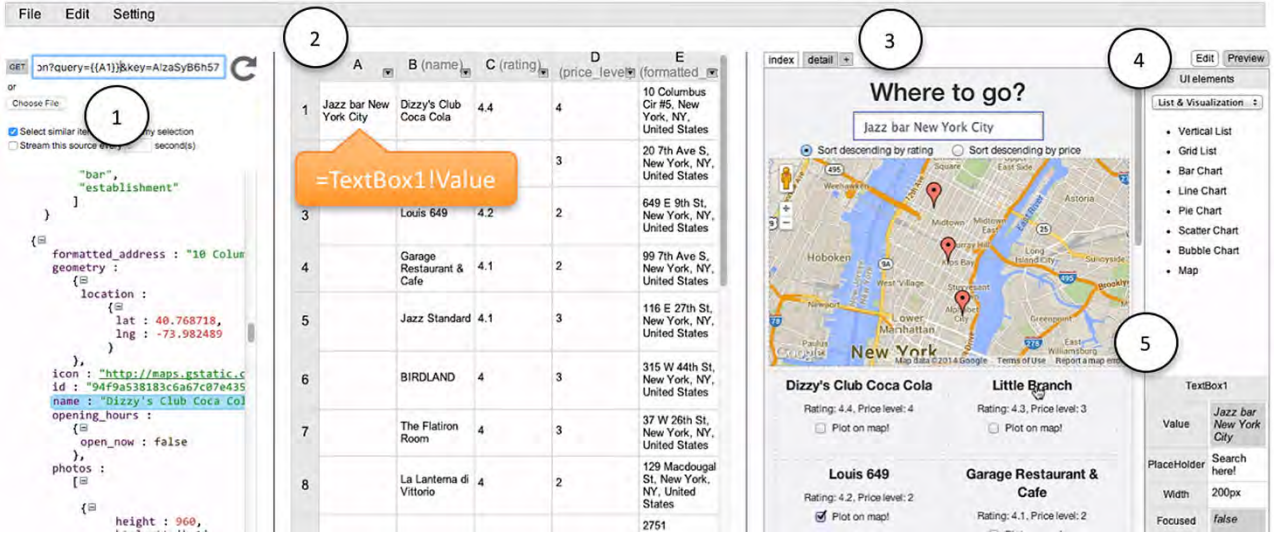

Fig. 4 Gneiss's user interface. (1) is the source pane where the user can load a web API in the URL box and view the returned JSON document. The user can send spreadsheet data to a web service by replacing parts of the web API with spreadsheet cell names. Here the user sends spreadsheet cell A1 to the web service as the value of the query parameter using the syntax {{A1}}. The user can extract fields from the returned JSON document to the center spreadsheet (2) by drag-and-drop. Here, spreadsheet column B-E hold four different fields extracted from the returned document in (1). (3) is the web interface builder where the user can create a web application by dragging-and-dropping GUI elements from the right toolbar (4) to the output page. The user can view and edit a selected GUI element's properties in (5). Here, (5) shows the properties of TextBox1 which is the search box in the output page. The textbox's Value property changes dynamically based on what the user enters in it (currently "Jazz bar New York City"). In the spreadsheet, cell A1 is set to be the value of the search box using the formula =TextBox1!Value, which is then sent to the web service in (1) as the query term to retrieve new data. GUI element properties in (3) can also use spreadsheet data as their values. For example, here the bold text in the grid list is set to show the data in spreadsheet column B using a spreadsheet formula

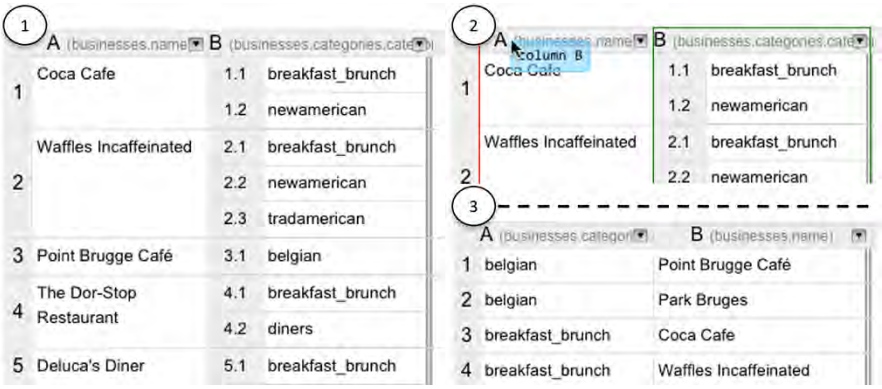

Fig. 5 Gneiss visualizes hierarchies in data using nested spreadsheet cells, and lets users restructure the data by any field by drag-and-dropping a column to a different location. Here, (1) shows a structured document of restaurant data grouped by restaurant names. The user can restructure this document to instead view the data by restaurant categories by dragging the categories column to the front of the names column (2 and 3)

spreadsheet cells and use functions to compute its value, and also be referenced in spreadsheet cells and functions to compute new values. This allows the use of spreadsheet languages to construct two-way data flow between a web page and a spreadsheet whose data can be local or from web services. Gneiss further introduces *interactive properties* in web GUI elements whose values change live based on how the user interacts with the elements. This enables the user to program many kinds of interactive behaviors in a web application, such as to search, sort, filter and visualize data using GUI controls, using spreadsheet languages without needing to write conventional event handler code (Chang & Myers, 2014a).

Finally, since most modern web services return hierarchical data such as JSON and XML data, we also extend spreadsheets to support hierarchical data, with control over how they are shown and manipulated in the spreadsheet (Chang & Myers, 2016). Gneiss introduces a new method to visualize hierarchical data as a spreadsheet using the relative hierarchical relationships among data in adjacent columns. Under this new visualization method, reshaping, regrouping, and joining hierarchical objects in a spreadsheet can be done using simple interaction techniques (see Fig. 5). This model also extends spreadsheet languages, sorting and filtering to support selecting and manipulating data by its hierarchies, allowing the user to calculate summaries of data using spreadsheet formulas without the need of pivot tables. In our user study, Gneiss helped spreadsheet users who were EUDs complete data exploration tasks that involve restructuring and joining two hierarchical JSON documents almost two times faster than Excel, and they even outperformed experienced programmers writing JavaScript or Python code doing the same tasks (Chang & Myers, 2016).

6 Azurite

Since developers are human, they often make mistakes while writing code. In other cases, developers intentionally make temporary changes to the code, either as an experiment or to help with debugging. As a consequence, developers often need to *backtrack* while coding, meaning that they revert their code back to an earlier state at least partially. For example, developers try out different values for various parameters. When developers try to learn an unfamiliar API, they might try writing some code and running it to see if the code works as expected, and if it does not, they backtrack and try something else. Backtracking support is much needed in *exploratory programming* (Sheil, 1983), where the correct solution to the given problem is not well known or when there are multiple potential solutions with their own strengths and weaknesses. Moreover, recent studies show that EUDs need easy access to past versions of their code as reference when rewriting parts of their code (Henley & Fleming, 2016; Kuttal, Sarma, & Rothermel, 2011).

However, we noticed that modern development tools for EUDs and professional programmers alike do not provide enough support for backtracking. The linear undo model used in development tools is not suitable for all situations. Notably, users can only undo the most recent edits, which can be very inconvenient when they realize their mistake after making some other changes that they want to keep. Another option is to use version control systems such as Subversion or Git, but backtracking is supported in these tools only if the desired code is already committed to the system, and version control is rarely used by EUDs (Grigoreanu, Fernandez, Inkpen, & Robertson, 2009).

To provide better backtracking support for developers, we first asked, *when and how do developers backtrack?* To answer this, we observed developers completing simple programming tasks in our lab, interviewed and surveyed developers about their backtracking experience, and finally collected and analyzed developers' coding logs while they are working on their own projects. Developers felt that backtracking happens quite frequently, and they had problems while backtracking, such as failing to locate the right code to be backtracked (Yoon & Myers, 2012). Our log analysis detected about 10 backtracking instances per hour, and for 34% of those backtracking situations, developers performed them manually by deleting or retyping code, confirming that there are backtracking situations not very well supported by existing tools (Yoon & Myers, 2014).

So how could we support backtracking better? One insight we had was that a *selective undo* in editors could help solve these backtracking problems. Users could use selective undo to revert only specific edits from the past, without affecting the following, more recent edits. Inspired by prior research in selective undo in the area of drawing editors (Berlage, 1994; Myers, 1998), we developed our tool Azurite[2], which is a selective undo tool that works in the Eclipse code editor.

[2]Azurite is a blue mineral, and here stands for **A**dding **Z**est to **U**ndoing and **R**estoring **I**mproves **T**extual **E**xploration.

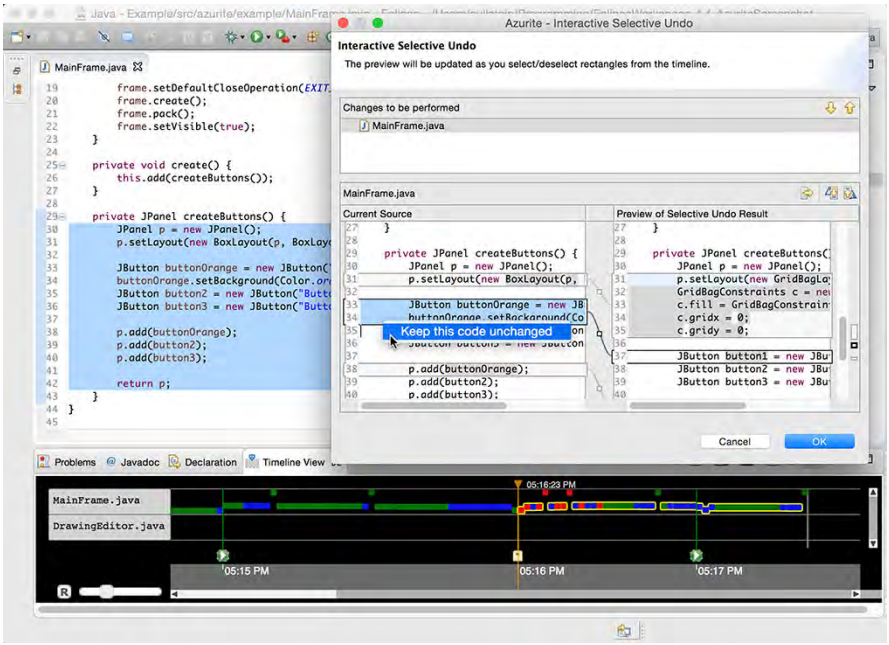

Fig. 6 An example screenshot of Azurite running in the Eclipse IDE. At the bottom, a timeline visualization of recent code changes is provided. The user is currently using the "Interactive Selective Undo" dialog, which is one of the more sophisticated selective undo features of Azurite

Although Eclipse is a tool mostly used by professional developers, the ideas explored in Azurite are relevant to EUD environments as well, since EUDs perform exploratory programming (Henley & Fleming, 2016; Kuttal et al., 2011), and must enter and edit their code.

Fig. 6 shows an example screenshot of the Azurite tool. The *timeline visualization*, shown at the bottom of Fig. 6, is the most basic user interface of Azurite, where the users can see the code change history by scrolling through it, select some past edits, and use the selective undo command (Yoon et al., 2013). All the fine-grained code changes are automatically tracked by the Azurite system, without users needing to manually commit their code. However, an important question still remains: how can users effectively and accurately find and select the desired edits which are to be undone?

The observations from our lab study showed that developers remember certain aspects about the code edits that they want to undo. Our goal was to provide a more natural way for users to express what they remember about the code changes. To this end, Azurite supports a rich set of user interfaces for selective undo besides the timeline visualization. One of the most popular form of selective undo Azurite provides is *regional undo*, where users can select some region of code in the editor and use a keyboard shortcut to perform

selective undo directly on only that region. This feature was driven by our observation that users often remember the *location* of the code changes they want to undo (Yoon & Myers, 2015). With these selective undo features implemented, we evaluated whether Azurite actually helps developers perform backtracking better. Through a controlled lab study with 12 developers, we confirmed that the users could quickly learn and use the features during the study, and the Azurite users completed the given backtracking tasks twice as fast compared to when not using Azurite (Yoon & Myers, 2015).

7 Variolite

In a current project, we are looking at exploratory programming in the context of *data scientists*. The term "data scientist" is open-ended (and often disputed who exactly it includes), but here we use it simply to encompass a broad range of people who write programs to work with data. Analyzing data and using techniques such as machine learning is increasingly important to many professions, including, for example, engineering, medicine, marketing, and research. Individuals in these diverse fields are very often EUDs.

Current tools for data science include GUI-based tools like SPSS or WEKA for relatively straightforward analyses. For more complex data manipulation, individuals often turn to programming, using languages such as Excel, MatLab, R, or Python. While much recent research has gone into making GUI-based tools (mostly for machine learning) more accessible to EUDs (Amershi, Cakmak, Knox, & Kulesza, 2014; Yang et al., 2013), the act of coding in this context is less studied. A few recent studies of professional programmers (Hill, Bellamy, Erickson, & Burnett, 2016) and machine learning experts (Patel, 2013) working in data science tasks have pointed to real struggles that experts faces with exploratory programming.

When "what code should I implement?" or "what is the precise goal of my code?" are questions that cannot be answered at the start of a project, *exploratory programming* is a way of understanding the problem better through a trial-and-error approach with code. In a concrete sense, this means changing code, parameters, and data to test out new ideas until something works. With data science in particular, this process is unfortunately not always straightforward code development. *Non-linear* iteration (Patel, 2013) where an attempt that failed in the past may be fruitful in the future is quite common. For this reason, data scientists often try, and struggle, to keep track of their experiments (Hill et al., 2016). Experimentation can cause code to be more and more unstable, as new chunks of code are added, tested, discarded, and run on different sets of input files. As developers try to answer complex questions with their data, ideas tried so far can be difficult to keep track of and confusion and logic errors are very real threats (Hill et al., 2016; Patel, 2013). Understanding these coding practices and developing new kinds of supports for exploration are crucial to making this kind of work more accessible to EUDs.

To investigate this problem more closely, we approached 10 data scientists and interviewed them about their recent projects (Kery et al., 2017). We grounded these discussions by viewing and discussing artifacts that went with their projects, such as their code, data, file folders, and notes. We followed this with a survey of an additional 60 data scientists.

What does it mean to develop exploratory code? We found that developers currently use ad-hoc strategies for keeping track of their experiments' code and data. For example, developers in our study often rely on commenting and copying code, as well as keeping around old code to facilitate "versioning" of different ideas within the same file. For instance, in order to try different variations of an algorithm, one participant used copy-and-paste to create functions such as "analyis1," "analysis2" and alternated which one was run. Another participant used code comments to alternate their code's execution, sometimes in complex sets of code switched "on" or "off" using a code comment symbol. Through the survey, we found that informal versioning techniques such as these are widely used. Furthermore, we found that even among data scientist who actively use version control software (VCS) systems such as Git or SVN for *other* kinds of work, they predominantly chose to rely on informal techniques, rather than a VCS, for exploratory code.

Finding ways to support data scientists' needs with versioning and experiment-tracking may help make their explorations more robust. Informal versioning that data scientists currently rely on allows them to perform interactions which typical VCSs currently do not support. For example, a data scientist using simple copy/paste and text commands can create versions of any size chunk of code, whereas standard VCS only support versions at the file level. Furthermore, with informal techniques, there is a far lower learning curve for EUDs who do not know VCS, since they can simply leverage their text editing skills to explore variants, rather than learning a new tool.

We created Variolite,[3] an extension to the Atom editor, to investigate new kinds of support for data science versioning (Kery et al., 2017). In Variolite (Fig. 7), a developer can select any size piece of code and issue the command "wrap in variant." This wraps the code chunk in a box, which can be tabbed, similar to a web browser, to keep different local versions of that code on different tabs. We used participatory design for Variolite by showing initial sketches of potential design ideas to data scientists and getting their feedback. In a preliminary usability test of an implemented version of Variolite with 10 participants, who were a mix of novice and advanced developers, the majority found this interaction usable and desirable. We are continuing work on Variolite, and are investigating new ways to support data scientists in their exploratory code.

[3]Variolite, which is a kind of rock structure, here stands for **V**ariations **A**ugment **R**eal **I**terative **O**utcomes **L**etting **I**nformation **T**ranscend **E**xploration.

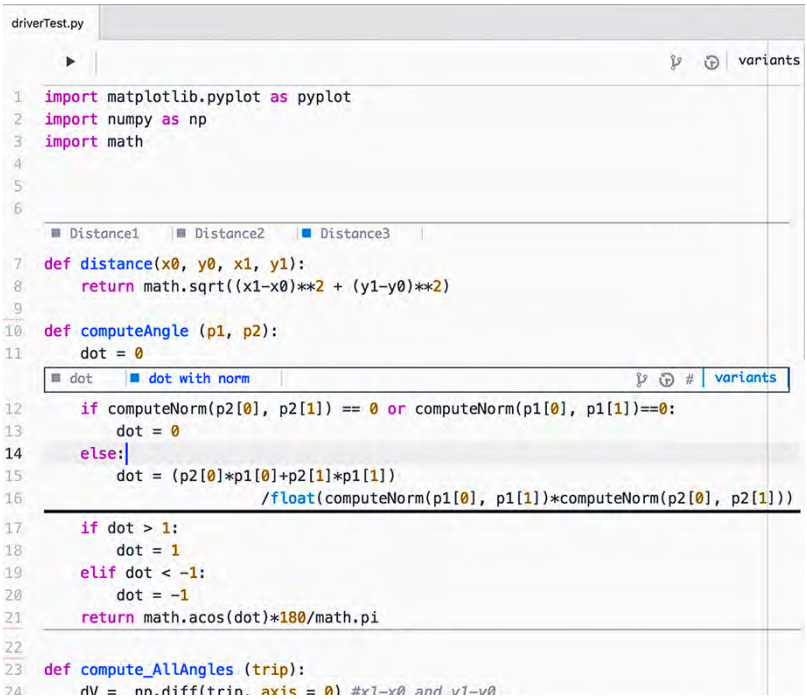

Fig. 7 A screenshot of Variolite. Here are two variant boxes. An outer box wraps the distance and computeAngle functions, and has three versions "Distance1," "Distance2," "Distance3" that the user can flip among with tabs. The inner variant box has two versions "dot," and "dot with norm." Versioning that is visible within the text editor is meant to be more accessible to novices and EUDs

8 Sugilite

In recent years, mobile phones have evolved from being solely communication devices into ubiquitous tools that support a wide range of computing tasks, including information seeking, game playing, entertainment, and navigating. Mobile devices have exceeded PCs in internet usage (O'Toole, 2014) and have become the main computing device for many users (Smith, 2015). Thus, it is increasingly important to study how end-user development can be applied to enable end-users to create automations to help perform personalized computing tasks on mobile devices. In this section, we report on our ongoing project to create a new EUD tool named Sugilite[4] (Li et al., 2017) to enable EUDs to automate mobile tasks using a Programming by Demonstration (PbD) approach

[4]Sugilite is named after a purple gemstone, and here stands for **S**martphone **U**sers **G**enerating **I**ntelligent **L**ikable **I**nterfaces **T**hrough **E**xamples.

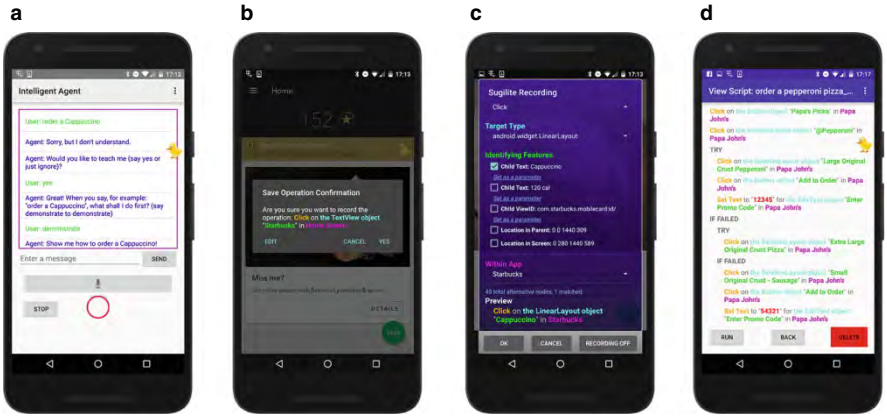

Fig. 8 Screenshots of SUGILITE: (a) the conversational interface; (b) the recording confirmation popup; (c) the recording disambiguation/operation editing panel; and (d) the viewing/editing script window

(Cypher et al., 1993; Myers, McDaniel, & Wolber, 2000) combined with a conversational agent.

Tasks on mobile devices are often performed using mobile apps. Each app usually has limited functionality within a single domain. As a result, complex tasks often require the use of *multiple* apps (Sun, Chen, & Rudnicky, 2016). For example, planning a dinner event may require steps like searching for a restaurant, viewing the transportation options, determining scheduling information, making the actual reservation, and entering information into a calendar, where each step is performed with a different app. However, coordinating multiple apps is particularly challenging on mobile compared to on a computer due to the small screen size and limited support for multi-tasking and cross-app data sharing. For the most common scenarios of cross-app usage, the developers of the apps may implement features to support a few built-in data sharing mechanisms (e.g. the "Share To" button to share data from Photo Gallery to Messenger or Social Media Apps) or the API of services (e.g. Google Maps showing Uber fare estimates in the results). Nevertheless, the "long tail" of personalized mobile computing tasks are mostly not supported directly. This is where EUD can play an important role in enabling the users to create their own automations for repetitive mobile tasks in order to improve their efficiencies in mobile computing.

The SUGILITE system uses the Programming by Demonstration (PbD) approach. It has a multi-modal interface where the user can give a verbal command to execute an automation through a voice conversational interface (Fig. 8a), while making demonstrations (Fig. 8b, c) and editing existing scripts (Fig. 8c, d) using direct manipulation. In the background, SUGILITE detects the apps' user interface hierarchical structures, such as the menu tree, for all the activities that users visit. Then, SUGILITE combines the voice command, the actions recorded, and an analysis of the app's structures to infer generalizations of the script. This allows SUGILITE to learn a

generalized script for the task from a single demonstration. SUGILITE also provides error handling and checking mechanisms that allow the user to demonstrate new steps to enable the script to handle new situations at runtime.

A major advantage of SUGILITE and the PbD approach compared to other Mobile EUD systems is that SUGILITE can automate tasks using *any* third-party Android app (with a few exceptions noted in the paper (Li et al., 2017)). It also enables the users to demonstrate directly in the interfaces of the third-party apps that they are already familiar with, which is particularly useful for EUDs.

In a lab study, 19 participants with various levels of programming experience (including seven non-programmers) were able to use SUGILITE to create automations for four tasks derived from common real-world smartphone usage scenarios with an 85.5% completion rate. No significant difference in either completion rate or completion time was found between participant groups with different levels of programming experience. The result also showed that for our four example tasks, using SUGILITE to automate tasks is more efficient timewise than using direct manipulation if a repetitive task is to be performed for more than 3 to 6 times (Li et al., 2017).

9 Lessons Learned and Implications for the Future

Here we collect some observations on EUD from our over twenty years of research in this area.

- Studying the target group of EUDs to investigate their *natural* ways to describe their tasks and procedures can reveal novel ways that the development system might operate. For example InterState's design was motivated by research into non-programmers' natural language descriptions of interface behaviors. Other design and evaluation methods from the human-computer interaction (HCI) area also have proven useful in improving our systems (Myers et al., 2016).
- Many of the problems that end user developers face are problems that professional developers face as well (Ko & Myers, 2005; Ko et al., 2006). The difference is that EUDs face them at a smaller scale and often with less experience, less effective strategies, and different motivation (Ko et al., 2011). Because of this overlap, our work has shown that it is often possible to make breakthroughs in professional developer tools by first starting with smaller scale EUD tools. For example, the ideas behind Gneiss, the Whyline, and InterState have been shown to benefit both EUDs and professional developers.
- Although debugging is just as important for EUDs as it is for professional developers, many EUD tools do not provide adequate debugging support. Even for the EUD tools that provide some debugging support, they often use the same techniques and metaphors as professional debugging tools. While teaching EUDs appropriate strategies is a great idea (Loksa et al., 2016), tools for EUDs can do a lot more to help with debugging, as shown by the Whyline.

- Although spreadsheets are an old tool, they are still a favorite EUD platform, and can be extended to support EUDs in several areas. We have presented three enhancements of spreadsheets – Topes, InterState and Gneiss, which extend what spreadsheets can process to more expressive strings, stateful formulas, web services and hierarchical data.
- Most EUDs use exploratory programming and write code that they may know they do not intend to keep, or which they plan to edit frequently, but this process is not supported by today's tools. Ideas such as visualizing edit history, selective undo, and light-weight variants have been shown to help.

10 Conclusions and Future Work

The Natural Programming Project has been studying end-user development and creating novel ways for end-users to create and debug their programs for many years, with much exciting research in progress. The "natural programming" approach has proven to be a useful way to understand the target users' real needs and what might be the appropriate ways to solve them. Across this work, we have found that supporting EUDs in all of these settings has required the same basic process: (1) understand what is difficult about a task, and then (2) identify ways of changing that task through new kinds of analyses and data.

For the future, we will continue to strive to produce a "gentle-slope system" where getting started with programming will be easy for EUDs, and there will be no walls that prevent them from learning what is needed to expand the kinds of programs they can build (Myers, Hudson, & Pausch, 2000). While we have made progress, research is still needed across all the topics mentioned above. In addition, the recent rise in computing power has made more powerful machine learning techniques such as deep learning possible, which computer scientists have leveraged to create artificial intelligence capable of complex tasks including driving automobiles, categorizing videos (Clark, 2012), learning games (Muncy, 2016), and doing science (Buchen, 2009). End-user programmers, each with their own unique and diverse needs and context, could potentially benefit from new systems enabling them to create artificial intelligences of their own. As the computing power of machines grows ever closer to that of animals, programming could some become as "natural" as training a dog.

Acknowledgements This article grows out of over 20 years of work by the Natural Programming group by more than 50 students, staff and postdocs in addition to the authors, and we thank them all for their contributions. The work summarized here has been funded at least by SAP, Adobe, IBM, Microsoft, Yahoo! and multiple NSF grants including CNS-1423054, IIS-1314356, IIS-1116724, IIS-0329090, CCF-0811610, IIS-0757511, and CCR-0324770. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of any of the sponsors.

References

Amershi, S., Cakmak, M., Knox, W. B., Kulesza, T. (2014). Power to the people: the role of humans in interactive machine learning. *AI Magazine*, *35*(4), 105–120.

Berlage, T. (1994). A selective undo mechanism for graphical user interfaces based on command objects. *ACM Transactions on Computer Human Interaction. ACM Transactions on Computer Human Interaction*, *1*(3), 269–294.

Buchen, L. (2009). Robot makes scientific discovery all by itself. *Wired UK Online*. https://www.wired.com/2009/04/robotscientist/.

Chang, K., & Myers, B.A. (2014a, October 5–8). Creating interactive web data applications with spreadsheets. In *UIST'14: ACM Symposium on User Interface Software and Technology* (pp. 87–96). Honolulu, Hawaii.

Chang, K., & Myers, B.A. (2014b, July 28–August 1). A spreadsheet model for using web service data. In *VL/HCC'14: IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 169–176). Melbourne, Australia.

Chang, K., & Myers, B.A. (2016, May 7–12). Using and exploring hierarchical data in spreadsheets. In *Proceedings CHI'2016: Human Factors in Computing Systems* (pp. 2497–2507). San Jose, CA.

Clark, L. (2012). Google's artificial brain learns to find cat videos. *Wired UK Online*. https://www.wired.com/2012/06/google-x-neural-network/.

Cypher, A., Halbert, D. C., Kurlander, D., Lieberman, H., Maulsby, D., Myers, B. A., Turransky, A. (1993). *Watch what I do: programming by demonstration*. Cambridge, MA: MIT Press.

Fisher, II, M., & Rothermel, G. (2004). *The EUSES spreadsheet corpus: a shared re-source for supporting experimentation with spreadsheet dependability mechanisms*. Lincoln: University of Nebraska. Technical Report 04-12-03.

Green, T. R. G., & Petre, M. (1996). Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, *7*(2), 131–174.

Grigoreanu, V., Fernandez, R., Inkpen, K., Robertson, G. (2009, September 20–24). What designers want: needs of interactive application designers. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC'09* (pp. 139–146). Corvallis, Oregon.

Henley, A.Z., & Fleming, S.D. (2016, September 4–8). Yestercode: improving code-change support in visual dataflow programming environments. In: *VL/HCC'16: IEEE Symposium on Visual Languages and Human-Centric Computing*. Cambridge.

Hill, C., Bellamy, R., Erickson, T., Burnett, M. (2016). Trials and tribula-tions of developers of intelligent systems: a field study. In *VL/HCC'2016: IEEE Symposium on Visual Lan-guages and Human-Centric Computing* (pp. 162–170). Denver, CO.

Kery, M.B., Horvath, A., Myers, B.A. (2017, May 6–11). Variolite: supporting exploratory programming by data scientists. In *Proceedings CHI'2017: Human Factors in Computing Systems* (pp. 1265–1276). Denver, CO.

Ko, A. J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., et al. (April, 2011). The state of the art in end-user software engineering. *ACM Computing Surveys*, *43*(3), Article 21 44 pages.

Ko, A.J., & Myers, B.A. (2004, April 24–29). Designing the whyline, a debugging interface for asking why and why not questions about runtime failures. In *Proceedings CHI'2004: Human Factors in Computing Systems* (pp. 151–158). Vienna, Austria.

Ko, A. J., & Myers, B. A. (2005, February). A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages and Computing*, *16*(1), 41–84.

Ko, A.J., & Myers, B.A. (2009, April 4–9). Finding causes of program output with the java whyline. In *CHI'2009: Human Factors in Computing Systems* (pp. 1569–1578). Boston, MA.

Ko, A. J., & Myers, B. A. (2010, August). Extracting and answering why and why not questions about java program output. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *20*(2), Article 4 36 pages.

Ko, A.J., Myers, B.A., Aung, H.H. (2004, September 26–29). Six learning barriers in end-user programming systems. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 199–206). Rome, Italy.

Ko, A. J., Myers, B. A., Coblenz, M., Aung, H. H. (2006, December). An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, *33*(12), 971–987.

Kuttal, S.K., Sarma, A., Rothermel, G. (2011). History repeats itself more easily when you log it: versioning for mashup. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 69–72). Pittsburgh, PA.

Li, T., Azaria, A., Myers, B. (2017, May 6–11). SUGILITE: creating multimodal smartphone automation by demonstration. In *Proceedings CHI'2017: Human Factors in Computing Systems* (pp. 6038–6049). Denver, CO.

Lin, J., Wong, J., Nichols, J., Cypher, A., Lau, T. A. (2009). End-user programming of mashups with vegemite. *Proceedings of the 14th International Conference on Intelligent User Interfaces* (pp. 97–106). Sanibel Island, FL: ACM.

Loksa, D., Ko, A. J., Jernigan, W., Oleson, A., Mendez, C. J., Burnett, M. M. (2016). Programming, problem solving, and self-awareness: effects of explicit guidance. *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (pp. 1449–1461). Santa Clara, CA: ACM.

Meyerovich, L. A., Guha, A., Baskin, J., Cooper, G. H., Greenberg, M., Bromfield, A., et al. (2009). Flapjax: a programming language for Ajax applications. *SIGPLAN Notices (Proc. OOPSLA'2009)*, *44*(10), pp. 1–20. 1640091.

Muncy, J. (2016). Making AI play lots of videogames could be huge (No, Seriously). *Wired UK Online*. https://www.wired.com/2016/04/videogames-ai-learning/.

Myers, B.A. (1991, November). Separating application code from toolkits: eliminating the spaghetti of call-backs. In *UIST'91: ACM SIGGRAPH Symposium on User Interface Software and Technology* (pp. 211–220). Hilton Head, SC.

Myers, B.A. (1998, April). Scripting graphical applications by demonstration. In *SIGCHI'98: Human Factors in Computing Systems* (pp. 534–541). Los Angeles, CA.

Myers, B. A., Hudson, S. E., Pausch, R. (2000, March). Past, present and future of user interface software tools. *ACM Transactions on Computer Human Interaction*, *7*(1), 3–28.

Myers, B. A., Ko, A. J., LaToza, T. D., Yoon, Y. S. (2016, July). Programmer are users too: human centered methods to improve software development. *IEEE Computer*, *49*(7), 44–52.

Myers, B., McDaniel, R., Wolber, D. (2000, March). Programming by example: intelligence in demonstrational interfaces. *Communications of the ACM*, *43*(3), pp. 82–89.

Myers, B.A., Park, S.Y., Nakano, Y., Mueller, G., Ko, A. (2008, September 15–18). How designers design and program interactive behaviors. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC'08* (pp. 185–188). Herrsching am Ammersee, Germany.

Norman, D. A. (1988). *The design of everyday things*. New York: Doubleday.

Oney, S., Myers, B.A., Brandt, J. (2012, October 7–10). ConstraintJS: programming interactive behaviors for the web by integrating constraints and states. In *UIST'2012: ACM Symposium on User Interface Software and Technology* (pp. 229–238). Cambridge, MA.

Oney, S., Myers, B.A., Brandt, J. (2014, October 5–8). InterState: a language and environment for expressing interface behavior. In *ACM Symposium on User Interface Software and Technology, UIST'14* (pp. 263–272). Honolulu, Hawaii.

O'Toole, J. (2014, February 28). Mobile apps overtake PC Internet usage in U.S. *CNN Money*. http://money.cnn.com/2014/02/28/technology/mobile/mobile-apps-internet/.

Ozenc, K., Kim, M., Zimmerman, J., Oney, S., Myers, B. (2010, April 10–15). How to support designers in getting hold of the immaterial material of software. In *CHI'2010: Human Factors in Computing Systems* (pp. 2513–2522). Atlanta, GA.

Pane, J. F., & Myers, B. A. (2006). More natural programming languages and environments. H. Lieberman, F. Paterno, V. Wulf (Eds.). *End-User development* (pp. 31–50). Dordrecht: Springer.

Park, S., Myers, B., Ko, A. (2008, September 15–18). Designers' natural descriptions of interactive behaviors. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC'08* (pp. 185–188). Herrsching am Ammersee, Germany.

Patel, K. D. (2013). *Lowering the barrier to applying machine learning*. Seattle, WA: University of Washington. PhD Dissertation.

Scaffidi, C., Myers, B., Shaw, M. (2008, May 10–18). Topes: reusable abstractions for validating data. In *ICSE'08: International Conference on Software Engineering* (pp. 1–10). Leipzig, Germany.

Sheil, B. (1983, February). Environments for exploratory programming. In *Datamation*. Reprinted in in "Papers on Interlisp-D," Sheil, B.A. and Masinter, L.M., eds., Xerox PARC Tech Report CIS-5.

Smith, A. (2015, April 1). U.S. smartphone use in 2015. *Pew Research Center*. http://www.pewinternet.org/2015/04/01/us-smartphone-use-in-2015/.

Sun, M., Chen, Y.N., Rudnicky, A.I. (2016, March 10). Learning user intentions spanning multiple domains. In *Proceedings of IUI 2016 Workshop on Interacting with Smart Objects (SmartObjects 2016)*. Sonoma, California.

Yang, H., Pupons-Wickham, D., Chiticariu, L., Li, Y., Nguyen, B., Carreno-Fuentes, A. (2013). I can do text analytics!: designing development tools for novice developers. In *CHI'2013: Human Factors in Computing Systems* (pp. 1599–1608). Paris, France.

Yoon, Y.S., Koo, S., Myers, B.A. (2013, September 15–19). Visualization of fine-grained code change history. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'13)* (pp. 119–126). San Jose, CA.

Yoon, Y.S., & Myers, B.A. (2012, June 2). An exploratory study of backtracking strategies used by developers. In *Cooperative and Human Aspects of Software Engineering (CHASE'2012), An ICSE 2012 Workshop* (pp. 138–144). Zurich, Switzerland.

Yoon, Y.S., & Myers, B.A. (2014, 28 July–1 August). A longitudinal study of programmers' backtracking. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'14)* (pp. 101–108). Melbourne, Australia.

Yoon, Y.S., & Myers, B.A. (2015, May 16–24). Supporting selective undo in a code editor. In *37th International Conference on Software Engineering, ICSE 2015* (vol. 1; pp. 223–233). Florence, Italy.

Zang, N., Rosson, M.B., Nasser, V. (2008). Mashups: who? what? why? In *CHI'08 Extended Abstracts on Human Factors in Computing Systems* (pp. 3171–3176). New York, NY.