Software Engineering for Humans

My interest in computer science research is fundamentally driven by the desire to build better systems with solutions framed by theoretical concepts and shaped by real-world applicability. I believe that research has substantial impact if it touches the lives of people and helps solve their day-to-day problems simply and elegantly. I am interested in inventing technologies by studying and modeling both *human factors* and *software engineering factors* in the context of programmers' tasks. My research interests span the areas of Software Engineering, Empirical Evaluation and Human-Computer Interaction. The primary goal of my research is to empower end-user programmers (non-professional software developers) by integrating software engineering activities into their existing workflow without changing the nature of their work or priorities.

Motivation. End users are creating complicated applications using domain specific languages. For example, mobile-savvy users are creating upmarket mobile apps using App Inventor, web developers are creating sophisticated web mashup applications using Yahoo! Pipes, designers are creating dynamic websites using JavaScript, data analysts are writing complex queries to retrieve data from massive data repositories using QuantCell etc. However, these programming environments lack software engineering support, making software application developed by end-user programmers difficult *to maintain, create, reuse, locate* and *fix*.

End-user software engineering can be broadly characterized as implicit, unplanned and opportunistic [5]. End-user programmers *prefer speed and ease, over robustness and maintainability* [12]. They typically *learn from available examples* and opportunistically employ *reuse* by "clone and own" mechanism while programming [5]. They tend to create applications viewing them to be "throw away" programs but their code ends up being long-lived and reused by other end-user programmers [5]. Moreover, end-user programmers' *requirements are ill defined* and there is no single correct way to satisfy them. While creating programs, end users' design decisions and coding activities are typically interleaved [10]. These facts encourage end users to *program opportunistically* [12] and *debug their programs into existence* [11]. End users program opportunistically as there tend to be several alternative solutions available, each with their own strengths and weaknesses, from which they can proceed. End users debug their programs into existence in that they investigate alternative strategies and backtrack through changes to arrive at solutions. None of these issues are well supported by existing end-user programming environments and this elevates programming barriers for end-user programmers.

Approach. Much of my work is result driven and looks at my own existing design and evaluation practice critically. I am motivated to develop new *strategies, theories, visualizations* and *prototypes* for users. I follow evidence-driven engineering processes and rely largely on user-centered design (Design-Prototype-Evaluate trend) to articulate the need to identify and test hypotheses related to end users' behavior and preferences. My research includes mixed methods -- from designing and building interactive systems by using *qualitative* and *quantitative* methods, to investigating user needs, current processes, and tool adoption in the field. While my research and the tools that I build directly help end users, they reveals users' behaviors and strategies while interacting with such systems and reveal different aspects of how modern programming environments can be designed, developed, and made capable of driving the future design of such systems.

Helping Users Utilize Software Engineering Techniques

My dissertation work explored various solutions to facilitate end-user programming behavior. Here are the four major projects, which emerged from my research work:

1.1. Support for Variation over Time: Pipe Plumber (TOCHI 2014, IS-EUD 2011, VLHCC 2011)

Problem: End-user programming environments provide central repositories to end users where they can store their programs. My analysis of a Yahoo! Pipes [1] (a web mashup environment) public repository showed that reuse was common, 56.3% of the pipes (programs) were "cloned" (copied from existing pipes) and 27.4% of the pipes contained at least one sub-pipe (a pipe present separately in the repository). In another study of the Yahoo! Pipes repository it was found that 43% of pipes submitted were variations of previously submitted pipes [6]. This indicates that users may be using the public repository as a private repository for manually storing different versions of their pipes that contain incremental changes. However, current environments do not provide facilities by which users can keep track of the versions (previous program states) or provenance of the programs they create. Hence end-user programmers lack basic version information about their programs, which is readily available for professionals.

Solution: I added versioning support called "Pipes Plumber" (Figure 1) to Yahoo! Pipes [1]. To better conceptualize versioning support in the Yahoo! Pipes environment my strategy was to map various tasks performed by users to versioning system concepts. Based on these mappings, I prototyped and implemented the versioning system for Yahoo! Pipes. "Pipes Plumber," keeps version histories of mashups automatically, in a manner that allows users to utilize the advantages of versioning without needing to be aware of the underlying functions known to professional programmers such as check-in, check-out, and so forth.

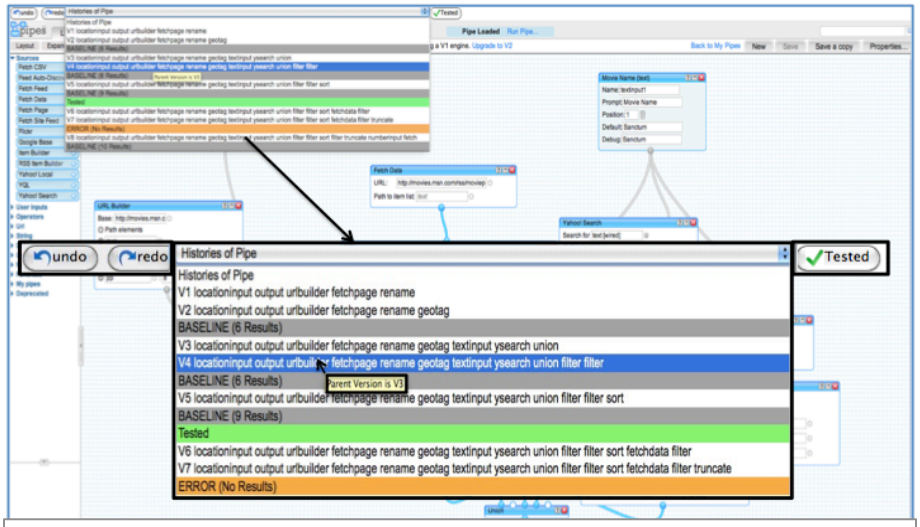

Figure 1: Our Pipe-Plumber adds four widgets. 1) "Undo", 2) "Redo", 3) "Tested", and 4) "History of Pipe". The third widget, "Tested", allows users to indicate that they have confidence in their pipe's correctness. The fourth widget, "History of Pipe" list, allows users to view versions, differences between them, and their operational status (color). To help users debug their programs, we color code the version history in the History of Pipe list with execution results so that users can distinguish between successful pipe runs (Gray), unsuccessful pipe runs (Orange), and pipes that users consider to be "Tested" (Green).

Results: Results from two user studies revealed that versioning support was useful to both end user and more sophisticated programmers working in Yahoo! Pipes. In particular, versioning support helped with the reuse and debugging activities that these programmers engaged in, alleviating some of the programming barriers that they faced. With versioning support, our participants were able to rely not only on examples found in the repository, but on prior versions of pipes as well. Participants with versioning support were less risk averse while creating mashups, and this helped them experiment with, and choose between, different ideas, aided by the ability to return to prior successful and unsuccessful changes.

Research Insights: My research brought benefits of version management into end-users' programming environments to help them overcome programming barriers during program understanding, creation, verification and debugging.
The results of studies drew attention to two main areas in which research is needed.

- *Debugging Support:* Observations underscore a need for better testing techniques and debugging tools for helping end-user programmers create dependable mashups. The user studies showed that end users faced difficulty locating the sources of faults and performed testing activities in an ad-hoc manner. Moreover, support for better error reporting and rigorous methodologies for ensuring dependability would help users while creating pipes.
- *Overcoming Understanding and Use Barriers:* While debugging, after identifying the cause of a fault, an end-user programmer must understand the usage of the program elements to correct it. End users struggled with this (due to use barriers) in tasks such as selecting the appropriate modules and comprehending their usage. Therefore, to facilitate debugging for end users, integrating various strategies for overcoming understanding and use barriers are needed.

The above research insights shaped my next steps, which was how to support end-user programmers with debugging problems in current web mashup programming environments.

1.2 Debugging Support for Mashup Programming Environments (CHI 2013)

Problem: Mashups interact with the web, and the web is a complex ecosystem of heterogeneous formats, services, protocols, standards and languages all of which tend to evolve. The dependence of mashups on this complex ecosystem makes them vulnerable to unexpected behaviors. Debugging techniques such as static or dynamic debugging and source code manipulation are not available in these environments. This hinders developers from understanding when a particular piece of code is executed and in what context. Further, mashups are constrained by API boundaries and the reliance on external sources, which continuously evolve. As a result, run-time observation of program behavior is the primary approach used for debugging mashups.

To better understand the prevalence of bugs in web mashups I analyzed a large corpus (51,468) of Yahoo! Pipes mashups code and execution logs. My study noted that 64.1% of these pipes contained bugs, and the environment detected only 27% of the bugs. Clearly, the prevalence of bugs in pipes and the tendency for users to reuse pipes create problems for mashup dependability. One implication of this data is that users may frequently need to debug mashups. Debugging mashups, however, intrinsically involves distance and visibility issues due to distributed and black box dependencies, and this renders it time and effort intensive. Hence, the time required to debug mashups that inherit faults through reuse will be just as substantial.

Solution: I added debugging support to Yahoo! Pipes (Figure 2) and to connect users to it designed an interface with two primary goals: first, reducing understanding barriers to help users quickly locate and understand the causes of bugs, and second, reducing use barriers by providing guidance on the correct usage of modules. This was achieved by: 1) designing an automated approach for identifying bugs, 2) informing users of bugs and their causes through a user-friendly UI and messages, and 3) offering guidance on ways to fix bugs.

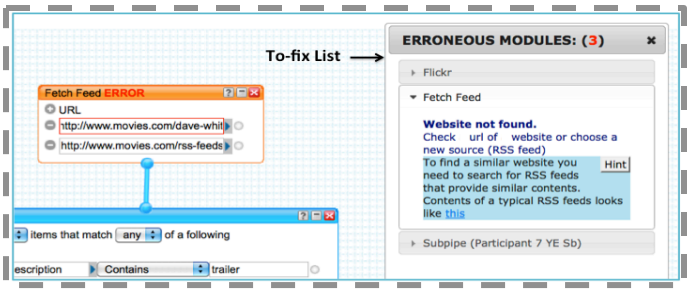

Figure 2: Automatic debugging support for Yahoo! Pipes. To-fix list is designed as an accordion list, where only the bug that is in focus is expanded, while all other information is collapsed. To-fix list displays erroneous modules with concise error message and a possible fix, followed by incremental help in the form of hints. Faulty modules are highlighted (orange) and cross-linked with To-Fix list.

To provide better automatic detection of bugs, I studied pipes extracted from the Yahoo! Pipes repository and created a classification scheme. The bugs were categorized as Intra-module (bugs that occur within a module) and Inter-module (bugs that involve interactions between modules) bugs. Based on these classifications, an anomaly detector that can automatically detect bugs was implemented. My research designed new approaches to handle each type of bug based on the application of *static analysis, dynamic analysis* or a *combination of both* to the program. The users were connected with the detected bugs by developing interface with the main goal of reducing cognitive load by following Nielsen's heuristics [7] and designing error messages by following Shneiderman's guidelines [8]. Constructive steps were provided to help users arrive at solutions to bugs by incremental help in the form of a "Hint" button.

Results: Results from a user study revealed that it was difficult for end users to identify and localize faults without debugging support. It was observed that bugs related to program nesting, silent failures of programs (program fails without any error message), and program reuse were hardest to localize. My research also revealed that control group participants spent substantial time looking for information to locate faults. Some of the participants using our debugging support spent substantial time foraging for information back and forth across the Yahoo! Pipes interface and websites to fix faults.

Research Insights: Debugging enhancements greatly helped mashup programmers localize and efficiently fix bugs. My research also provided design guidelines for incorporating debugging support into end-user environments. These guidelines are: 1) provide automated fault localization, 2) use simple language in error messages, 3) cross-link faults with error messages, 4) provide contextualized help, 5) provide incremental assistance, and 6) provide versioning support. Contextualized help and incremental assistance can be supported by providing a high-level overview of a possible solution, followed by "hints" that users can employ. This helps sustain a user's interest, as they are otherwise likely to be overwhelmed by the number and type of errors

1.3 Information Foraging Theory for End Users' Debugging (VLHCC 2013)

Problem: Mashup debugging involves foraging through the code and understanding web and mashup content, components and behavior. This type of foraging behavior is specific to mashup users. This motivates the need to understand the foraging behavior of end-user programmers in the context of debugging mashups.

Solution: I analyzed the debugging behavior of participants in the earlier study (Section 1.2) from an Information Foraging Theory (IFT) perspective. In IFT theory a predator (end-user programmer) forages for prey (bugs, while finding or fixing) by following cues (e.g., label on links) in patches (e.g., webpages, IDEs). IFT has been studied and applied in connection to the process of "foraging" by users on the web [3], and to professional programmers navigating through programs and debugging [4]. My research developed a model for foraging behavior of end-user programmer while debugging (Figure 3). The study analysis helped in understanding the debugging processes and strategies followed by end users.

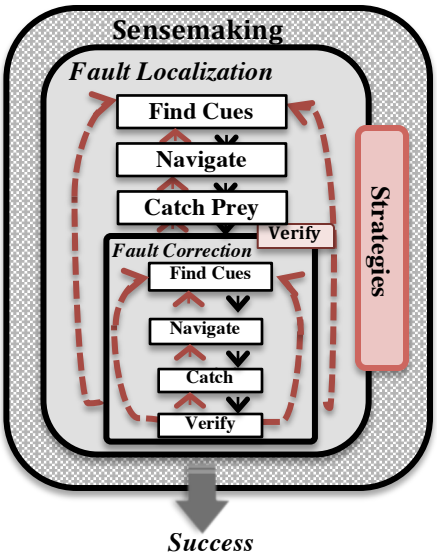

Figure 3: Model for foraging behavior of end-user programmers while debugging. The model has two phases (finding and fixing faults). For each phase, a user's foraging behavior during debugging involves: 1) find cues, which are then processed into scents, 2) navigate through patches, 3) catch prey, and 4) "verify" the viability of the prey. Strategies are cross cutting across both phases.

Research Insights: My research led to several implications for theory and design. The implications for theory can help researchers and practitioners focus on behaviors of users relative to our new model of debugging. My study showed that the cues differ in how well they: 1) add conceptual clarity to error messages, 2) promote detectability, 3) connect users with relevant debugging information, and 4) narrow the search space for users. It also showed various strategies followed by end-user programmers while finding and fixing the faults. I also discovered several ways in which mashup programming environments, and end-user programming

environments in general, can be enhanced to provide better cues in order to support end users' debugging strategies. Implications for design can help inform better design of interfaces for visual programming/black-box oriented environments. Some of the guidelines include strengthening information scents (chances of finding prey) through cue clarity, making diet constraints (i.e., what users should look for) explicit, supporting multi-context views, supporting a To-fix list, and supporting fine-grained backtracking.

1.4 Support for Variation over Space: AppInventorHelper (VLHCC 2013)

Problem: Finding code and abstractions to reuse, or even knowing that they exist, is a challenging task for end-user programmers. Changing requirements, code reuse, and programming styles (opportunistic programming and debugging programs into existence) tend to create large numbers of program variants scattered throughout code repositories.

To better understand the usage of variants by end-user programmers, I conducted an online survey to determine how and why end users create, find and manage variants. The results helped us discover several design requirements for an end user variation management system. The results of the survey revealed that end users do create variants frequently (88%). End-user programmers do not use tools to track changes or manage variants but rely primarily on memory to manage variants (82%). I discovered that correctness, author names, and similarity among features are attributes used by end users while searching for specific variants. To find specific variants, respondents considered program output. Therefore, there is a need for automatic support for variation management, which can provide the features that end users rely on, and improve their ability to access and use variants.

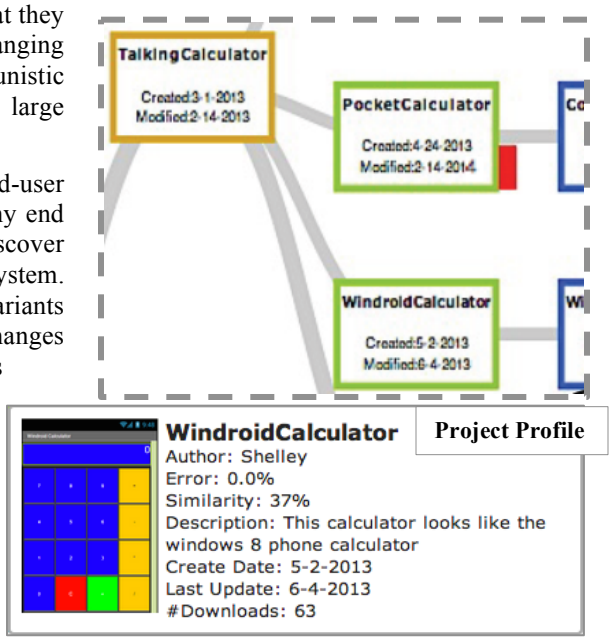

Figure 4: AppInventorHelper. The nodes of the graph represents the app. developed and edges represent the provenance of the app. Different colors of the nodes represent different authors and edge width represents the similarity between parent and children apps. The red bar represents the percentage of errors in the app. Project Profile represents the meta-information about an app and can be viewed by hovering (mouse) over the application of interest.

Solution: To support automatic variation management for end users, I developed AppInventorHelper (Figure 4), a variation management system for end-users working in the App Inventor [2] environment (an Android mobile application development environment). AppInventoHelper visualization was designed to help end users 1) organize their program variants automatically, 2) visualize all program variants at once, and 3) select appropriate program variants based on parameters such as similarity of code, authors, date of creation, and date of update.

Results: Results from a user study showed that AppInventorHelper helps users in exploring and reusing variants, and efficiently understanding the evolution of code.

Toward Future Research

The goal of my research is to broaden the impact of human factors in software engineering techniques and tools by expanding the benefits they provide to both software developers and end users. Below is a sampling of both short-term and long-term projects that I am currently involved in or plan to launch (independently or with collaborators):

2.1 Connecting End-user Programmers to the Variants they Need

My dissertation work was an initial step towards understanding the need for variation management for end-user programmers. We are currently developing approaches to help end-user programmers explore prior program states and combine them using JavaScript in Cloud9 [19] – a web development environment. This work is part of the Exploratory Programming project [20], a collaboration involving Carnegie Mellon University (CMU), Oregon State University (OSU), University of Nebraska-Lincoln (UNL), and University of Washington (UW).

Variation Foraging Theory: We believe that information foraging theory may help us to better understand the foraging behavior of end users while performing programming tasks such as program exploration, program understanding, verification and debugging in the context of variations. In collaboration with 2 graduate students and 2 faculty members, I am currently managing and conducting additional studies to understand how end users forage through variants. The results can inform the design of tools to help end-user programmers engage in exploratory programming tasks. The findings will also help in developing algorithms for recommendation systems, focused on modeling the variation foraging behavior of end-user programmers. This project is a collaboration between UNL and OSU.

Avoiding redundancy: Software development encompasses many redundancies, involving code (writing the same code) or development efforts (debugging the same code). I believe that such redundancies are exacerbated in collaborative software development (online repositories), where one developer's effort is not captured and reused by other developers. I want to utilize program partitioning methods and regression testing techniques to avoid redundancies caused by re-analyzing and re-testing changed programs. I believe that the results will be applicable to professional programming environments as well as end-user programming environments.

Extending Idea Garden Principles for Variations: I am also involved in an ongoing project at OSU called the Idea Garden [13]. The Idea Garden supports end-user programming environments by providing context-sensitive advice, mini-patterns and problem-solving strategies that help them learn in the context of their own activities. It has been shown to entice end-user programmers to learn and use concepts and strategies in the context of their own tasks. The Idea Garden approach is a theory-based approach based on minimalist learning theory [21] and has helped end-user programmers transfer their learning into practice. However, the existing Idea Garden approach has considered how end-user programmers generate ideas when learning programming concepts for the first time. We are now exploring the Idea Garden in the context of variations to help end-user programmers with *multiple* idea generation. The suggestions will be informed by statically and dynamically analyzing end-user programmers' source code.

2.2 Big Data for End-user Programmers

As the need for gathering and analyzing large amount of data (commonly known as big data analysis) becomes more prevalent in the business community, there is an increased demand for system and language environments that can run on inexpensive hardware and software that can be programmed and operated by programmers and analysts with average mainstream skills. The users of these systems are end-user programmers who are domain experts in their field, data scientists, analysts, researchers, and consumers of the analyses (decision makers, managers) [16].

Software Engineering perspective: According to the 2011 McKinsey report, the potential value of global personal location data is estimated to be $700 billion and can result in up to a 50% decrease in product development and assembly costs. The report also predicts the equally great effect of big data on employment: 140K-190K workers with deep analytical experience will be needed in the US and 1.5 million managers will need to become data-literate [17]. Recently researchers have started looking for end-user programming languages to support big data (e.g., QuantCell [14] (a big data spreadsheet), Optique [15]). Drawing from my previous work in end-user software engineering, I would like to develop techniques for automatically detecting bugs by analyzing program code statically, dynamically, or with a combination of both techniques.

HCI perspective: With the increased relevance on big data's numbers, there is a risk of misunderstanding the results and in turn misallocating important public resources. Too many big data projects stall or fail [16]. An example is that of public health officials who relied exclusively on Google Flu trends, which mistakenly estimated that peak flu levels reached 11% of the US public (2012-13 flu season), almost double the CDC's estimate (6%) [18]. To help end users analyze data, I want to provide supplementary information along with the results to provide a better explanation of how the results were derived, based on the inputs, i.e., the provenance of the result data.

2.3 Information Foraging Theory in Software Engineering

Information Foraging Theory (IFT) is a behavioral theory of how humans seek information from a computing environment [3]. My vision is to leverage IFT to build a foundation for Software Engineering (SE) approaches for both professional programmers and end-user programmers when involved in information-intensive software engineering tasks.

IFT for end-user programmers: My research projects have explored different type of cues, strategies and behavior of end-user programmers while performing programming tasks. In the future, I want to conduct similar studies using IFT in big data and cloud computing environments. Interaction is a powerful way for exploring and understanding massive datasets. I strongly believe that the designs of the big data and cloud computing environments can directly benefit from the cues and strategies informed from the user studies. Thus, I want to investigate questions such as: "*Can IFT be applied to big data and cloud computing environments?"* and *"Can IFT be exploited to reduce the difficulty of developing and maintaining such software?"*

IFT for professional programmers: I am collaborating with researchers at OSU to understand the motivation of professional programmers' foraging decisions. Based on these findings, we will create a predictive recommendation system for predicting the intended behavior of programmers. I am also involved in another research project working to improve API design in collaboration with a large financial organization. In the future, to enhance IFT's utility for software engineering researchers, I would like to study IFT along with other human-behavioral theories to understand and support software developers' activities, such as collaborative problem solving and design. Such theories will enable software engineering researchers to build SE approaches on solid theoretical foundations that have been empirically demonstrated to work.

References

[1] Yahoo! Pipes: http://pipes.yahoo.com/pipes/

[2] App Inventor: http://appinventorapi.com/

[3] P. Pirolli, *Information Foraging Theory: Adaptive Interaction with Information* Oxford University Press, 2009.

[4] S. D. Fleming, C. Scaffidi, D. Piorkowski, M. Burnett, R. Bellamy, J. Lawrance, I. Kwan, *An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks*, ACM Transactions on Software Engineering and Methodology 22(2), Article 14, 2013.

[5] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, S. Wiedenbeck, *The state of the art in end-user software engineering*, ACM Computing Surveys, 43(3), p.1-44, 2011.

[6] K. Stolee, S. Elbaum, A. Sarma, *End-user programmers and their communities: An artifact-based analysis*, In Proceedings of the empirical software engineering and measurement, p.147–156, 2011

[7] J. Nielsen, R. Molich, Heuristic evaluation of user interfaces. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, p.249–256, 1990.

[8] B. Shneiderman, *Designing computer system messages,* Communication ACM 25, p.610–611, 1982.

[9] L. Grammel, M.-A. Storey, *An end user perspective on mashup makers*, In Technical Report DCS-324-IR, Department of Computer Science, University of Victoria, 2008.

[10] J. Cao, Y. Riche, S. Wiedenbeck, M. Burnett, V. Grigoreanu, *End-user mashup programming: through the design lens*, Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, p.1009-1018, 2010.

[11] M. B. Rosson, J. M. Carroll, *The reuse of uses in Smalltalk programming*, ACM Transactions on Computer-Human Interaction, 3(3), p.219-253, 1996.

[12] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, S. R. Klemmer. *Opportunistic Programming: Writing Code to Prototype, Ideate, and Discover*. IEEE Software, 26(5), 2009.

[13] J. Cao, I. Kwan, F. Bahmani, M. Burnett, S. D. Fleming, J. Jordahl, A. Horvath, S. Yang, End-User Programmers in Trouble: Can the Idea Garden help them to help themselves? *IEEE Symposium on Visual Languages and Human-Centric Computing*, San Jose, CA, p.151-158, 2013.

[14] QuantCell Research: http://www.quantcell.com/web/index.html

[15] Optique Project: http://www.optique-project.eu/

[16] H. V. Jagadish, J. Gehrke, A. Labrinidis, Y. Papakonstantinou, J. M. Patel, R. Ramakrishnan, C. Shahabi, *Big data and its technical challenges,* Communications of the ACM, 57(7), p.86-94.

[17] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers. *Big data: The next frontier for innovation, competition, and productivity.* McKinsey Global Institute, 2011.

[18] Google vs. CDC data: https://hbr.org/2013/04/the-hidden-biases-in-big-data.

[19] Cloud9: https://c9.io/

[20] Exploratory programming: http://www.exploratoryprogramming.org/

[21] J. Carroll, *The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill*, MIT Press, 1990