Preprint. To appear in: New Perspectives in End-User Development (F. Paterno and V. Wulf, eds.), Springer, (to appear 2017).

Toward Theory-Based End-User Software Engineering

Margaret Burnett[1], Todd Kulesza[2], Alannah Oleson[1], Shannon Ernst[1],
Laura Beckwith[3], Jill Cao[1], William Jernigan[1], Valentina Grigoreanu[2]

[1]Oregon State University, [2]Microsoft, [3]Configit

Abstract

One area of research in the end-user development area is known as end-user software engineering (EUSE). Research in EUSE aims to invent new kinds of technologies that collaborate with end users to improve the quality of their software. EUSE has become an active research area since its birth in the early 2000s, with a large body of literature upon which EUSE researchers can build. However, building upon these works can be difficult when projects lack connections due to an absence of cross-cutting foundations to tie them together. In this chapter, we advocate for stronger theory foundations and show the advantages through three theory-oriented projects: (1) the Explanatory Debugging approach, to help end users debug their intelligent assistants; (2) the GenderMag method, which identifies problems with gender inclusiveness in EUSE tools and other software; and (3) the Idea Garden approach, to help end users to help themselves in overcoming programming barriers. In each of these examples, we show how having a theoretical foundation facilitated generalizing beyond individual tools to the production of general methods and principles for other researchers to directly draw upon in their own works.

1. Introduction

Since the first book on end-user development (EUD) [Lieberman et al. 2006], research on EUD has made significant progress. From its early beginnings focusing mostly on supporting end users' software development using spreadsheets and event-based computing paradigms, EUD research has emerged to support end-user development of web automations, mobile devices, personal information management systems, business processes, programming home appliance devices, and even internet-of-things programming. Further, EUD research now spans much more of the software development lifecycle, supporting not only creating new programs alone and collaboratively and testing/debugging them, but also designing, specifying, and reusing them. For a survey of related works, see [Ko et al. 2011].

However, EUD research as a field also has an important shortcoming: it lacks cross-cutting foundations. Although it is common to ground EUD research efforts in formative empirical studies so as to understand a particular target group of end-user developers, this (important!) practice still leaves a gap: weak connections among similar EUD research projects that target different audiences or tasks. This lack of broad, cross-cutting foundations, in turn, silos our research [Burnett and Myers 2014], making it difficult for EUD researchers to build on one anothers' shoulders in principled ways.

1.1 Theory: What It Is and How It Can Help

This is where theory can help. The essence of theory is generalization through abstraction—mapping instances of successful approaches to cross-cutting principles. In the realm of human behavior, these abstractions can then produce explanations of *why* some software engineering tools succeed at supporting people's efforts and why some tools that were expected to succeed did not.

As Shaw eloquently explains, scientific theory lets technological development pass previously imposed limits inherent in relying on intuition and experience [Shaw 1990]. For example, her summary of civil engineering history points out that structures (buildings, bridges, tunnels, canals) had been built for centuries—*but only by master craftsmen*. Not until scientists developed theories of statics and strength of materials could the composition of forces and bending be tamed. These theories made possible civil engineering accomplishments that were simply not possible before, such

as the routine design of skyscrapers by ordinary engineers and architects [Shaw 1990]. In computer science, we have seen the same phenomenon. Expert developers once built compilers using only their intuitions and experiences, but the advent of formal language theory brought tasks like parser and compiler writing to the level that undergraduate computer science students now routinely build them in their coursework [Aho et al. 2006]. In a more recent example, deSouza shows how semiotic theory can provide a unified theoretical underpinning for the communication aspects of a wide range of EUD projects [DeSouza 2017].

Opinions differ on exactly what a theory is, but as Sjoberg et al. explain, most discussions of what a theory is come down to four points: what a theory *does*, what its *elements* are, where they *come from*, and how they are *evaluated* [Sjoberg et al. 2008]. At least five types of theory have been identified from the perspective of what theory *does* [Gregor 2006], but the types of interest to this chapter are those that explain why something happens, those that predict what will happen under different conditions, and those that prescribe how to do something. According to Sjoberg et al., scholars agree that theory's basic *elements* are constructs, relationship, and scope; and theories *come to* a given discipline by borrowing them from other disciplines (with or without adaptation along the way) or may be generated from scratch within the discipline using data collected. Sjoberg et al.'s fourth point, how theories are *evaluated*, brings up a particularly important point: theories of human behavior, which are the theories of interest to this chapter, can never be completely "proven"—because we cannot see inside humans' heads—but they can be supported or refuted through empirical evidence.

This chapter advocates for more theory-based research in the end-user development domain. Informing EUD research with theories—i.e., being good *consumers* of theory—can lead to the kinds of advantages Shaw described. In terms of Sjoberg et al.'s four points, this chapter is a consumer of theories (1) that explain, predict, or prescribe; (2) that come from other disciplines; (3) whose constructs, relationships, and scope are potentially good fits to end-user development projects; and (4) are already well supported through significant amounts of empirical evidence. Further, informing next steps of our research with principles that we can derive from our results—i.e., being good *producers* of theory—enables EUD researchers to build upon one anothers' findings effectively because the derived principles identify the new fundamentals of successful results. This chapter demonstrates this point by deriving prescriptive theories in the form of principles accompanied by a beginning body of supporting evidence.

1.2 Overview of this chapter: Three examples of theory-based research

To illustrate EUD research from both a theory-consuming perspective and a theory-producing perspective, we present theory-oriented views of three of our recent EUD projects in the area of end-user software engineering.

End-user software engineering (EUSE) is a particular type of EUD research that focuses on the *quality* of end-user developed software [Ko et al. 2011]. A significant challenge in EUSE research is to find ways to incorporate software engineering activities into users' existing workflows without requiring people to substantially change the nature of their work or their priorities. For example, rather than expecting spreadsheet users to incorporate a testing phase into their programming efforts, tools can simplify the tracking of successful and failing inputs incrementally, providing feedback about software quality as the user edits the spreadsheet program. Approaches like these allow users to stay focused on their primary goals (balancing their budget, teaching children, recording a television show, making scientific discoveries, etc.) while still achieving software quality [Ko et al. 2011].

The three EUSE examples in this chapter illustrate how starting with a theory foundation (being *consumers* of theory) facilitated our ability to not only chart a path toward new visions, but also to derive principles to guide others following similar paths in their own works (being *producers* of theory).

We will begin with Explanatory Debugging, an approach to enabling end users without backgrounds in debugging or in computer science to personalize or "debug" their intelligent assistants. Explanatory Debugging draws mainly upon Mental Model Theory [Johnson-Laird, 1983]. From this foundation, we derived a set of principles to form a design-oriented theory about how to create EUD tools that support Explanatory Debugging, instantiated them in a prototype, and used that prototype to empirically evaluate the principles. The second example we discuss, Gender HCI, describes a series of theory-based works that originated in the EUSE domain to investigate how gender differences in problem solving styles come together with software for problem-solving tasks (such as end-user development tasks). It draws from several theories, most notably Self-Efficacy Theory [Bandura 1986] and Information Processing Theory as per Meyers-Levy's Selectivity Hypothesis [Meyers-Levy 1989, Meyers-Levey and Maheswaran 1991, Meyers-Levy and Loken 2014]. Finally, we discuss the Idea Garden, an explanation approach designed to supplement existing end-user

development environments to enable end users who are "stuck" to help themselves to overcome their barriers. As in the Explanatory Debugging approach, the Idea Garden draws from theory—here, the primary theory used was Minimalist Learning Theory [Carroll and Rosson, 1987; Carroll, 1990; Carroll 1998; van der Meij, H. and Carroll 1998]. From these foundations, we derived principles to form a design-oriented theory for how to create Idea Garden tools for EUD environments, instantiated the principles in prototypes, and used the prototypes to empirically evaluate the principles. The evaluation of the principles served a dual role: it not only empirically evaluated the principles themselves, but also evaluated the derivation of the principles (i.e., whether we introduced problems in the course of deriving them from the original theory).

2. Explanatory Debugging

2.1 End Users Personalizing ("Debugging") Machine Learning: Foundations

An increasing amount of software includes some form of machine learning, from facial recognition to personalized search results, document classification, and media recommendation. Often these learning systems allow users to personalize their behavior by providing examples (e.g., "I really liked *Casablanca*, so recommend more movies like it") or by placing limits on the system (e.g., "Only recommend movies in the *film noir* genre").

End users change the system's logic when they personalize it in the above manners, resulting in a software artifact that is now (hopefully) more closely aligned with the way its end-user expects it to behave. We term such changes "debugging" the system—a form of end-user software engineering—because they are changing the learning system to make its behavior better match their requirements.

However, debugging learning systems can be difficult, regardless of the mechanisms available to debug, because many machine learning systems are complex—their internal logic is often Byzantine, and may change after (learn from) each user action. Because of this difficulty, many learning systems do not try to empower end users to understand or debug them—instead, they act as a "black box" to the user. For example, such systems usually provide little or no explanation of their predictions. Further, if the system allows any user feedback at all about a prediction, it is usually limited to users being able to say they disagree with the system's prediction or possibly to say what the answer should be. However, users are rarely allowed to explain *why* they disagree or what the system should start or stop taking into account in its reasoning. In contrast, we subscribe to Shneiderman's viewpoint that, just as professional software developers need to understand the software they maintain and debug, end users likewise need to understand (at least some of) how the learning system works, and further, must be empowered to fix (debug) the system so that its behavior becomes more useful to their needs [Shneiderman 1995].

To investigate how to enable ordinary users to understand such complex systems as machine learning environments, we turned to the mental model theory of reasoning. In this theory, *mental models* are internal representations that people generate based on their experiences in the real world. These models, when they are reasonably accurate, allow people to understand, explain, and predict phenomena, then act accordingly [Johnson-Laird, 1983]. In the context of machine learning systems, reasonably accurate mental models should help users understand why a system is behaving in a given manner and allow them to predict how it will respond if they make specific changes to it. Given this foundation, we hypothesized that by being able to accurately predict how a learning system will respond to specific adjustments, a user with a reasonably accurate mental model will be able to debug the system more successfully than a user whose mental model is flawed.

We explored this hypothesis and investigated the accuracy and malleability of end users' mental models through a succession of user studies [Kulesza et al. 2010, Kulesza et al. 2011, Kulesza et al. 2012, Kulesza et al. 2013, Kulesza et al. 2015]. Critically, we found that users rarely have accurate mental models of how common machine learning systems operate, and that in the absence of explanations, these models do not improve over time through continued interaction with the system. Even brief explanations, however, were able to increase the quality of users' mental models. This increase was often matched by a corresponding increase in users' abilities to personalize the learning system to their satisfaction. Thus, the mental model theory of reasoning helped us identify the importance of explanations to interactive machine learning systems.

The mental model theory of reasoning also helped direct our research efforts as we strove to better understand how end users build mental models of machine learning systems. For example, the theory posits that mental models are "runnable", meaning that people should be able to compare the results of different actions by "running" their mental

model on multiple inputs. This implied that our research should consider how much and what kind of information end users need in order to create runnable models, as well as how users might want to use the insights gained from these models to correct their learning systems.

Our Explanatory Debugging approach for interactive machine learning systems was born from these foundations. Our formative research found a significant correlation between the quality of a user's mental model and their ability to control the learning system as desired, suggesting that the better someone understands the underlying system, the better they will be able to control it [Kulesza et al. 2012]. Explanatory Debugging is therefore characterized by eight principles: four for *explainability*, and four for *correctability*.

2.2 The Explainability Principles

The first four principles of Explanatory Debugging are intended to enable end users to build high-quality mental models. They must, however, be carefully balanced: there is a tension among these four principles, such that increasing some of them may cause undesirable decreases in others.

Principle 1: Be Iterative

Our formative research suggested that users personalize a learning system best if they build their mental models *while interacting with it* [Kulesza et al. 2012], and therefore that explanations should support an iterative, *in situ* learning process. For example, explanations could take the form of concise, easily consumable "bites" of information in the context of the system's recent outputs. Such incremental, situated ways of explaining can thereby allow more interested users to incrementally attend to more of these explanations, while still allowing less interested users to incrementally attend to fewer bite-sized chunks of information, all in the context of the system's most recent actions on their behalf.

Principle 2: Be Sound

We use the term *soundness* to mean the extent to which each component of an explanation's content is truthful in describing the underlying system [Kulesza et al. 2013]: does the explanation include "nothing but the truth"? Kulesza et al. detailed the impact of explanation fidelity on mental model development, finding that users did not trust—and thus, were less likely to attend to—the least sound explanations [Kulesza et al. 2013]. Thus, because reducing soundness reduces the likelihood that users will invest attention toward it, Explanatory Debugging requires designing explanations that are as sound as practically possible.

One method for evaluating explanation soundness is to compare the explanation with the learning system's mathematical model. For each of the model's terms that are included in the explanation, how accurately is it explained? If those terms are derived from more complex terms, is the user able to "drill down" to understand those additional terms? The more accurately these explanations reflect the underlying model, the more sound the explanation is.

Principle 3: Be Complete

Completeness is a complement to soundness, and describes the extent to which *all* of the underlying system is included in the explanation: does it explain "the whole truth"? Thus, a complete explanation does not omit important information about the model. In our formative research, we found that end users built the best mental models when they had access to the most complete explanations, which informed them of all the information the learning system had at its disposal and how it used that information [Kulesza et al. 2013]. Also pertinent is work showing that users often struggle to understand how different parts of the system interact with each other [Kulesza et al. 2011]. Complete explanations that reveal how different parts of the system are interconnected may help users overcome this barrier.

One method for evaluating completeness is via Lim and Dey's intelligibility types [Lim and Dey 2009]; a more complete explanation system will cover more of these intelligibility types than a less complete system.

Principle 4: Don't Overwhelm

Balanced against the soundness and completeness principles is the need to remain comprehensible and to engage user attention. Achieving this goal while maintaining reasonably high soundness and completeness is still an underexplored problem, but the following approaches provide several starting points.

Findings from [Kulesza et al. 2013] suggest that one way to engage user attention is to frame explanations concretely, such as referencing the predicted item and any evidence the learning system employed in its prediction. In some

circumstances, selecting a more comprehensible machine learning model may also be appropriate. For example, a neural network can be explained as if it were a decision tree [Craven and Shavlik 1997], but this reduces soundness because a different model is explained, and may in turn require an additional explanation of the differences between the two models. Similarly, a model with 10,000 features can be explained as if it only used the 10 most discriminative features for each prediction, but this reduces completeness by omitting information that the model uses. Still, when the differences in outcome are small, the omissions may not be problematic. Alternative approaches that embody the Explanatory Debugging principles include selecting a machine learning model that can be explained with little abstraction (e.g., [Lacave and Diez 2002, Stumpf et al. 2009, Szafron et al. 2003]) or using feature selection techniques [Yang and Pedersen, 1997] in high-dimensionality domains to prevent users from struggling to identify which features to adjust (as happened in [Kulesza et al. 2011]).

2.3 The Correctability Principles

Given an appropriate mental model, we posited that users would be able to correct a learning system that has somehow gone awry, such as by having not enough data or having received skewed data. To empower them to do this, we added four correctability principles to our Explanatory Debugging approach; these four correctability principles are interdependent on the four explainability principles. In other words, besides informing correctability itself, the correctability principles also reinforce—and even suggest mechanisms helpful to—instantiating the explainability principles.

Besides the mental model theory of reasoning, two additional theoretical foundations were prominent in informing correctability. The first was the *attention-investment model* [Blackwell 2002], which posits that users will invest their attention toward something (e.g., learning a little bit more about a specific machine learning system) if they expect their benefits (e.g., time saved through better recommendations or predictions) to be greater than the costs and risks (e.g., time, effort, or potential lack of payoff) of investing their attention. This model helped us shape an interactive paradigm in which users immediately see the benefits of their feedback, with the intent of encouraging further interaction. *Minimalist learning theory* (to be more fully introduced in Section 4.1) also informed our approach, especially as it relates to helping users learn and work with a concept *in-situ* [van der Meji and Carroll 1998]. (Both of these foundations were also useful in informing Explainability Principle 1.)

Principle 5: Be Actionable

Both theory [Blackwell 2002] and prior empirical findings [Bunt et al. 2012, Kulesza et al. 2012, Kulesza et al. 2013] suggest that end users will ignore explanations when the benefit of attending to them is unclear. By making explanations actionable, we hoped to lower users' perceived (and actual) cost of attending to them by obviating the need to transfer knowledge from one part of the user interface (the explanation) to another (the feedback mechanism). Actionable explanations also fulfill three aspects of minimalist learning [van der Meji and Carroll 1998]: (1) people are learning while performing real work; (2) the explanatory material is tightly coupled to the system's current state; and (3) people can leverage their existing knowledge by adjusting the explanation to match their own mental reasoning.

Principle 6: Be Reversible

A risk inherent in enabling users to provide feedback to a machine learning system is that they may actually make its predictions worse (e.g., [Kulesza et al. 2010, Stumpf et al. 2009]). Being able to easily reverse a harmful action can help mitigate that risk, which is especially important to the risk component of the attention investment model. Reversibility may also encourage self-directed exploration and tinkering, which can facilitate learning [Rowe 1973]. When combined with Principle 8, reversibility also fulfills a fourth aspect of minimalist learning [van der Meji and Carroll 1998]: helping people identify and recover from errors.

Principle 7: Always Honor User Feedback

As Yang and Newman found when studying users of smart home thermostat systems (which learn from their users to predictively adjust the home's temperature) [Yang and Newman 2013], a system that appears to disregard user feedback deters users from continuing to provide feedback. However, methods for honoring user feedback are not

always straightforward. Handling user feedback over time (e.g., what if new instance-based feedback[1] contradicts old instance-based feedback?) and balancing different types of feedback (e.g., instance-based feedback versus feature-based feedback[2]) requires careful consideration of how the user's feedback will be integrated into the learning system.

Principle 8: Incremental Changes Matter

In our formative work [Kulesza et al. 2013], participants claimed they would attend to explanations only if doing so would enable them to more successfully control the learning system's predictions, a result predicted by the attention-investment model. Thus, continued user interaction likely depends on users being able to see the incremental effects their feedback has had on the learning system's reasoning immediately after each interaction. (This is an example of closing Norman's gulf of evaluation—enabling the user to see the results of their last action to interpret the state of the system, so as to evaluate how well their expectations and intentions were met [Norman 2002]). This principle is also related to Principle 1 (Be iterative) because our thesis is that users will develop better mental models iteratively, requiring many interactions with the learning system. These interactions may not always result in large, obvious effects, so being able to communicate even small, incremental effects a user's feedback has had upon a learning system may be critical to Explanatory Debugging's feasibility.

2.4 EluciDebug: A Prototype of Explanatory Debugging in Action

To evaluate the Explanatory Debugging approach's viability with end users, we prototyped it in the context of text classification. We designed a prototype, which we call EluciDebug, to look like an email program with multiple folders, each representing a particular topic (Figure 1). The prototype's machine learning component attempts to automatically classify new messages into the appropriate folder.

To support Explanatory Debugging's Soundness and Completeness principles (Principles 2 and 3), we designed our EluciDebug explanations to detail *all* of the information the classifier could potentially use when making predictions (*completeness*) and to *accurately* describe how this information is used (*soundness*). For example, the explanation shown in Figure 2 tells users that both feature presence and folder size played a role in each prediction. The explanation shown in Figure 3 builds on this, telling the user all of the features the classifier knows about and may use in its predictions. To make clear to users that these features can occur in all parts of the document—message body, subject line, and sender—EluciDebug highlights features in the context of each message (Figure 1, part C).

[1] Instance-based feedback, also known as label-based feedback, is when a user tells a machine learning system what a specific item's predication label should be. A common example is telling a junk mail filter that a specific email message is (or is not) SPAM.

[2] Feature-based feedback is when the user tells a machine learning system which particular features of the data (e.g., which words or fields of email messages) it should or should not use in its reasoning. One example would be a user telling a junk mail filter that every message from a specific email address should always go to the junk folder.

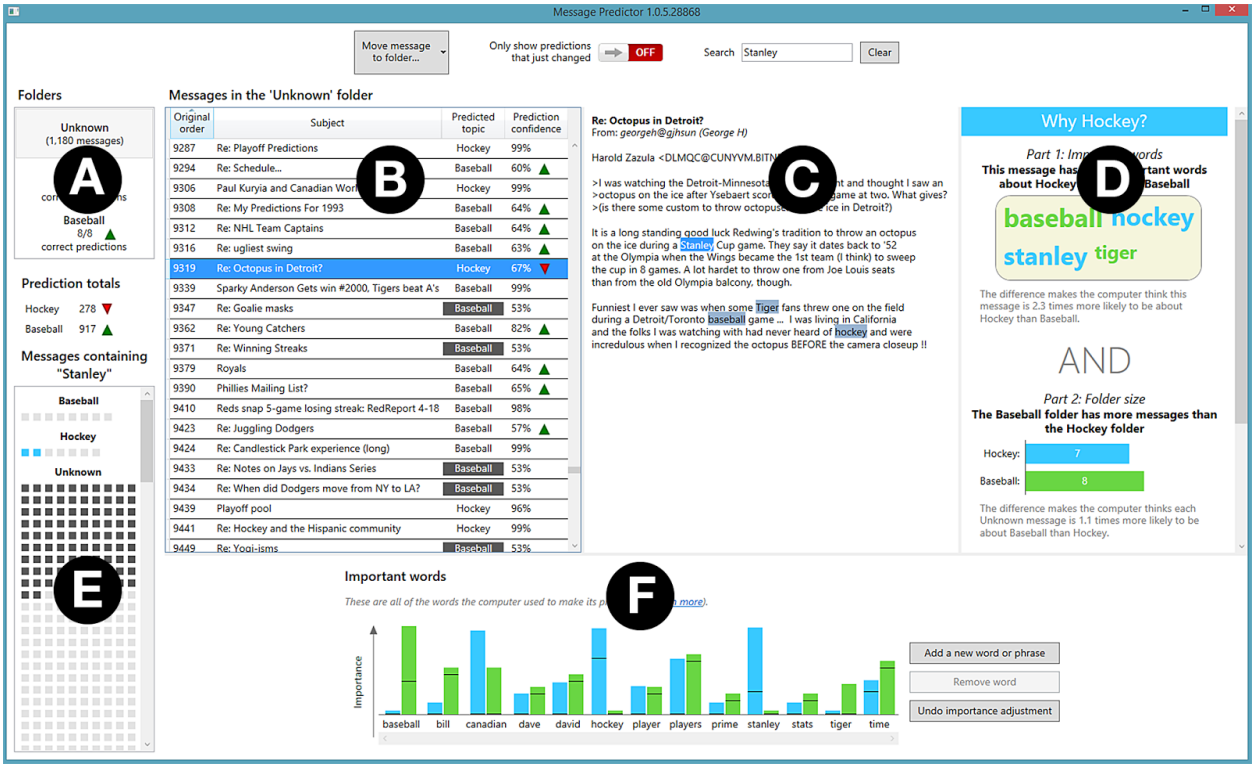

Figure 1: The EluciDebug prototype. (A) List of folders. (B) List of messages in the selected folder. (C) The selected message. (D) Explanation of the selected message's predicted folder. (E) Overview of which messages contain the selected word. (F) Complete list of words the learning system uses to make predictions.

The *Why* explanation (Figure 2), which was inspired in part by the WhyLine for end-user debugging [Ko and Myers 2004], is a concrete explanation and includes only information that was used for the selected prediction. This brevity is intentional; as per Principle 1, it is intended to be consumed iteratively, with users learning more about the overall system as they attend to more *Why* explanations. Such a design also supports Principle 4 by reducing the amount of information continuously shown to end users, to avoid overwhelming them with details that they may not (yet) care about.

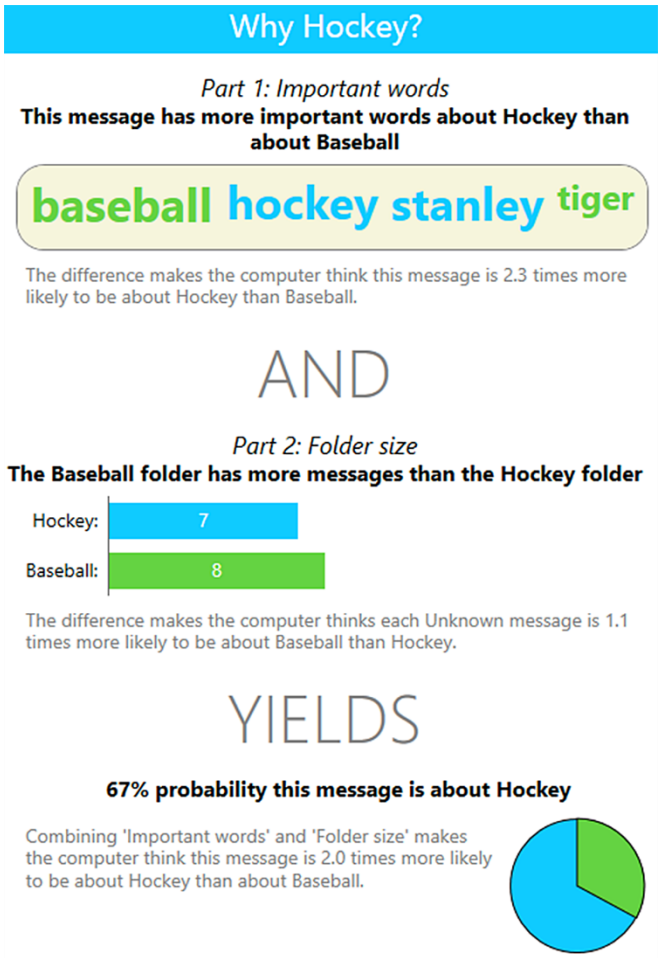

Figure 2: The *Why* explanation tells users how features and folder size were used to predict each message's topic. This figure is a close-up of Figure 1 part D.

The *Important words* explanation (Figure 3) is the primary actionable explanation in EluciDebug (Principle 5). Users can add words to—and remove words from—this explanation, which in turn will add those words to (or remove them from) the machine learning model's feature set. Users are also able to adjust the importance of each word in the explanation by dragging the word's bar higher (to make it more important) or lower (to make it less important), which then alters the corresponding feature's weight in the learning model. As described in [Kulesza et al. 2015], EluciDebug incorporates these corrections into its classifier in such a way that the user's feedback is always honored (Principle 7).

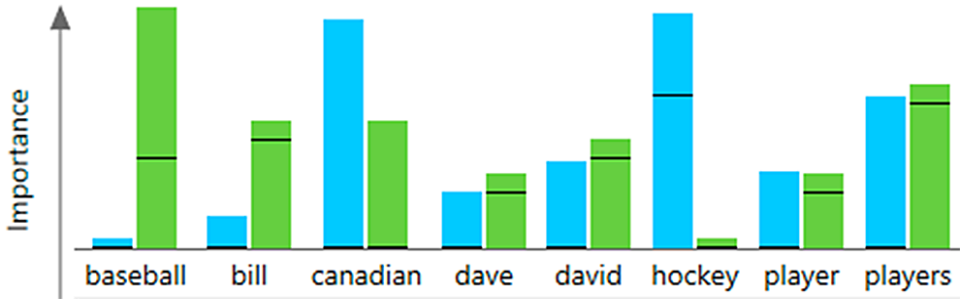

Figure 3: The *Important words* explanation tells users all of the features the classifier is aware of, and also lets users add, remove, and adjust these features. Each topic is color-coded (here, the left, blue bar in a pair for *hockey* and right, green bar for *baseball*) with the difference in bar heights reflecting the difference in the word's probability with respect to each topic (e.g., the word *canadian* is roughly twice as likely to appear in a document about *hockey* as one about *baseball*, while the word *player* is about equally likely to appear in either topic). This figure is an excerpt from Figure 1 part F.

Finally, EluciDebug incorporates infinite undo functionality to allow the user to retrace their steps as far back as they like (Principle 6), and highlights recent changes to the classifier's predictions and certainty levels (Principle 8). The latter is especially important for motivation, because a single user correction often will not cause a classifier's predictions to change; instead, the classifier's *certainty* in predictions will shift, becoming either more or less certain. By observing these certainty changes, users can see how the system is honoring their feedback and can identify whether it is moving in the right or wrong direction.

2.5 Evaluation

We conducted an empirical evaluation to learn whether the Explanatory Debugging principles, as instantiated in the EluciDebug prototype, would indeed enable users to effectively and efficiently "debug" a machine learning system. The full details of our experiment design and procedures are enumerated in [Kulesza et al. 2015], but we present a brief summary of the results below to highlight the outcomes achievable by grounding EUSE research in theory.

Overall, Explanatory Debugging's cycle of explanations—from the learning system to the user, and from the user back to the system—resulted in smarter users and smarter learning systems. For example, treatment participants (i.e., those using the Explanatory Debugging prototype) understood how the learning system operated about 50% better than control participants using a traditional learning system that lacked actionable explanations (Figure 4). Further, this improvement correlated with the F1 scores of participants' classifier accuracy ($\rho[75] = .282$, $p = .013$), which were significantly more accurate (given the same number of corrections) for EluciDebug participants than for control participants (Figure 5). Finally, participants liked Explanatory Debugging, rating EluciDebug significantly better than the control group and responding enthusiastically to the system's explanations.

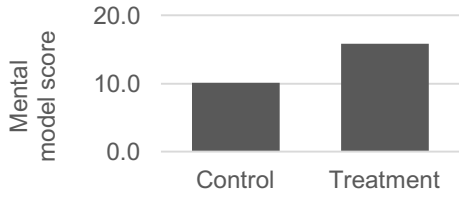

Figure 4: Participants who used EluciDebug for 30 minutes learned approximately 50% more about how the classifier worked than control participants.

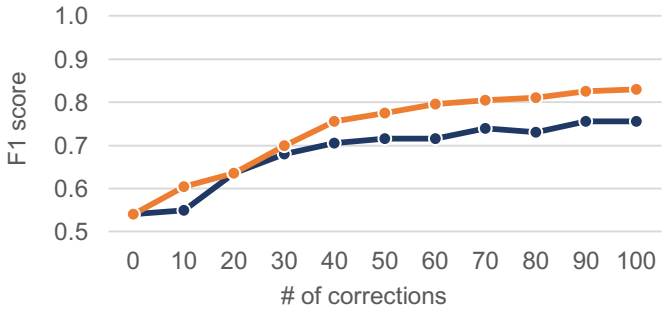

Figure 5: Corrections provided by participants using EluciDebug (light orange) resulted in significantly more accurate classifiers than the same number of corrections provided by control participants (dark blue).

We attribute these positive outcomes to Explanatory Debugging's strong theoretical foundations. The psychological theories of how people learn, understand, and invest their attention when working with interactive systems provided us the guidance to stay on an efficient research track. This guidance not only foretold our promising results, but enabled us to avoid false starts in wrong directions that might have later needed to be discarded and redone. For example, the mental model theory of reasoning guided our approach to enable ordinary end users to understand a fairly complex machine learning system. The attention investment and minimalist learning models combined with the mental model foundation to guide our research toward the correctability principles.

Finally, these theoretical foundations enabled us to think about Explanatory Debugging at the level of principles rather than systems and UI widgets, so that future approaches for different systems and situations can still leverage them. We hope that others will build upon these principles for interactive machine learning to further advance the state of the art in producing more controllable and satisfying experiences for end users than traditional machine learning systems, enabling ordinary end users to gain the most benefit from the learning systems on which they increasingly depend.

3. Gender HCI and GenderMag

In the Gender HCI project, our path toward improving EUSE systems again began with theories, then moved to an iterative series of refinements and applications of the theories, and ultimately produced a general method that we call GenderMag. The overall project goals were (1) to investigate whether gender differences reported in social science domains, such as psychology and education, applied to EUSE tools; and when the importance of these differences became clear, (2) to develop a theory-based method that would enable creators of EUSE environments and tools to identify features in their software that were not gender-inclusive.

Toward these ends, we built on social science theories using the research paradigm presented in Figure 6. We began with social science literature containing theories potentially applicable to gender differences in how people problem-solve with computers. We then derived hypotheses about what these theories might predict in EUSE situations and empirically tested these hypotheses in controlled laboratory studies. Using the results of these studies to guide our designs, we prototyped new features in our EUSE tools and again evaluated them empirically, keeping in mind the foundational theories, and decided whether the EUSE setting required further potential refinements. The highly iterative process then began again with refined theories.

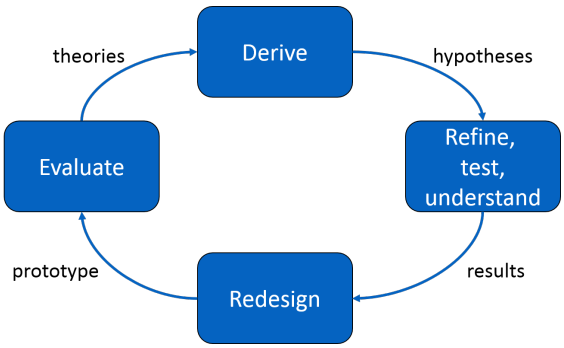

Figure 6: The Gender HCI project's iterative research paradigm.

We illustrate this paradigm using two of the theories that strongly influenced the Gender HCI project: self-efficacy theory and information processing theory.

3.1 A GenderMag Foundation: Self-Efficacy Theory

Self-efficacy is a person's belief that they can succeed at a specific task [Bandura 1986]. It is a form of self-confidence specific to a situation or task; for example, someone may have high bike-riding self-efficacy and low computer self-efficacy. According to this theory, someone with high self-efficacy who encounters problems will persist through adversity, such as by trying different approaches and strategies to overcome the problems. In contrast, someone with low self-efficacy is more likely to stay with faulty approaches and strategies and to experience high levels of self-doubt. As a result, low self-efficacy individuals are often unsuccessful in their tasks, leading to lower and lower self-efficacy in an unfortunate feedback loop. Research shows that self-efficacy is particularly important when it comes to challenging tasks and is a reasonably accurate predictor of success and behavior in such situations [Bandura 1986].

Computer self-efficacy is a person's belief about their capabilities to use computers in a variety of situations [Compeau and Higgens 1995]. Previous studies have found that females generally have lower computer self-efficacy than males. This holds true across a number of computing situations, from females majoring in computer science to female end users [Appel et al. 2011; Beyer et al. 2003; Huffman et al. 2013; Luger 2014]. Research has also reported females having lower computer self-efficacy even when objective measures show them to be just as competent at completing the task as males [Hargittai and Schafer 2006].

Our own investigations of how self-efficacy relates to gender differences with debugging and programming tools also showed that females often had lower computer self-efficacy than males. More important, their low self-efficacy was tied to the way females used software features [Burnett et al. 2011, Hartzel 2003]. For example, females' self-efficacies were predictive of their successful use of debugging tools—even though there was no predictive relationship for males [Beckwith et al. 2005, Beckwith et al. 2006]. Females were also (statistically) less willing to engage with or explore novel debugging features—despite these features being tied to improved debugging performance by both females and males [Beckwith et a. 2006, Beckwith et al. 2007]. Further, Subrahmaniyan et al. showed that females *used different features* and *used features differently* than males [Subrahmaniyan et al. 2008]. Finally, the features most conducive to females' successes were features that are not commonly found in EUSE environments [Subrahmaniyan et al. 2008].

Informed by self-efficacy theory and in light of the above empirical results, we devised two new features to add to our EUSE prototype. The previous prototype, a spreadsheet system, had a testing feature, in which users could "check off" any spreadsheet cell value that they decided was right, or "X out" any spreadsheet cell value they decided was wrong. Based on a hypothesis that the previous "it's right" and "it's wrong" checkmark and X in the spreadsheet cells might seem like a stronger statement than low self-efficacy users would feel comfortable making, we introduced "seems right maybe" and "seems wrong maybe" options to the check box widget in each spreadsheet cell (Figure 7). The intention of this change was to communicate the idea that a user didn't have to be confident about a testing decision in order to be "qualified" to make judgements [Grigoreanu et al. 2008].

The second feature we introduced to the prototype was to add strategy explanations in two formats: short video snippets and equivalent hypertext content (Figure 8). The original prototype had included only feature-oriented tooltips to explain the system's functionalities. Our addition of video snippets was informed by work with the "vicarious experience" source of self-efficacy, which is self-efficacy based on watching someone similar to the user

perform the task [Bandura 1977, Zeldin and Parajes 2000]. In each video snippet, the female debugger works on a debugging problem, and a male debugger, referring to the spreadsheet, converses with her about strategy ideas [Grigoreanu et al. 2008], ultimately ending in the female succeeding. The hyperlink contained the same information as the video and was offered as an option for those who don't learn best pictorially, or who want to quickly scan to a particular part of the information as a time saver.

Figure 7: Clicking on the checkbox turns it into four choices whose tool tips say "it's wrong," "seems wrong maybe," "seems right maybe," "it's right." Our decision to add this feature to the prototype was based on self-efficacy theory.

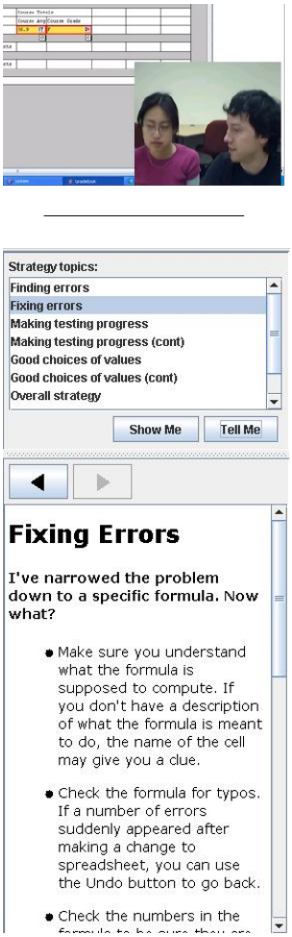

Figure 8: (Top): 1-minute video snippets. (Bottom): Hypertext version. As with Figure 7, our decision to add these features to the prototype was based on self-efficacy theory.

With the addition of these new features, both females' and males' (but especially females') usage of testing/debugging features increased, which translated into testing and debugging improvements. Females' confidence levels also improved to an appropriate level—where their self-judgments were roughly appropriate indicators of their actual ability levels. Males' confidence levels had already been appropriate indicators of their abilities, and remained so. Females' and males' (but especially females') post-session verbalizations showed that their attitudes toward the software environment were significantly more positive. Note that the gains for females came without disadvantaging the males, and, in fact, usually ended up helping them as well [Grigoreanu et al. 2008].

3.2 A GenderMag Foundation: Information Processing Theory

Another theory that has strongly influenced our gender investigations is the Selectivity Hypothesis of information processing styles [Meyers-Levy 1989, Meyers-Levey and Maheswaran 1991, Meyers-Levy and Loken 2014]. This theory proposes that males process information in a selective manner, attending to highly available and salient cues and then acting, whereas females process information in a comprehensive manner, gathering as much information from all available cues as possible to develop a full picture of the issue before acting. The theory also claims that males and females categorize and classify information into categories differently, with females creating more categories than males, and placing statements with conceptual similarity into these categories in more consistent ways than males [Meyers-Levy 1989, Meyers-Levey and Maheswaran 1991, Meyers-Levy and Loken 2014].

We hypothesized that someone's information processing style should affect the way they approach a problem in the realm of debugging. Indeed, our studies have shown this link. For example, females tended to process information comprehensively using code inspection to get a relatively complete picture of the problem. This strategy often resulted in success for female participants. On the other hand, males had more success using dataflow and following particular dependencies one at a time, which coincides with selective information processing [Grigoreanu et al. 2009, Subrahmaniyan et al. 2008].

A close-up look at the most successful female participant and the most successful male participant from one of our studies helps to illustrate these distinct information processing styles. In [Grigoreanu et al. 2012], we investigated how male and female end users made sense of spreadsheet correctness when debugging. Participants were given 45 minutes to decide whether a (flawed) spreadsheet was correct, and, if it wasn't, to fix it.

The most successful female demonstrated the comprehensive information processing style, gathering a lot of information before taking fix action(s) (stars above the graph in Figure 9), which often occurred in bursts. The lengths of her bursts of information gathering are indicated by the superimposed horizontal arrows. In all cases her fixes were successful, indicated on the graph by all the stars being filled in. This burst-y style, during which she spent a burst of time gathering a large batch of information and then acted upon that batch in a burst of activity, is consistent with a comprehensive style of information processing.

This strategy generally worked well for our female participant, but it had a downside as well. In times of uncertainty, the successful female would continue gathering information even though it did not yield results, as evident in Figure 9 where she sometimes did not emerge from the information gathering stages for long periods of time. This resulted in her noticing bugs while foraging but not marking them or fixing them at the time, so she sometimes forgot to fix them.

In contrast to the female's style, the successful male's style was a classic example of selective processing. In fact, his *maximum* time spent gathering information (about 5 minutes in the graph) was less than the successful female's *minimum* time spent on gathering information. Instead, he iterated tightly between gathering a little information and trying out a fix. As Figure 9 shows with hollow stars, many of his attempted fixes were unsuccessful, but he often realized that and went back to gather a little more information and then try again.

As with the female's use of her comprehensive processing strategy, our male participant's selective processing strategy worked well for him, but also had a downside. He fixated on one bug and focused on only that bug until it was fixed. This cost him time and also discouraged him from fixing or identifying other bugs that he came across while working on his target bug.

In total, the two participants fixed the same number of bugs. (The number of filled-in stars shows one extra for the male because he also implanted a new bug along the way that he also had to fix.) As these two participants illustrate, both styles have advantages and disadvantages, but can lead people differently to equal levels of success.

Insights into these information processing styles are then reasonably prescriptive: they suggest that an EUSE tool should support both styles. In Section 4, one example of how to do this will be seen in Table 2, in which expandable tooltips are used to support this diversity of information processing styles.

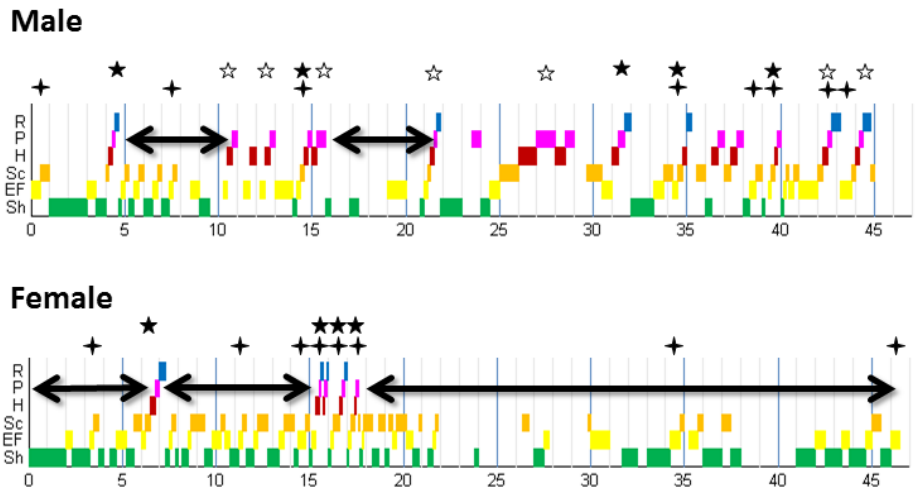

Figure 9: Two end-user developers' information processing activities while debugging a spreadsheet over about 45 minutes. The bottom three rows of each graph (green, yellow, and orange) are information gathering activities, and the top three rows (red, pink, and blue) are acting upon that information. (Top graph): The most successful male participant. (Bottom graph): The most successful female participant.

3.3 GenderMag: A Theory-to-Practice Method

We have used foundations like those mentioned above to develop a practical method for evaluating software features for gender inclusiveness issues. We call it GenderMag (Gender-Inclusiveness Magnifier) [Burnett et al. 2015].

3.3.1 The GenderMag Method

GenderMag is based on five theory-based facets of individual differences that statistically cluster according to gender. The five facets are: motivation, information processing, computer self-efficacy, risk-aversion, and tinkering. We discussed two of them (self-efficacy and information processing) above, and a discussion of all five facets can be found in [Burnett et al. 2015]. Our criteria for selecting these particular five facets were that (1) they had to be extensively researched in prior literature, (2) they had to be usable to ordinary software developers or UX designers without needing backgrounds in gender research, and (3) they had to have implications for software usage in problem-solving situations. We conducted several formative studies, a lab study, and a field study to ensure that the facets satisfied these criteria [Burnett et al. 2015, Burnett et al. 2016].

These facets are embedded in a set of four personas to bring them to life: Tim, Abby, Pat(ricia), and Pat(rick). Personas are archetypes of particular user groups of interest [Marsden 2014; Turner and Turner 2011]. Thus, in our case, each persona captures facets of individual differences that statistically cluster by gender. The Tim persona has facet values that were most frequently seen in males across numerous studies, and Abby's facet values are those *least* frequently seen in males while still occurring frequently in females. Most of Pat(ricia)'s and Pat(rick)'s facet values are "between" Tim's and Abby's values. The only difference between the two Pat personas is their gender, to reinforce the point that the road to gender inclusiveness lies not in a person's gender identity, but in the facet values themselves.

As a concrete example, Figure 10 demonstrates a portion of the data behind the Motivation facet values of each persona. As with all the facets, motivation is backed by multiple studies, but for simplicity of presentation, only one of them is illustrated in the figure [Burnett et al. 2010]. In that study, about 2/3 of males and 1/3 of females were motivated by exploring next-generation technology (covered by Tim), and about 1/5 of both males and females felt neutral about it (covered by the two Pats). The largest percentage of females and smallest percentage of males did not enjoy exploring next-generation technology (covered by Abby). Figure 11 shows the persona side of such mappings, with text snippets of how another facet, self-efficacy, maps to each persona.

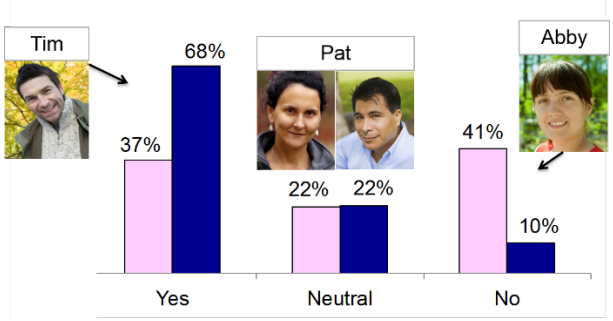

Figure 10: A portion of the empirical foundations behind the personas' Motivation facet values (see text).

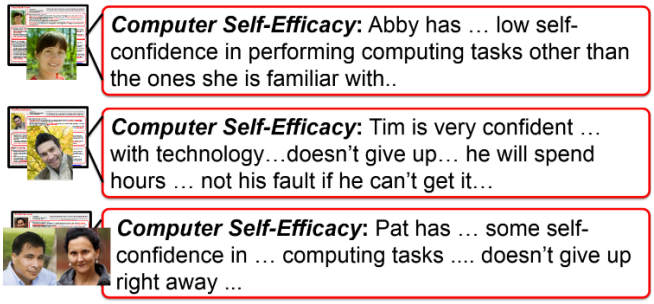

Figure 11: [Burnett et al. 2016]. The self-efficacy portions of Abby, Tim, and the two Pats, drawn from self-efficacy and empirical data (see text).

GenderMag intertwines these personas with a specialized Cognitive Walkthrough (CW). The CW is a long-standing inspection method for identifying usability issues for users new to a system or feature [Wharton et al. 1994]. We based the GenderMag CW on a streamlined version of the CW [Spencer 2000] and further streamlined from there to increase the method's practicality.

More important, we embedded mentions of the personas and the appropriate facets to consider in each CW questions. Our reason for mentioning the personas was to address the problem of evaluators thinking about the way *they* would respond to the interface, not the way the persona would respond. (Some researchers term this unfortunate tendency that evaluators sometimes have as the "I methodology".) Our reason for embedding the facets in the questions was to keep evaluators focused on the 5 facet values, instead of falling back on their own biases and stereotypes.

3.3.2 Evaluation: GenderMag in the Field

GenderMag has just started being used in the field, but it has already produced encouraging and rather surprising results.

More specifically, we recently conducted a field study of four software teams in U.S. technology organizations who used GenderMag to evaluate their own software for gender-inclusiveness issues [Burnett et al. 2016]. Their software spanned a range of types and maturity levels, and evaluation team members spanned a range of job titles, including software developers, UX researchers, computer science researchers, and software managers.

Despite these differences, the results consistently showed that all teams found gender-inclusiveness issues in their software. Most issues were those that would disproportionately affect users in the group represented by Abby, but one team also uncovered an issue that would disproportionately affect users in the group represented by Tim. (This field study occurred before the Pat personas had been released.) On average, the teams turned up gender-inclusiveness issues in 25% of the software features they evaluated (Figure 12). In many cases, these issues had gone unnoticed for months or even years.

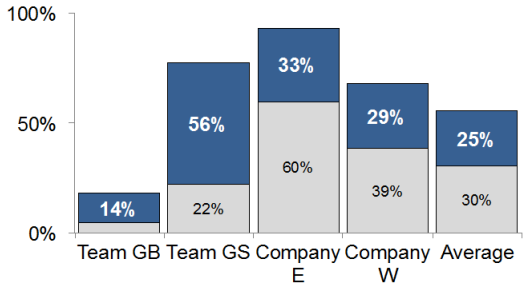

Figure 12: Issues each team found as a percentage of the number of user actions and subgoals evaluated [Burnett et al. 2016]. Above bars: total issues. Dark blue: gender-inclusiveness issues. Light gray: other issues.

All the teams found the method to be useful. Agency G's two teams (GB and GS) found four issues that they deemed important enough to pursue fixing, even though their software had been in maintenance status for years. Team E fixed 3 issues right away, and Team W convinced the software's designers to fix 3 issues. Teams GB, GS, and W also made longer-term follow-up plans involving GenderMag. Perhaps most important, all of the teams ultimately realized that for software to be gender-inclusive, it needs to support a range of facet values, not just facet values matching the designers' own personal styles. In essence, they realized that gender inclusiveness is not about sorting people into gender bins–it is about supporting the entire range of facet values [Burnett et al. 2016].

4. The Idea Garden

Our third example is the Idea Garden, a theory-based approach we developed to support end users' problem-solving during programming and debugging. Evidence abounds that, despite advances in EUSE environments and tools, users continue to encounter programming barriers. For example, Ko et al. identified six learning barriers faced by end users in using Visual Basic [Ko et al. 2004]. In the realm of spreadsheets, Chambers and Scaffidi found that spreadsheet users face barriers similar to those faced by Visual Basic learners as identified by Ko et al. [Chambers and Scaffidi 2010]. Our own research found the same barriers in other end-user programming platforms as well, such as CoScripter and Gidget [Cao et al. 2011, Lee et. al 2014].

To help end users overcome barriers like these, the Idea Garden aims to help end users generate new ideas and problem-solve in a self-directed way. This goal contrasts with approaches that aim to remove such barriers. For example, we do not seek to change the programming language to make it simpler or more natural [Myers et al. 2004; Little et al. 2007; Kelleher and Pausch 2006; Blackwell and Hague 2001], do not seek to automatically eliminate or solve problems for the user [Repenning and Ioannidou 2008; Little et al. 2007; Lin et al. 2009; Miller et al. 2010; Hartmann et al. 2010; Ennals et al. 2007], and do not seek to delegate programming responsibilities to someone other than the end users themselves [Oney and Myers 2009; Brandt et al. 2010]. Rather, we seek to help end users generate their *own* ideas to overcome the programming barriers they do encounter.

4.1 The Idea Garden's Foundations

Toward that end, we followed a theory-based approach to devise the Idea Garden. The principle theoretical foundation behind the Idea Garden is minimalist learning theory [Carroll and Rosson, 1987; Carroll, 1990; Carroll 1998; van der Meij, H. and Carroll 1998]. Rooted in the constructivism of Bruner and Piaget, minimalist learning theory (MLT) is an education theory that explains how (and why) to design instructional materials for end users: the theory terms people like this "active" computer users. Active computer users are those whose primary motivation is to *do* some computer-based task of their own, not particularly to *learn* computing skills. Active users are so focused on the task at hand that they are often unwilling to invest time in taking tutorials, reading documentation, or using other training materials—even if such an investment would be rational in the long term. Helping users who face this paradox learn *despite* their lack of interest in learning per se is the goal of MLT. This theory is especially suited to the design of the Idea Garden features because the Idea Garden aims to help end users generate new ideas while they are working on their own, self-directed programming tasks.

MLT suggests several design principles that lend themselves to effective learning activities for active users. The Idea Garden follows these guidelines in the following ways:

(MLT-1) Permit self-directed reasoning: The Idea Garden suggests strategy alternatives and provides (intentionally) incomplete hints, all of which require the user to actively reason and problem solve in order to make substantive progress on the task at hand (as opposed to simply giving them the solution). The parts that are not problem-specific are "correct" in that they apply regardless of the granular details of the user's task. The incomplete parts of hints are so specific that they are unlikely to be exactly what the user needs and must be adapted to the user's specific problem or barrier. In addition, the Idea Garden uses negotiated-style interruptions, which inform users of pending messages but do not force them to acknowledge the messages [McFarlane 2002]. This allows users to decide for themselves whether to read and consider the messages, permitting self-directed reasoning. This interruption style contrasts with assertive instructional agents that violate users' self-directedness with immediate-style interruptions, such as Microsoft Office's paperclip ("Clippy"), which hijack the user's attention [McFarlane 2002]. Negotiated-style interruptions have been shown to be superior to immediate-style interruptions in end-user debugging situations [Robertson et al. 2004].

(MLT-2) Be meaningful and self-contained and *(MLT-3) Provide realistic tasks early on:* In the Idea Garden, hints are task-oriented, and many of them are closely coupled with the user's context. To follow MLT-2, hints are generally tied to tasks that the user has already chosen to initiate with the goal of giving meaning, value, and realism. Hints also uphold MLT-3 using examples of tasks that the user is likely to be working on or may have performed in the past.

(MLT-4) Be closely linked to the actual system: The Idea Garden is a layer on top of a host environment. The Idea Garden can be a layer on any end-user programming environment that allows the Idea Garden to: retrieve the user's data and code as it appears to the user (i.e., on the screen) and as it appears to the machine (i.e., after parsing); change the user's code (e.g., by inserting constants or lines of code); and annotate the programming environment and/or user's code with interactive widgets (e.g., tooltips, buttons, graphics, or font changes). Many programming environments, including Excel, CoScripter, Gidget, and Cloud9, satisfy these constraints, providing pluggable architectures into which Idea Garden features can be added [Cao et al. 2010, Cao et al. 2011, Jernigan et al. 2015, Jernigan et al. 2017. Loksa et al. 2016].

(MLT-5) Provide for error recognition and recovery: The Idea Garden is not intended to replace the programming environment's native error recovery system, so it leaves most of the detection of and recovery from errors to the host environment itself. Rather, the Idea Garden adds onto its host programming environment, expanding whatever help systems and error supports the host provides.

Using the MLT principles as our foundation, we define the Idea Garden as [Cao et al. 2013]:

(Host) A subsystem that extends a "host" end-user programming environment to provide hints that...
(Theory) follow the 5 MLT principles (MLT-1 through MLT-5) and...
(Content/Presentation) non-authoritatively give intentionally imperfect guidance about problem-solving strategies, programming concepts, and design patterns, via negotiated interruptions.
(Implementation) In addition, the hints are presented via host-independent templates that are informed by host-dependent information about the user's task and progress.

4.2 The Idea Garden in Action

We have implemented the Idea Garden in three host programming environments: CoScripter [Cao et al. 2010, Cao et al. 2011], Gidget [Jernigan et al. 2015, Jernigan et al. 2017], and Cloud9 [Loksa et al. 2016, Jernigan et al. 2017]. Although each version looks different, all follow the Idea Garden definition and MLT guidelines we have just presented. In the next subsections, we show examples of ways the five MLT guidelines can be instantiated, using these implementations.

4.2.1 CoScripter

The first environment in which we prototyped the Idea Garden was CoScripter, an end-user programming-by-demonstration environment for web scripting [Cao et al. 2010, Cao et al. 2011, Lin et al. 2009]. Users demonstrate to CoScripter how they would carry out a task on the web by actually performing the task themselves. The system watches and translates users' actions into a script that they can later execute to perform the same task again, or can generalize to perform the task in a variety of situations. When working on these scripts, users can turn to Idea Garden hints if they get "stuck".

Recall that, to stay in accordance with MLT-1, Idea Garden hints have both "correct" and "incomplete" parts. Figure 13 demonstrates with a representative hint from the CoScripter prototype. The "correct" part suggests using a repeat

statement to iterate over cells in a column rather than addressing each cell as a separate case. The "incomplete" part gives a specific example of using this strategy that does not directly solve the user's problem, but that the user can use as a model for the correct solution.

The hint shown in Figure 13 also demonstrates how the Idea Garden supports MLT-2, MLT-3, and MLT-4 as follows. The hint example started as a "repeat" template, then plugged in URLs ("http://maps.google.com/"), cell references ("'Address' column of row 1"), and widgets ("the first button") from the user's own code. This content's concrete connections to the user's own task are to fulfill the "meaningful" aspect of MLT-2 and also the "realistic task" aspect of MLT-3. In addition, the hint's example does not require background knowledge of the user beyond common copy/paste editing operations and programming constructs they have already used, thereby adhering to the "self-contained" aspect of MLT-2.

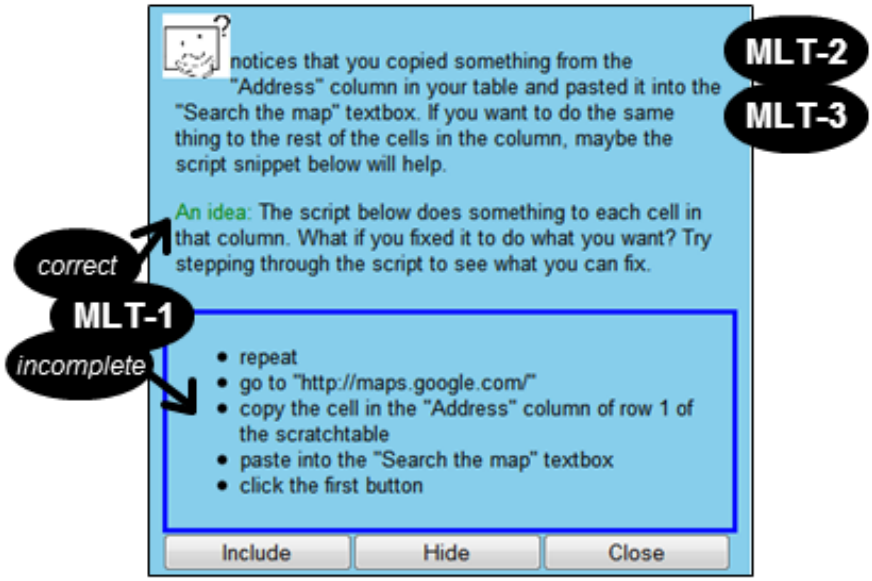

Figure 13: The hint's content is triggered if the user's scripting suggests that they are trying to repeat the same action on more than one data element, adhering to MLT-2 and MLT-3. The hint does not pop up; its availability is indicated in situ (for MLT-4) by an icon that decorates the user's script at the relevant script segment. The user can click on the icon to see this hint. To uphold MLT-1, the hint's example has incomplete parts, so that the user must actively engage with it by modifying the script to make it work. MLT-5 is not shown in this example.

4.2.2 Gidget

Gidget, an online puzzle game that centers on debugging, hosted the second version of the Idea Garden. In the game, a robot named Gidget provides players with code to complete missions. According to the game's backstory, Gidget was damaged, and the player must help Gidget by diagnosing and debugging Gidget's faulty code. Game levels introduce or reinforce different programming concepts.

Gidget's adherence to MLT-1 through MLT-4 are similar to CoScripter's, so we do not repeat them here. However, the Gidget example shown in Figure 14 demonstrates particularly well its adherence to MLT-5, the Idea Garden's integration with the host environment's built-in error identification and recovery systems. The superimposed callouts ("grab /piglet/" and "Oh no! ...") show the way the host environment points out and helps with a logic error in the user's code. Figure 14 shows Gidget handling the same error while the Idea Garden augments the environment with a tooltip based on the user's code. By acting as a layer on top of the native environment rather than a full replacement of its help system, the Idea Garden allows users to fully use Gidget's built-in error handling.

Figure 14: How MLT-5 is supported in Gidget. The ? (in the code pane, upper left) and the tooltip-like hint (lower left) are the Idea Garden's error recognition and recovery mechanisms, which supplement Gidget's built-in error/recovery features (outlined with pink boxes in the upper left ("grab /piglet/") and lower right ("Oh no! …")). The superimposed callouts are for readability.

4.2.3 Cloud9

The third host environment for the Idea Garden was Cloud9, a web-based IDE. We used this version of the Idea Garden in two 2-week high school web development camps that taught HTML and JavaScript, as a resource for students to turn to when they were stuck [Loksa et al. 2016]. The Idea Garden runs within Cloud9 in the form of a plugin. The pluggable architecture of the environment gave us access to Cloud9's background processes, such as its parser, tokenizer, and listener, which allowed us to directly inspect and handle the user's code, as was also the case in the other two environments.

The Cloud9 implementation of the Idea Garden adheres to the MLT principles in ways similar to the other two host environments, but its appearance is different. Figure 15 shows the Idea Garden as it appears in the expandable panel within the programming environment, designed to look and act similarly to the rest of Cloud9's expandable panels for MLT-4. Figure 16 shows how the use of the negotiated interruption style to inform users of potentially useful hints by decorating the environment with a ? icon, as per MLT-1.

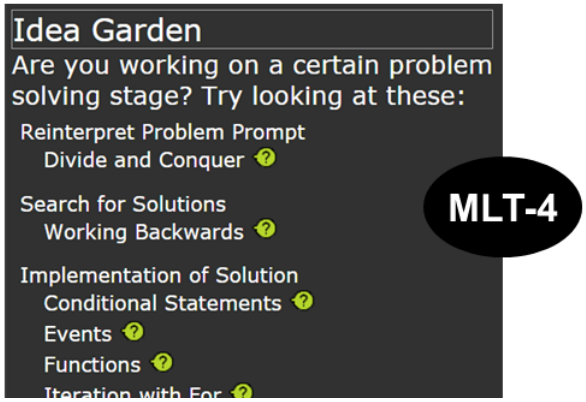

Figure 15: The top portion of the Idea Garden panel in the Cloud9 IDE. To uphold MLT-4, the panel mimics the functionality of Cloud9's other expandable panels.

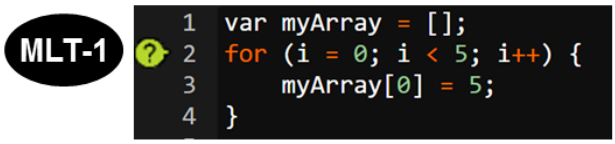

Figure 16: An example of the Idea Garden decorating the code with an icon. Here, the icon links to the *Iteration with For* hint if the user decides to click on it.

4.3 The Idea Garden's Principles

As we empirically investigated these instantiations of the Idea Garden with end users, we found that the *Content/Presentation* aspect of the definition (Section 4.1)—arguably the most important of the four aspects—was difficult to get right. The collection of MLT principles MLT-1 though MLT-5 were too high-level to adequately address this problem. We needed a lower-level set of principles that would expand upon MLT's guidelines while still being implementation-independent enough to support future Idea Gardens.

To address this need, we derived seven lower-level principles using prior Idea Garden studies and the theoretical foundations of Section 4.1 to better ground the Content/Presentation aspect [Jernigan et al. 2015], which we enumerate in Table 1. As we discussed in the introduction, a theory can explain, predict, or prescribe behaviors, and theory can be both consumed and produced. Here, we *consume* prescriptions and predictions from MLT and related theories, and *produce* our own prescriptive set of principles on how a system like the Idea Garden might be most effective.

Most of the principles build upon MLT's points as to how to serve active users. For instance, IG1-Content provides content that relates to what the user is *already doing*; IG2-Relevance displays content in a way that the user believes it to be *relevant to the task at hand*; IG3-Actionable gives the user something to *do* with the newfound information; and IG6-Availability ensures that users can access content within the context in which they are working to *keep their focus on getting the task done* rather than searching for solutions from external sources. Figures 17 and 18 and Table 2 show concrete examples of how we implemented the Gidget-based Idea Garden according to these principles.

Principle	Explanation
IG1-Content	Content that makes up the hints need to contain at least one of the following:
IG1.Concepts	Explains a programming *concept* such as iteration or functions. Can include programming constructs as needed to illustrate the concept.
IG1.Mini-patterns	*Design mini-patterns* show a usage of the concept that the user must adapt to their problem (minipattern should not solve the user's problem).
IG1.Strategies	A problem-solving strategy such as working through the problem backward.
IG2-Relevance	For Idea Garden hints that are context-sensitive, the aim is that the user perceives them to be relevant. Thus, hints use one or more of these types of relevance:
IG2.MyCode	The hint includes some of the user's code.
IG2.MyState	The hint depends on the user's code, such as by explaining a concept present in the user's code.
IG2.MyGoal	The hint depends on the requirements the user is working on, such as referring to associated test cases or pre/post-conditions.
IG3-Actionable	Because the Idea Garden targets MLT's "active users," hints must give them something to *do* [Carroll and Rosson 1987, Carroll 1990]. Thus, Idea Garden hints must imply an action that the user can take to overcome a barrier or get ideas on how to meet their goals:
IG3.Explictly Actionable	The hint prescribes actions that can be physically done, such as indenting or typing something.
IG3.Implicitly Actionable	The hint prescribes actions that are "in the head," such as "compare" or "recall".
IG4-Personality	The personality and tone of Idea Garden entries must try to encourage constructive thinking. Toward this end, hints are expressed non-authoritatively and tentatively [Lee and Ko 2011]. For example, phrases like "try something like this" are intended to show that, while knowledgeable, the Idea Garden is not sure how to solve the user's exact problem.
IG5-Information Processing	Because research has shown that (statistically) females tend to gather information comprehensively when problem-solving, whereas males gather information selectively [Meyers-Levy 1989], the hints must support both styles. For example, when a hint is not small, a condensed version must be offered with expandable parts.
IG6-Availability	Hints must be available in these ways:
IG6.Context Sensitive	Available in the context where the system deems the hint relevant.
IG6.ContextFree	Available in context-free form through an always-available widget (e.g., pull-down menu).
IG7-Interruption Style	Because research has shown the superiority of the negotiated style of interruptions in debugging situations [Robertson et al. 2004], all hints must follow this style. In negotiated style, nothing ever pops up. Instead, a small indicator "decorates" the environment (like the incoming mail count on an email icon) to let the user know where the Idea Garden has relevant information. Users can then request to see the new information by hovering or clicking on the indicator.

Table 1: The seven Idea Garden Principles and their explanations. The middle column contains the work that helped inform each Idea Garden principle. A hyphenated name signifies a principle (e.g. IG1-Content), while a name with a dot signifies a subprinciple (IG1.Concepts).

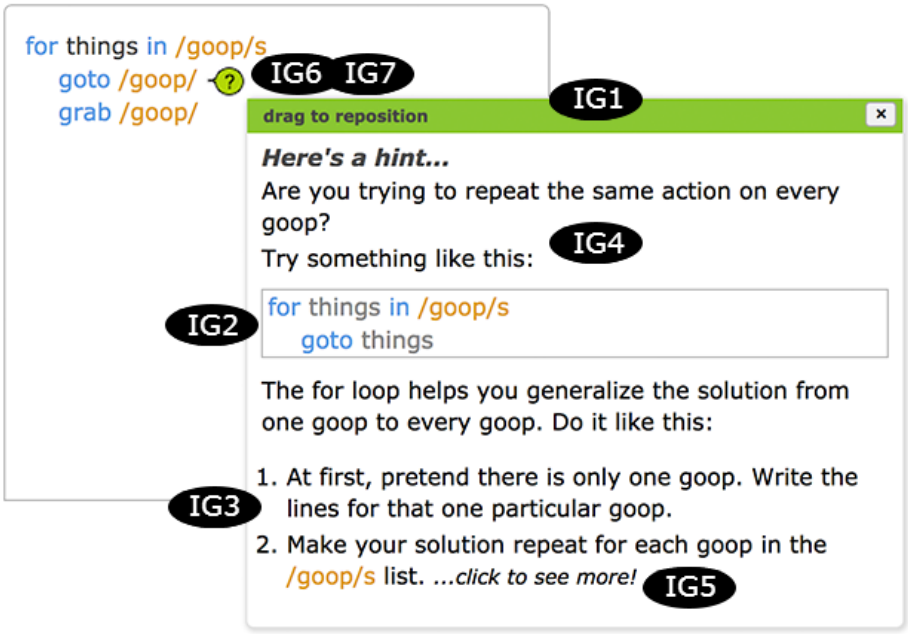

Figure 17: An example of a context-sensitive Idea Garden hint in Gidget. Hovering over the ? shows the hint, and the superimposed black ovals show where the 7 principles are instantiated in the hint.

Figure 18: As part of IG6 and IG7, the Idea Garden's hints appear as tooltips when the user hovers over a ?. The superimposed callouts are for readability.

Detecting *antipatterns* enables support for two of the Idea Garden principles. Antipatterns, a notion similar to "code smells", are implementation patterns that suggest some kind of conceptual, problem-solving, or strategy difficulty,

such as not using an iterator variable within the body of a loop or defining a function without calling it. We used antipatterns in all three Idea Garden implementations. The examples shown here are from the Gidget version.

Antipatterns define context for IG6.ContextSensitive, letting the hint be derived from and shown in the context of the user's problem. For IG2-Relevance, the hint communicates relevance (to the user's current problem) by being derived from the player's current code as soon as they enter it, such as using the same variable names (Figure 17, IG2 and IG6). The Gidget-based Idea Garden brings these two principles together by constructing a context-sensitive hint whenever it detects a conceptual antipattern. The Idea Garden then displays the hint indicator ( in the Gidget-based version) beside the relevant code to show the hint's availability.

Principle	Which part(s) of Idea Garden affected	Example in Gidget
IG1-Content	Body of hint: Code example (drawn from *antipattern*); strategy	Figure 18, large callout; Figure 17, middle
IG2-Relevance	Gist of hint ("Are you trying to…?"); Inclusion of user's own code and variable names	Figure 17, gist and code example
IG3-Actionable	Suggested strategy in hint, using actionable words like "pretend" and "write"	Figure 17, numbered steps
IG4-Personality	Tentative phrasing ("You might try.../...Try something like this")	Figure 17, text
IG5-Information Processing	Expandable strategies in longer hints	Figure 17, "Click to see more"
IG6-Availability	In-context availability through antipattern-triggered icons; Context-free availability via a UI widget that always gives access to all hints	Figure 17,  ; Figure 18, small callout of dictionary button
IG7-Interruption Style	Icon users can *choose* to interact with rather than hijacking control to display a hint	Figure 17, 

Table 2: How the seven Idea Garden principles were implemented, with examples in Gidget.

4.4 Evaluation

We conducted an empirical study with teams of teenaged students to evaluate these principles [Jernigan et al. 2015] as instantiated in Gidget. Teams worked through Gidget's levels, and, if they had time, designed their own levels using the built-in editor. As it turned out, teams did not always need the Idea Garden; they solved many of their problems just by discussing them with each other, reading the reference manual, etc. However, when these measures did not suffice, they turned to the Idea Garden for more assistance.

As Table 3 shows, the Idea Garden enabled campers to problem-solve their way past the majority of these barriers (52%) without any guidance from the helper staff. Particularly noteworthy is the fact that their problem-solving progress with the Idea Garden alone (52% solved their problem using the Idea Garden alone, *without* in-person help from camp helpers) was almost as high as their progress with in-person help (59% solved their problem using only in-person help from camp helpers, *without* the Idea Garden).

IG Available and On-screen?	Progress without in-person help	Progress if team got in-person help	No evidence of progress
Yes (149+25 instances)	77/149 (52%)	25/25	72/(149+25)
No (155 instances)	53/155	91/155 (59%)	11/155

Table 3: Barrier instances and teams' progress with/without getting in-person help. Teams did not usually need in-person help when an Idea Garden hint and/or an antipattern-triggered was on the screen (top row). The third column, containing instances of barriers where teams did not make progress, is shown to ensure the rows sum to the correct values.

Further, the results suggested that each principle complemented the others in coming together to support a diversity of problems, information processing and problem-solving styles, cognitive stages, and people.

For example, teams' successes across a diversity of concepts served to demonstrate this point while also validating the concept aspect of IG1-Content. Antipatterns were especially involved in teams' success rates with different kinds of barriers. Together, these aspects enabled the teams to overcome, without any in-person help, 41%-68% of the barriers they encountered across *diverse barrier types.* Adding to this, teams could overcome these barriers in diverse ways by IG2-Relevance and IG6-Availability working together to provide relevant, just-in-time hints to afford teams diverse paths by which to use the to make progress. This suggests that IG2 and IG6 help support *diverse EUP problem-solving styles*.

IG3-Actionable's explicit vs. implicit approaches also played to different strengths. Teams tended to use explicitly actionable instructions (e.g., "Indent...") to translate an idea into code, at the Bloom's Taxonomy of Educational Objectives "apply" stage (using information in new situations) [Anderson et al. 2001]. In contrast, teams seemed to follow implicitly actionable instructions more conceptually and strategically ("recall how you..."), as with Bloom's "analyze" stage (drawing connections among different ideas). This suggests that the two aspects of IG3 can help support EUPs' learning across *diverse cognitive process stages*. Finally, IG5-InformationProcessing requires the Idea Garden to support both the comprehensive and selective information processing styles, as per previous research on gender differences in information processing styles. The teams used both of these styles, mostly aligning by gender with the previous research, implying that following IG5 helps support *diverse EUP information processing styles*.

Much of the success of the Idea Garden principles can be ascribed to the theories upon which they are grounded. Minimalist learning theory (MLT) defined our intended audience of active users and prescribed ways in which we could target them with the Idea Garden's resources. The guidelines laid down by MLT gave us a strong foundation from which to begin the research. Past successes with MLT-based systems also predicted our success with the early versions of the Idea Garden.

When it became clear that we needed to develop lower-level principles tailored to the Idea Garden, we again turned to theory to help us build some theory of our own. By beginning with the MLT foundations that had served us so well and then expanding to encompass theory from the realms of cognition and information processing styles, we were able to create a specialized set of principles to support a diverse range of problem solving styles and situations. For instance, Meyers-Levy's theories on information processing styles [Meyers-Levy 1989] informed the way that we structured Idea Garden hints in order to benefit both selective and comprehensive learners.

Most important, however, basing the Idea Garden in a principled way provided generality, which we evaluated by implementing the system for three different EUSE environments and evaluating each [Cao et al. 2013, Jernigan et al. 2015, Jernigan et al. 2017, Loksa et al. 2016]. These general principles then contribute to the theory pool to further research in end user problem solving and debugging. We hope that others in the EUSE research community will be able to draw from these principles as they develop new systems to support end users' problem solving endeavors.

5. Discussion and Concluding Remarks

This chapter has advocated for researchers in the EUSE research community to increase their connections to theory, both as consumers and producers. We see this approach as a complement to the many other EUSE design methods in our community, with each method having its own unique advantages and disadvantages.

One example of an atheoretic approach is the idea-tool-summative sequence, in which a researcher gains inspiration from some kind of problem they have seen or learned about in the literature, devises an approach to solve that problem, and evaluates its effectiveness empirically. Its strength lies in showing that the new tool works better than some comparative approach. However, its weakness is that it cannot easily get from the success of the tool to underlying causes of why exactly it worked better. As Ruthruff et al. showed, the underlying causes might be quite surprising to a researcher; for example, his study of alternative EUSE fault localization tools showed that, although one might have assumed that differences in effectiveness were due to differences in the algorithms' reasoning approaches, they actually had more to do with nuanced differences in the user interface than algorithmic differences [Ruthruff et al. 2006].

Other approaches emphasize theory building over theory consuming. For approaches following this philosophy, research begins with formative empirical work to understand the detailed needs of users in a particular domain, then builds new theories based upon the results of this research. Grounded theory work is an example of this kind of approach (e.g., [Stol et al. 2016]). The theories derived from this method are intended to be consumed in later tool building phases (whether by those researchers or by other researchers). The outcome of this approach is a highly detailed understanding of the particular domain. However, the resulting theories are often not particularly parsimonious or cohesive; perhaps for that reason, it is often challenging for researchers other than the original investigators to then consume the theories built by this method.

Practice-based computing [Wulf et al. 2015] adds an emphasis on the contextual and social aspects of technology usage in practice. This method may or may not result in new theories, but its main emphasis is not theory. Rather, its distinguishing characteristic from most other approaches is its emphasis on in-the-wild settings for knowledge discovery. This method can reveal social aspects of technology appropriation and usage that many other methods may miss; but its findings may be so situated in the system(s) studied that reusing its findings in other types of systems can be challenging.

Although these other approaches all have advantages, we argue that theory consumption should be brought into the picture much more often and that it is currently underutilized in EUSE research. The essence of theories is abstraction—mapping instances of successful approaches to crosscutting principles. In the realm of human behavior, these abstractions can produce explanations of why some software efficiently and effectively supports people's work and other software does not. In a recent lecture, Herbsleb eloquently explained one example when he pointed out that certain theories of how people self-organize in groups reflect processes that have evolved in humans over thousands of years [Herbsleb 2016]. Rather than ignore these theories, it makes sense to leverage them (i.e., become good consumers of them) in designing our own tools.

Another example is information foraging theory (IFT), a theory devised to explain and predict how people seek information [Pirolli 2007]. The basic building blocks are that "predators" (people seeking information) look for "prey" (information relevant to their goals) through "patches" of information (such as different files) that contains "cues" (such as clickable links) about where to find the information they want. Such a simple set of concepts can be remarkably powerful when applied to software tools. For example, Fleming et al. used IFT to reveal commonalities among different sub-branches of software engineering research that could be leveraged toward faster progress [Fleming et al. 2013], and Piorkowski et al. used IFT to reveal a kind of "lower bound" in software tool usage that determines programmers' minimum costs in using a software tool [Piorkowski et al. 2016].

Theory likewise brought a number of advantages to the projects we presented in this chapter. In Explanatory Debugging, the mental model theory of reasoning's notion of "runnable" mental models gave us a deep understanding of how Explanatory Debugging can effectively support the actionability necessary for debugging. This, in turn, led to the EluciDebug prototype and a set of general principles for Explanatory Debugging approaches. In the GenderMag work, the attention investment model, self-efficacy theory, and information processing theory gave us the ability to make reasonable predictions about what diverse users might attend to and follow through with (or not) and why. Because these theories went deeper than gender itself, they provided a basis for GenderMag to help software

developers improve the inclusiveness of the software they build for a diversity of users – without the need to navigate the political waters of talking about one gender versus another. Finally, in the Idea Garden project, minimalist learning theory provided us a number of prescriptive ideas on how to not only enable, but also entice, untrained end users to incrementally succeed when faced with software development misbehaviors, difficulties, and barriers.

These theory-grounded projects demonstrate the idea that a EUSE research project can be both theory-informed and theory-producing – allowing EUSE researchers to participate in both the verification and creation of theories that can, in turn, further inform EUSE. As examples like these show, theories enable us to go beyond instances of successful approaches to cross-cutting principles and methods that can be used across a breadth of situations. Further, theories can give us ways to connect multiple research efforts, pattern-matching existing projects to a common theory to evolve to a deeper understanding of the domain. In essence, theories can provide powerful means by which EUSE researchers can better "stand on the shoulders" of the researchers who came before them.

Acknowledgments

We thank our students and collaborators who contributed to our work, all the participants of our empirical studies, and the reviewers for their helpful suggestions. Our work in developing this chapter was supported in part by the National Science Foundation under grants CNS-1240957, IIS-1314384, and IIS-1528061, and by the DARPA Explainable AI (XAI) program grant DARPA-16-53-XAI-FP-043. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the sponsors.

References

[Aho et al. 2006] Aho, A., Lam, M., Sethi, R., and Ullman, J. (2006) *Compilers: Principles, Techniques & Tools*. Addison Wesley.

[Anderson et al. 2001] Anderson, L. (Ed.), Krathwohl, D. (Ed.), Airasian, P., Cruikshank, K., Mayer, R., Pintrich, P., Raths, J., Wittrock, M. (2001) *A Taxonomy for Learning, Teaching, and Assessing: A revision of Bloom's Taxonomy of Educational Objectives (Complete edition)*. Longman.

[Appel et al. 2011] Appel, M., Kronberger, N. and Aronson, J. (2011) Stereotype threat impair ability building: Effects on test preparation among women in science and technology. *European Journal of Social Psychology,* 41(7), 904-913.

[Bandura 1977] Bandura, A. (1977) Self-efficacy: Toward a unifying theory of behavioral change. *Psychological Review* 8(2), 191-215.

[Bandura 1986] Bandura, A. (1986) *Social Foundations of Thought and Action*. Prentice Hall, Englewood Cliffs, NJ, USA.

[Beckwith et al. 2005] Beckwith, L., Sorte S., Burnett, M., Wiedenback, S., Chintakovid, T., and Cook C., (2005) Designing features for both genders in end-user programming environments, *IEEE Symp. VLHCC*, 153-160.

[Beckwith et al. 2006] Beckwith, L., Burnett, M., Wiedenbeck, S., Grigoreanua, V., and Wiedenbeck, S., (2006) Gender HCI: What about the software? *Computer*, Nov. 2006), 83-87.

[Beckwith et al. 2007] Beckwith, L., Inman, D., Rector, K., and Burnett, M. (2007) On to the real world: Gender and self-efficacy in Excel. *IEEE Symp. Visual Languages and Human-Centric Computing*, 119-126.

[Beyer et al. 2003] Beyer, S., Rynes, K., Perrault, J., Hay, K. and Haller, S. (2003) Gender differences in computer science students. *SIGCSE: Special Interest Group on Computer Science Education*. ACM, 49-53.

[Blackwell 2002] Blackwell, A.F. (2002). First steps in programming: A rationale for attention investment models. In IEEE *VL/HCC*, 2–10.

[Blackwell and Hague 2001] Blackwell, A., and Hague, R. (2001). AutoHAN: An architecture for programming the home. *IEEE Symp. Human-Centric Computing Languages and Environments* (pp. 150-157). Stresa: IEEE.

[Brandt et al. 2010] Brandt, J., Dontcheva, M., Weskamp, M., and Klemmer, S. (2010). Example-centric programming: Integrating web search into the programming environment. *ACM Conference on Human Factors in Computing Systems*, 513-522.

[Bunt et al. 2012] Bunt, A., Lount, M. and Lauzon, C. 2012. Are explanations always important? A study of deployed, low-cost intelligent interactive systems. *ACM IUI*, 169–178.

[Burnett et al. 2010] Burnett, M., Fleming, S., Iqbal, S., Venolia, G., Rajaram, V., Farooq, U., Grigoreanu, V. and Czerwinski, M. (2010). Gender differences and programming environments: Across programming populations. *ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. 10 pages. http://doi.acm.org/10.1145/1852786.1852824

[Burnett and Myers 2014] Burnett, M. and Myers, B. (2014) Future of end-user software engineering: Beyond the silos, *ACM/IEEE International Conference on Software Engineering: Future of Software Engineering track (ICSE Companion Proceedings)*, 201-211.

[Burnett et al. 2011] Burnett, M., Beckwith, L., Wiedenbeck, S., Fleming, S. D., Cao, J., Park, T. H., Grigoreanu, V., Rector, K. (2011) Gender pluralism in problem-solving software, *Interacting with Computers* 23, 450-460.

[Burnett et al. 2015] Burnett, M., Stumpf, S., Macbeth, J., Makri, S., Beckwith, L., Kwan, I., Peters, A., Jernigan, W. (2015) GenderMag: A method for evaluating software's gender inclusiveness, *Interacting with Computers*. doi:10.1093/iwc/iwv046

[Burnett et al. 2016] Burnett, M., Peters, A., Hill, C. and Elarief, N. (2016) Finding gender-inclusiveness software issues with GenderMag: A field investigation, *ACM Conference on Human Factors in Computing Systems* (CHI).

[Cao et al. 2010] Cao, J., Rector, K., Park, T., Fleming, S., Burnett, M., and Wiedenbeck, S. (2010b) A debugging perspective on end-user mashup programming. *IEEE Symposium on Visual Languages and Human-Centric Computing,* 149-156.

[Cao et al. 2011] Cao, J., Fleming, S., and Burnett, M. (2011) An exploration of design opportunities for "gardening" end-user programmers' ideas. *IEEE Symposium on Visual Languages and Human-Centric Computing*, 35-42.

[Cao et al. 2013] Cao, J., Fleming, S., Burnett, M., and Scaffidi, C. (2014) Idea Garden: Situated support for problem solving by end-user programmers. *Interacting with Computers*. (21 pages)

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

[Carroll and Rosson 1987] Carroll, J. and Rosson, C. (1987) The paradox of the active user. *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*, MIT Press, 26-28.

[Carroll 1998] Carroll, J. (editor) (1998) *Minimalism Beyond the Nurnberg Funnel*. MIT Press.

[Chambers and Scaffidi 2010] Chambers, C., and Scaffidi, C. (2010). Struggling to excel: A field study of challenges faced by spreadsheet users. *IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 187-194). Pittsburg, USA: IEEE.

[Compeau and Higgins 1995] Compeau, D. and Higgins, C. (1995) Application of social cognitive theory to training for computer skills. *Information System Research*, 6(2), 118-143.

[Craven and Shavlik 1997] Craven, M.W. and Shavlik, J.W. (1997) Using neural networks for data mining. *Future Generation Computer Systems*. 13, (Nov. 1997), 211–229.

[deSouza 2017] deSouza, C., (2017) Semiotic Engineering: A cohering theory to connect EUD with HCI, CMC and more, *New Perspectives in End-User Development* (F. Paterno and V. Wulf, eds.), Springer, (to appear).

[Ennals et al. 2007] Ennals, R., Brewer, E., Garofalakis, M., Shadle, M., and Gandhi, P. (2007). Intel Mash Maker: Join the web. SIGMOD Record, 36(4), 27–33.

[Fleming et al. 2013] Fleming, S., Scaffidi, C., Piorkowski, D., Burnett, M., Bellamy, R., Lawrance, J., and Kwan, I. (2013). An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks. *ACM Trans. Soft. Engr. and Method. (TOSEM)*, 22(2), 14:1–14:41.

[Gregor 2006] Gregor, S. (2006) The nature of theory in information systems, *MIS Quarterly*, 30(3): 611–642.

[Grigoreanu et al. 2008] Grigoreanu, V., Cao, J., Kulesza, T., Bogart, C., Rector, K., Burnett, M., and Wiedenbeck, S. (2008) Can feature design reduce the gender gap in end-user software development environments? *IEEE Symposium on Visual Languages and Human-Centric Computing*, 149-156.

[Grigoreanu et al. 2009] Grigoreanu, V., Brundage, J., Bahna, E., Burnett, M., ElRif, P. and Snover, J. (2009) Males' and females' script debugging strategies. *Symposium on End-User Development*, Springer.

[Grigoreanu et al. 2012] Grigoreanu, V., Burnett, M., Wiedenbeck, S., Cao, J., Rector, K. and Kwan, I. (2012) End-user debugging strategies: A sensemaking perspective. *Transactions on Computer-Human Interaction* 19, 1, ACM.

[Hargittai and Shafer 2006] Hargittai, E. and Shafer, S. (2006) Differences in actual and perceived online skills: The role of gender. *Social Science Quarterly* 87(2), 432-448.

[Hartmann et al. 2010] Hartmann, B., MacDougall, D., Brandt, J., and S. Klemmer, S. (2010). What would other programmers do: Suggesting solutions to error messages. *ACM Conference on Human Factors in Computing Systems,* 1019–1028.

[Hartzel 2003] Hartzel, K. (2003) How self-efficacy and gender issues affect software adoption and use. Comm. ACM 46, 9, 167-171.

[Herbsleb 2016] Herbsleb, J. (2016) Building a socio-technical theory of coordination: Why and how, ACM Symp. Foundations of Software Engineering, 2-10.

[Huffman et al. 2013] Huffman, A., Whetton, J. and Huffman, W. (2013) Using technology in higher education: The influence of gender roles on technology self-efficacy. Computers in Human Behavior 29, 4, 1779-1786.
[Jernigan et al. 2015] Jernigan, W., Horvath, A., Lee, M., Burnett, M., Cuilty, T., Kuttal, S., Peters, A., Kwan, I., Bahmani, F., Ko. A. (2015). A principled evaluation for a principled Idea Garden. In Proceedings IEEE Visual Languages and Human-Centric Computing (VL/HCC '15).
[Jernigan et al. 2017] Jernigan, W., Horvath, A, Lee, M., Burnett, M., Cuilty, T., Kuttal, S., Peters, A., Kwan, I., Bahmani, F., Ko, A., Mendez, C., Oleson, A. (2017) General principles for a generalized Idea Garden, *Journal of Visual Languages and Computing*, (to appear).
[Kelleher and Pausch 2006] Kelleher, C., and Pausch, R. (2006). Lessons learned from designing a programming system to support middle school girls creating animated stories. Symposium on Visual Languaes and Human-Centric Computing (pp. 165-172). Brighton: IEEE.
[Ko et al. 2011] Ko, A., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., Rosson, M., Rothermel, G., Shaw, M., and Wiedenbeck, S. (2011) The state of the art in end-user software engineering, *ACM Computing Surveys* 43(3), Article 21, 44 pages.
[Ko and Myers 2004], Ko, A. and Myers, B. (2004) Designing the Whyline: A debugging interface for asking questions about program behavior, *ACM Conference on Human Factors in Computing Systems*, 151-158.
[Ko et al. 2004] Ko, A., Myers, B., and Aung, H. (2004). Six learning barriers in end-user programming systems. *IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 199-206.
[Kulesza et al. 2010] Kulesza, T., Stumpf, S., Burnett, M. M., Wong, W.-K., Riche, Y., Moore, T., Oberst, I., Shinsel, A., and McIntosh, K. (2010) Explanatory debugging: Supporting end-user debugging of machine-learned programs. *IEEE Symposium on Visual Languages and Human-Centric Computing*, 41–48.
[Kulesza et al. 2011] Kulesza, T., Stumpf, S., Wong, W.-K., Burnett, M.M., Perona, S., Ko, A.J. and Oberst, I. (2011). Why-oriented end-user debugging of naive Bayes text classification. *ACM Transactions on Interactive Intelligent Systems*. 1, 1.
[Kulesza et al. 2012] Kulesza, T., Stumpf, S., Burnett, M.M. and Kwan, I. (2012). Tell me more? The effects of mental model soundness on personalizing an intelligent agent, *ACM CHI*, 1–10.
[Kulesza et al. 2013] Kulesza, T., Stumpf, S., Burnett, M.M. and Yang, S. (2013). Too much, too little, or just right? Ways explanations impact end users' mental models. *IEEE Symposium on Visual Languages and Human-Centric Computing*. (2013), 3–10.
[Kulesza et al. 2015] Kulesza, T., Burnett, M.M., Wong, W.-K.,Stumpf, S. (2015). Principles of explanatory debugging to personalize interactive machine learning, *ACM Conference on Intelligent User Interfaces*, 126-137.
[Lacave and Diez 2002] Lacave, C. and Díez, F.J. (2002). A review of explanation methods for Bayesian networks. *The Knowledge Engineering Review*. 17, 2, 107–127.
[Lee and Ko 2011] Lee, M. and Ko, A. (2011) Personifying programming tool feedback improves novice programmers' learning. In Proc. *ICER*, ACM Press, 109-116.
[Lieberman et al. 2006] Lieberman, H., Paterno, F., and Wulf, V., Eds. (2006) *End-User Development*. Kluwer/ Springer.
[Lim and Dey 2009] Lim, B.Y. and Dey, A.K. (2009). Assessing demand for intelligibility in context-aware applications. (Sep. 2009), 195–204.
[Lin et al. 2009] Lin, J., Wong, J., Nichols, J., Cypher, A., and Lau, T. (2009). End-user programming of mashups with Vegemite. *ACM International Conference on Intelligent User Interfaces*, 97–106).
[Little et al. 2007] Little, G., Lau, T., Cypher, A., Lin, J., Haber, D., and Kandogan, E. (2007). Koala: Capture, share, automate, personalize business processes on the web. *ACM Conference on Human Factors in Computing Systems*, 943-946.
[Loksa et al. 2016] Loksa, D., Ko, A.J., Jernigan, W., Oleson, A., Mendez, C.J., and Burnett, M. (2016) Programming, problem solving, and self-awareness: Effects of explicit guidance, *ACM Conference on Human Factors in Computing Systems (CHI)*.
[Luger 2014] Luger, E. (2014) A design for life: Recognizing the gendered politics affecting product design, In *CHI Workshop: Perspectives on Gender and Product Design*. https://www.sites.google.com/site/technologydesignperspectives/papers.
[Marsden 2014] Marsden, N. (2014) *CHI 2014 Workshop on Perspectives on Gender and Product Design*. https://www.sites.google.com/site/technologydesignperspectives/papers.
[McFarlane 2002] McFarlane, D. (2002) Comparison of four primary methods for coordinating the interruption of people in human-computer interaction. *Human-Computer Interaction*, 17, 1, 63-139.

[Meyers-Levy 1989] Meyers-Levy, J. (1989) Gender differences in information processing: A selectivity interpretation, In P. Cafferata and A. Tubout (eds.), *Cognitive and Affective Responses to Advertising*, Lexington Books, 1989.
[Meyers-Levy and Loken 2014] Meyers-Levy, J. and Loken, B. (2014) Revisiting gender differences: What we know and what lies ahead. *Journal of Consumer Psychology.*
[Meyers-Levy and Maheswaran 1991] Meyers-Levy, J. and Maheswaran, D. (1991) Exploing differences in males' and females' processing strategies. *Journal of Consumer Research* 18, 63-70.
[Miller et al. 2010] Miller, R., Bolin, M., Chilton, L., Little, G., Webber, M., and Yu, C.-H. (2010). Rewriting the web with chickenfoot. In A. Cypher, M. Dontcheva, T. Lau, and J. Nichols, In *No Code Required: Giving Users Tools to Transform the Web*, Morgan Kaufmann, 39-63.
[Myers et al. 2004] Myers, B. A., Pane, J. F., and Ko, A. (2004). Natural programming languages and environments. *Communications of the ACM*, 47(9), 47-52.
[Norman 2002] Norman, D. A. (2002) *The Design of Everyday Things. Revised and Expanded Edition.* Basic Books.
[Oney and Myers 2009] Oney, S., and Myers, B. (2009). FireCrystal: Understanding interactive behaviors in dynamic web pages. *IEEE Symposium on Visual Languages and Human-Centric Computing*, 105-108.
[Piorkowski et al. 2016] Piorkowski, D., Henley, A., Nabi, T., Fleming, S., Scaffidi, C., and Burnett, M. (2016) Foraging and navigations, fundamentally: Developers' predictions of value and cost, *ACM Symp. Foundations of Software Engineering*, 97-108.
[Pirolli 2007] Pirolli, P. (2007). *Information Foraging Theory: Adaptive Interaction with Information*. Oxford University Press.
[Repenning and Ioannidou 2008] Repenning, A., and Ioannidou, A. (2008). Broadening participation through scalable game design. International Conference on Software Engineering (pp. 305–309). Leipzig: ACM.
[Robertson et al. 2004] Robertson, T., Prabhakararao, S., Burnett, M., Cook, C., Ruthruff, J., Beckwith, L., and Phalgune, A. (2004) Impact of interruption style on end-user debugging. *ACM Conference on Human Factors in Computing Systems* (CHI), 287-294.
[Rowe 1973] Rowe, M. B. (1973) *Teaching Science as Continuous Inquiry*. McGraw-Hill.
[Ruthruff et al. 2006] Ruthruff, J., Burnett, M., and Rothermel, G. (2006) Interactive fault localization techniques in a spreadsheet environment, *IEEE Transactions on Software Engineering*, 2(4), 213-239.
[Shaw 1990] Shaw, M., (1990) Prospects for an engineering discipline of software, *IEEE Software*, 15-24.
[Shneiderman 1995] Shneiderman, B. (1995) Looking for the bright side of user interface agents, *ACM Interactions*. 2(1). pp 13-15. January.
[Sjoberg et al. 2008] Sjøberg,, D., Dybå, T. , Anda, B., and Hannay, J. (2008) Building theories in software engineering, In *Guide to Advanced Empirical Software Engineering* (Forrest Shull, Janice Singer, Dag I.K. Sjøberg, Editors), Springer, 312-336.
[Stol et al. 2016] Stol, K., Ralph, P., Fitzgerald, B. (2016) Grounded theory in Software Engineering research: A critical review and guidelines, *ACM/IEEE International Conference on Software Engineering*, 120-131.
[Stumpf et al. 2009] Stumpf, S., Rajaram, V., Li, L., Wong, W.-K., Burnett, M.M., Dietterich, T., Sullivan, E. and Herlocker, J. (2009). Interacting meaningfully with machine learning systems: Three experiments. *International Journal of Human-Computer Studies*. 67, 8 (Aug. 2009), 639–662.
[Subrahmaniyan et al. 2008] Subrahmaniyan, N., Beckwith, L., Grigoreanu, V., Burnett, M., Wiedenbeck, S., Narayanan, V., Bucht, K., Drummond, R. and Fern, X. (2008) Testing vs. code inspection vs. … what else? Male and female end users' debugging strategies, In *Proc. CHI*, ACM, 617-626.
[Szafron et al. 2003] Szafron, D., Greiner, R., Lu, P., and Wishart, D. (2003) Explaining naive Bayes classifications. Tech report TR03-09, University of Alberta.
[Turner and Turner 2011] Turner, P. and Turner, S. (2011) Is stereotyping inevitable when designing with personas? *Design Studies* 32, 1, January 2011, 30-44.
[van der Meji and Carroll 1998] van der Meij, H., and Carroll, J. M. (1998) Principles and heuristics for designing minimalist instruction. In *Minimalism Beyond the Nurnberg Funnel,* J. M. Carroll, Ed. MIT Press, Cambridge, MA, 19–53.
[Wharton et al. 1994] Wharton, C., Rieman, J., Lewis and C., Polson, P. (1994) The cognitive walkthrough method: A practioner's guide. In J. Nielsen and R. Mack (Eds.), *Usability Inspection Methods*, 105-140, John Wiley, NY.
[Wulf et al 2015] Wulf, V., Müller, C., Pipek, V., Randall, D., & Rohde, M. (2015). Practice based computing: empirically-grounded conceptualizations derived from design cases studies. In V. Wulf, K. Schmidt, & D. Randall (Eds.), *Designing Socially Embedded Technologies in the Real-World*. London: Springer.

[Yang and Newman 2013] Yang, R., and Newman, M. W.(2013) Learning from a learning thermostat: Lessons for intelligent systems for the home. *ACM International Joint Conference on Pervasive and Ubiquitous Computing*, 93–102.

[Yang and Pedersen 1997] Yang, Y. and Pedersen, J.O. (1997) A comparative study on feature selection in text categorization. *Twentieth International Conference on Machine Learning*, 412-420.

[Zeldin and Pajares 2000] Zeldin, A. L. and Pajares, F. (2000) Against the odds: Self-efficacy beliefs of women in mathematical, scientific, and technological careers. *American Educational Research Journal* 37, 215-246.