Designing for Advanced Personalization in Personal Task Management

Mona Haraty, Joanna McGrenere
Department of Computer University of British Columbia
{haraty, joanna}@cs.ubc.ca

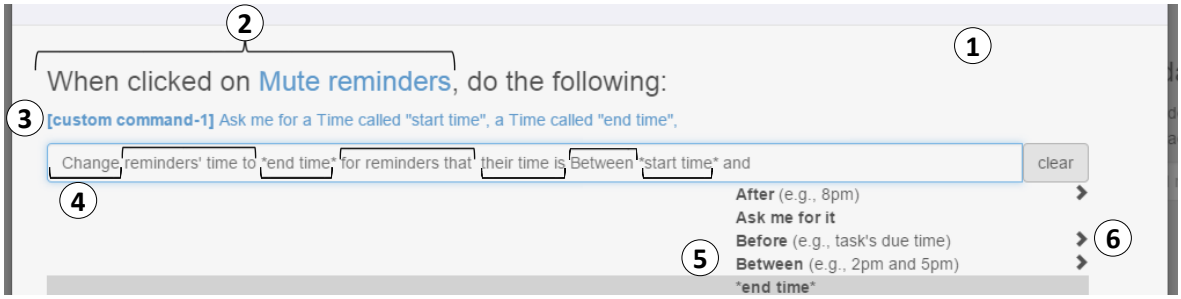

Figure 1: ScriPer is a guided scripting mechanism for constructing a new feature. Here, a user is adding a "Mute reminders" button to postpone reminders between a user-defined period. She has already created the button (trigger is marked as 2) and the first part of its behavior which is to ask for "start time" and "end time" (labeled as [custom command-1], shown in blue (3)). The screenshot is capturing the creation of the second part of the script (4). The script is composed by selecting one of the options from ScriPer's suggestions at each step (5). ScriPer starts with suggesting a set of actions, and updates its suggestions based on the rest of the script. Typing in the textbox filters the suggestions. Clicking on arrows (6) cycles through usage examples of the suggestions.

ABSTRACT
Many applications provide personalization mechanisms through which users can make changes to adapt a system to better fit ptheir needs or preferences. However, *advanced* personalization, such as extending system functionality, is often only available to programmers. Building on ideas from end-user programming and personalization literature, we developed an adaptable task management tool that allows advanced personalization using a self-disclosing mechanism and a guided scripting mechanism, ScriPer. We present our design process, its outcome, and the results of a user study (n=24). Participants, even those with no to some background in programming, were able to use ScriPer to perform advanced personalization (in 142 of 144 trials). We also found error patterns differed across programming expertise.

Author Keywords
Meta-design; personalization; personal task management.

ACM Classification Keywords
H.5.2 User Interfaces (D.2.2, H.1.2, I.3.6);

DIS 2016, June 04 - 08, 2016, Brisbane, QLD, Australia

ACM 978-1-4503-4031-1/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2901790.2901805 .

INTRODUCTION
Many applications provide personalization (customization) mechanisms through which users can make changes to adapt the system to better fit their needs or preferences. But the personalization available is often quite basic, which cannot support the diversity of user needs. Advanced personalization—broadly defined as personalization that goes beyond changing the look and feel, and involves changing functionality—often requires programming skills. In this research, our goal is to bridge the gap between simple and advanced personalization mechanisms by designing a mechanism that supports creation of advanced personalization without requiring the user to code.

Current apps are often limited to basic personlizations such as making simple changes to the visual appearance of interface elements (e.g., changing icons or a background), customizing access to functionality (e.g., adding, removing, re-arranging commands/buttons to/in a toolbar or defining a shortcut), and modifying system behavior by choosing options from a list of predetermined alternative behaviors. More advanced personalization such as extending system functionality are possible through mechanisms such as macros and add-ons, but these mechanisms have limitations. Recording a macro extends a system's functionality by encapsulating a sequence of repeated user actions that can be invoked later. But sophisticated macros that add new functionality require users to edit the code generated by the macro recorder which requires programming skills. Tools such as web browsers enable users to extend system functionality by creating and installing add-ons. However,

end users are restricted to *using* pre-existing add-ons, unless they have the programming skills to develop new add-ons.

To achieve our goal of designing tools that support advanced personalization, we built on ideas from end user programming (EUP) approaches such as controlled natural languages and sloppy programming [19], and followed guidelines on designing personalizable tools such as meta-design guidelines [8]. We chose personal task management (PTM) as the design domain, because PTM tools need to support advanced personalization in order to accommodate differences in PTM behaviors both across individuals and over time [12,13]. We designed a prototype of a personalizable PTM tool with two key components for enabling the creation of new functionalities: 1) a self-disclosing mechanism that reveals system functionality to users and thus makes it easier for users to understand what can be changed, 2) a guided scripting personalization mechanism (ScriPer) that enables users to construct new features by combining building blocks that are familiar to them. To investigate the strengths and challenges of these two components, we conducted an exploratory user study. Participants, even those with no or some programming background, were able to use our personalization mechanism to complete personalization tasks, except for 2 out of 144 trials. All the participants made mistakes. While programming expertise was not associated with the number of mistakes made, participants with no to some programming background produced different error patterns than programmers. Our primary contributions are: 1) the design of a personalizable PTM tool with two key personalization components, 2) empirical evidence of the challenges and strengths of both our personalization and self-disclosing mechanisms. A secondary contribution is our design process that provides additional insights into how to employ the theoretical guidelines on designing personalizable tools.

RELATED WORK

We review the guidelines on designing personalizable tools as well as EUP approaches that informed our design process.

Guidelines on designing personalizable tools

Henderson and Kyng looked at the practice of designing in use and described three activities that change the behavior of a technology: choosing between alternative anticipated behaviors, constructing new behaviors from existing pieces, and altering an artifact through modifying the source code [12]. The focus of our work is on constructing new behaviors from existing pieces. One of the comprehensive sets of principles for designing for adaptability is outlined by Moran as the principles of everyday adaptive design: overbuild infrastructure, under-build features, convey the adaptable quality of a tool as opportunity to the user, allow for recombining and repurposing (modularity), and make adaptations sharable [21]. Similarly, meta-design provides another comprehensive set of guidelines. Meta-design is a theoretical framework for empowering users to design their own tools by providing them with appropriate tools and opportunities [8]. Meta-design guidelines include: provide building blocks, under-design for emergent behavior, establish cultures of participation, share control, promote mutual learning and support of knowledge exchange, and structure communication to support reflection on practice. Key common requirements of both sets of guidelines is that software systems provide mechanisms that allow users to create complex personalizations by combining building blocks (modular components), and that systems should under design to promote personalization.

In our research, we focus on providing users with building blocks as well as mechanisms for combining the building blocks to create advanced personalization in a PTM tool.

While some of the other design methodologies (e.g., software shaping workshops [4]) include somewhat concrete practical steps for specific design situations, prior systems that have explicitly employed *meta-design* guidelines to our knowledge have been mostly domain-oriented design environments. Two examples are FRAMER for user interface design [16], and JANUS for kitchen design [9]. In these design environments, the primary user activity was to design, thus the building blocks were "design units" such as sink and refrigerator in the case of the kitchen designer, and windows and menus in the case of the user interface designer. Identifying building blocks of a non-design environment, such as a PTM tool, is less clear.

End user programming (EUP) approaches

EUP methods often take one of the following approaches: programming by demonstration, visual languages, and scripting. In our work, we focus on the scripting approach. Two approaches to improving a scripting mechanism are: (1) simplifying the format or syntax, and (2) using a scripting editor that ensures creation of a correct script, often referred to as a structure editor [5,18]. Natural languages [23] take the simplifying format approach. Sloppy programming is a form of natural language that attempts to simplify format by making programming similar to entering keywords into a Web search engine [19]. Systems such as CoScripter [17] for automating repetitive Web tasks and Inky [20]—a web command interface that allows users to automate tasks by entering unstructured text—have taken the sloppy programming approach. One limitation of this approach is that users might try commands that are not supported [17].

The structure editor approach addresses both this limitation and the issue of poor discoverability which is a limitation with all command line interfaces. The structure editor approach enables users to create commands by choosing options from menus, and it guarantees that only correct combination of options are selected. Controlled natural languages (CNLs), which are a subset of natural languages that have restricted dictionaries and grammars for reducing the complexity and ambiguity, combine both approaches of simplifying formats and structure editor. While CNLs have been explored for ontology authoring and semantic annotation (e.g., [3,10,11]), they have rarely been explored

for the purpose of automation or personalization. Atomate is an exception that has used a CNL interface to enable end user construction of reactive rules using information sources on the web such as one's online calendar, email client, and messaging services [15]; an example of a reactive rule constructed with Atomate is: "Have Atomate automatically update your facebook status when you are at a concert." While ScriPer is similar to Atomate in that they both use a CNL interface for creating behaviors, they differ in both the type of CNL interface and the usage context. As a result of the difference in the usage context, our approach provides finer grain building blocks as well as integration with the rest of the interface. In addition, our approach allows users to extend functionality of an under-designed PTM system by changing the behavior of already existing UI elements and defining behavior of new UI elements. Alfred is an automation tool that, similar to Inky, offers a command line for *running* commands [25]. Unlike Inky, Alfred supports creation of new commands but not through its command line interface; simple commands can be created using a visual programming interface where users can define flow of data between different apps; creating advanced commands requires programming knowledge. Unlike Inky and Alfred, the scripting mechanism in ScriPer is for *creation* of new behaviors, and using those behaviors—which is equivalent to running commands in Inky and Alfred—is done through the GUI elements in our prototype. While some of the EUP approaches have been studied, their effectiveness for people with no to little programming experience has been largely unexplored [15,20].

The contribution of our work is in bringing the EUP techniques to the context of personalization in PTM, and providing empirical evidence on the challenges and strengths in using them. We designed and developed a personalizable PTM prototype that includes a scripting mechanism (ScriPer) for creating advanced personalization. Our design incorporated both approaches of simplifying format and structure editor by using a scripting language that resembles natural language, and by presenting the space of applicable building blocks (language expressions) that can be used at each step of composing a script. A key difference between our approach and that of tools such as Alfred or Inky is that we use a command line interface for *creating* new behaviors for interface elements using very basic building blocks such as change, move, show, etc., rather than *running* predefined commands. Our goal was to design an approach for command creation that does not require programming.

Show my tasks' deadlines on a timeline
See & select appropriate tasks that can be done in a given time slot
Filter & show me tasks that were recorded today
Focus on the current tasks, minimize distraction by other tasks
Add an icon next to the tasks that [meet a certain condition]
View task lists & calendar together
Print tasks that are due today in a particular format
Set timer on tasks for tracking time
Color code tasks based on their list / goal
Strike through tasks when done

Table 1. Examples of user needs from prior PTM studies.

META-DESIGNING A PTM TOOL

Following the guidelines discussed earlier, we had two primary research questions in meta-designing a PTM tool: What are the building blocks of a PTM tool? And what personalization mechanisms should be provided to users for enabling them to combine those building blocks to create new functionality? Below we describe how we addressed these questions by reviewing our design process.

We developed a prototype of a basic PTM tool that supports basic functionalities such as creating task lists, adding tasks to lists, editing task attributes (e.g., color, due date, reminder), marking tasks as done, and deleting tasks.

Establishing the building blocks of a PTM system

Providing users with building blocks is the cornerstone of the existing guidelines on designing personalizable tools [2,8,22]. However, none provided concrete actionable guidelines as to how to come up with the building blocks for a system. To address our first question (what are the building blocks of a PTM system?), we hypothesized that understanding the types of desired personalizations would provide insight into what needs to be modifiable and thus the building blocks of a system. Several types of personalization (e.g., interface and functionality adaptation [21]) have been identified in the past, in domains other than PTM. However, previous categorizations of personalization were based on the personalizations that were *available* in the existing personalizable tools. What we needed, by contrast, were the types of personalizations that were not necessarily available but were *desired* and needed to be supported for accommodating differences in PTM behaviors both across individuals and over time. Thus, we reviewed users' various PTM needs reported in prior PTM studies (e.g., [1,11]), as well as the feature requests made by users of PTM tools such as Remember The Milk [26] which is one of the most active feature request forums related to PTM. See Tables 1 and 2 for examples of user needs and feature requests.

We considered user needs and feature requests as forms of personalizations that users should be able to create. Thus, we treated the words (e.g., task, change, due date) mentioned in the feature requests and user needs as the building blocks of a meta-designed tool, and categorized them into UI elements, actions, interactions, external events, entities, entities'

Snooze button for notifications
A "make current" button, that takes all selected overdue tasks and moves them to the present day
Ask for date to which to postpone when postponing a task
Customize reminders for specific lists/tags
Make 'delete' a button instead of an option in 'more actions'
Show tasks due today in bold
Show overdue tasks in the 'Today' tab on the Overview screen

Table 2. Example of feature requests in RTM.

attributes, and attributes' values. Table 3 illustrates examples of the building blocks in each of these categories.

The under-design guideline of meta-design [6] (or overbuild infrastructure and underbuild features of [21]) guided our decision of what actions to include as building blocks. According to this guideline: 1) building blocks should offer enough functionality that they are useful and usable as a unit and, 2) they should not be too complex to require users to break them down in order to combine them with other blocks [6]. In our design process, before adding a new feature based on a user need, we assessed the feasibility of building that feature using more basic blocks. If feasible, we added the new building block instead of the new feature. For example, we skipped adding an 'archive' feature, because archiving involves moving a completed task to a list called 'archive' and thus could be built by creating a list and using a 'move' building block which is more generic than 'archive.'

Creating new personalizations using the building blocks

After reviewing the user needs and feature requests for identifying the building blocks, we decided to focus on designing personalization mechanisms for two classes of personalizations: the first is adding a new feature to the system that can be invoked by interacting with a new interface element (e.g., a button or a menu-item); the second class is modifying the effect of an existing user interaction by adding new behaviors to it or changing its current behavior. While both classes require a mechanism for defining a new behavior, the first class involves creating a new interface element and attributing the new behavior to an interaction with it, and the second one involves attributing the new behavior to an existing interaction. Below, we describe how we designed for the above requirements.

ScriPer: Scripting for personalizing

To allow users to combine the building blocks for creating a new behavior, we designed ScriPer (**Scri**pting for **Per**sonalizing) which is a guided scripting mechanism. ScriPer allows users to create a script—representing their desired behavior—by choosing from a list of suggested building blocks that gets updated based on users' selected building blocks so far. ScriPer starts with suggesting a set of action building blocks (Figure 2.6), each of which has their own grammatical template. For example, the 'change' action block has the following template:

[[1]**change] [**[2]objects' attributes **to] [**[3]attribute's values**] [**[4]**for** (all) objects (that)**] [**[5]objects' attributes**] [**[6]attributes' values**].**

Category	Examples of building blocks
UI element	Button, checkbox
Action	Change, ask me for [data], show, move, remove.
Entity	Task, list, reminder
Attribute	Color, due date, type, status, importance, etc.
Value	Gray, tomorrow, long-term, done, high, etc.
Interaction	Click, right click, double click, drag, drop, hover
External event	Closing a web page, starring an email, etc.

Table 3. Categories of building blocks in a PTM system.

The numbers represent the order of ScriPer's suggestions for the 'change' block. After choosing 'change,' it suggests all the attributes of all the objects in the system to ask users what they want to change. Once an attribute is selected, it suggests all the possible objects that have that attribute (e.g., 'all the selected tasks' or 'tasks that'). If users choose objects such as 'tasks that' for which they have to specify conditions, then ScriPer suggests conditions in two steps: first by suggesting the attributes of the objects on which users want to apply a condition, next by suggesting possible values of the selected attributes (see Fig 1.4). We chose the order of ScriPer's suggestions such that a complete script forms a correct English sentence. This order was chosen to increase accessibility to non-programmers, at the cost of being contrary to mainstream programming paradigms (e.g., object oriented programming where objects come before actions).

The 'move' block has a slightly different template:

[[1]**move] [**[2](all) objects (that)**] [**[3]objects' attributes**] [**[4]attributes' values**] [**[5]to (day)(list)(position in a list)**] [**[6](day's values) (list names) (positions' values)**].**

ScriPer suggests values for an attribute based on the type of the attribute. For example, if the user selects an attribute such as 'due date' whose type is date, ScriPer shows a list of dates such as today and tomorrow, as well as a 'pick a date' block that when chosen, a date picker will be shown. Fig 3 illustrates this for the reminder's time attribute. When the script inside the textbox represents a complete script, ScriPer shows the two buttons of 'and' and 'save' (Fig 4) to signal to users that they can either add another script or just save their current script. ScriPer is implemented as a modal pop-up window that can be invoked either through a self-disclosing mechanism [5] or by clicking on a newly created interface element (e.g., a button) in the "personalization mode"; we describe the purpose of each approach next.

Personalization mode

To allow users to create new interface elements, we distinguish between the main mode, where all the regular PTM-related activities take place, and the personalization mode where personalization-related activities such as adding a new button or a menu-item happen. Switching modes is done by clicking on 'Personalization'/'Exit personalization' button (Fig 2.1); personalization mode adds a gray overlay to the main interface and all the regular PTM-related interactions are disabled such that the user interactions will show their expected effects in a panel (through the self-disclosing mechanism described next) rather than being executed. To add a new button, users click on the plus button next to other buttons (Figure 2.7), name the button, and then click on it to define its behavior using the ScriPer.

A self-disclosing mechanism

The second class of personalization that our prototype supports is modifying the effect of an interaction with an existing interface element (e.g., the effect of clicking on a button) by either adding a new effect or replacing an existing

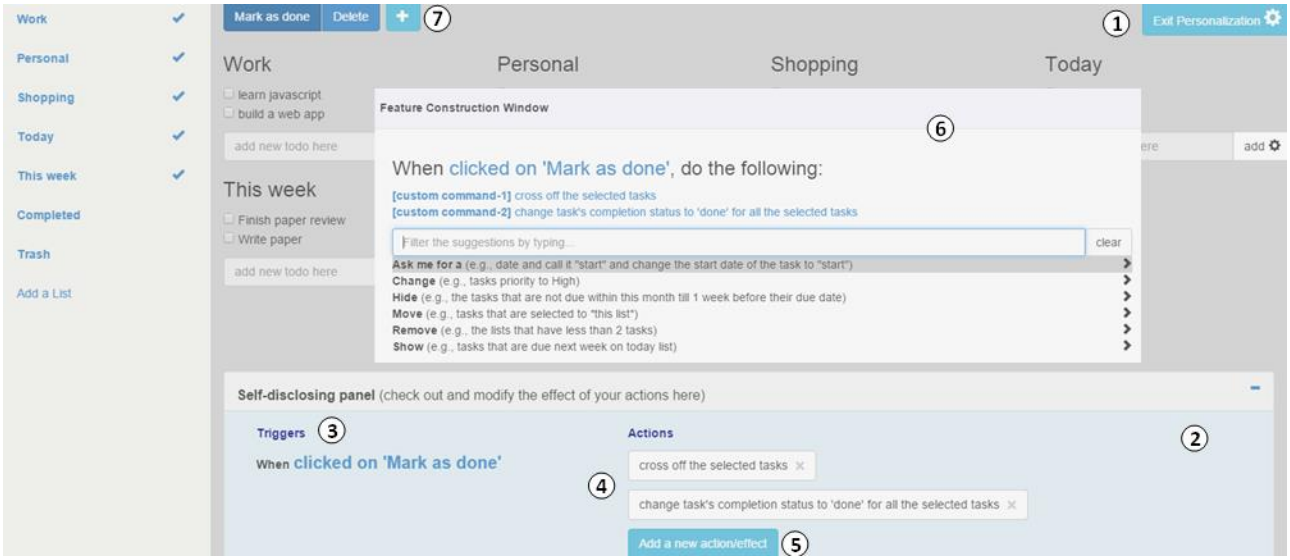

Figure 2: The prototype in personalization mode, hence the gray overlay (1). The plus button (7) is only displayed in personalization mode. The 'Mark as done' button has been clicked and thus the panel (2) is showing the effects (4) of that event (3). In this screenshot, the user is adding a new effect to the 'Mark as done' button by clicking on "Add a new action/effect" (5) which has invoked the ScriPer window (6). ScriPer starts with suggesting a set of action building blocks. Next to each action block are examples of using the block to familiarize the user with the block.

one. To support such personalization, following one of Moran's principles of everyday adaptive design [21], we first needed to *convey the adaptable quality* of user interactions so users know they can change the effect of their interaction. To do this, we display the effect of an existing interaction, building on the idea of self-disclosing systems that disclose their behaviors to users [6]. Whenever a user interacts with an interface element, the interaction (Fig 2.3) and its effects (Fig 2.4) are displayed in a fixed panel at the bottom of the page (Fig 2.2). The interaction is shown as a trigger and its effects are shown as actions. The background color of the panel changes to blue for one second when the user interacts with an interface element to clarify for the user the connection between her interaction and what is being displayed in the panel. The panel appears in both the main and the personalization modes, and its display can be toggled. The effects of an interaction—displayed as actions—can be modified via the ScriPer window. New effects can be constructed and assigned to the displayed trigger by clicking the "Add a new action/effect" button (Fig 2.5) which invokes ScriPer (Fig 2.6).

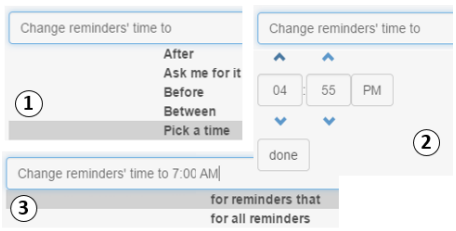

Figure 3: To select a value for a reminders' time, (1) the user chooses 'pick a time', (2) then ScriPer shows a time picker for the user to select a time, (3) after picking 7 AM and pressing done, ScriPer adds the picked value to the script.

EVALUATION

We conducted an exploratory lab study where people with various levels of programming experience used the tool to perform a set of predetermined personalization tasks. The goal of our study was twofold: (1) to evaluate our design decisions and understand the strengths and potential challenges involved in using the two components of our meta-designed tool—ScriPer and the self-disclosing mechanism—for personalizing software, and (2) to assess the effect of programming experience on the ability to perform personalization tasks.

Participants

Twenty four participants completed the study (13 females). Participants were recruited by posting signs around a large North American university campus as well as emails to different departments. Participants ranged from 21 to 31 years of age. They were all university students, and only 6 (3 females) were from the computer science department.

Prior to signing up for the experiment, interested participants filled out a short questionnaire to describe their programming expertise and rated it on a scale of 1-3 (1 little to no programming background, 2 some programming background, and 3 proficient in programming). While our design was targeted at the first two groups, we included the third group for comparison purposes. We were able to recruit 8 participants in each expertise category. In the rest of the

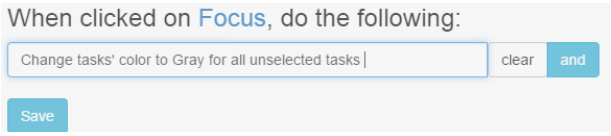

Figure 4: A grammatically correct script.

paper, participants are referred to as their gender (M/F) + expertise (N/S/P) + a number (1-8). For example, a male proficient programmer is referred to as MPx where x is between 1 and the number of participants in that category.

Tasks

To maintain ecological validity in designing personalization tasks, we reviewed user needs from prior PTM studies as well as the feature request forum of a PTM tool, Remember the Milk. Table 1 and 2 shows examples of user needs and feature requests that influenced our tasks. Based on those examples, we designed six personalization tasks, all of which could be performed using our prototype.

Four of the tasks involved creating a new button and defining its behavior. For each of these tasks, we designed a group of two tasks—henceforth a task group (TGi). The second task in each group was a personalization task, which explicitly asked participants to create a button that performs a desired personalization. The first task was to perform what the button would do prior to performing any personalization. The first task involved some repetition, which was to motivate the need for the personalization. This is often challenging when designing personalization tasks for lab studies. Another goal of the first task was to familiarize participants with the goal of the personalization they were asked to perform in the 2nd task. The task groups TG1, TG2, TG5 and TG6 were designed this way (see Table 4).

The remaining task groups (TG3 and TG4) involved using the self-disclosing mechanism. TG3 included 3 tasks, and TG4 included 4 tasks. The last task of each of these task groups was a personalization task that asked participants to change the effect of an interaction with already-existing interface components (e.g., 'mark as done' button). The other tasks in each of the task groups were designed to familiarize the participants with that interaction and its current effects. In those tasks, participants performed an action, e.g., marking a few tasks as done, and explained its current effect.

Procedure

First, to familiarize participants with the system, we walked them through performing a personalization task. Next, they were given all tasks one at a time in the order shown in Table 4, and were asked to think aloud while performing the tasks without any time limit. The screen and the audio were recorded. After finishing the tasks, participants were asked about their experience with ScriPer and the self-disclosing mechanism in a semi-structured interview. The session took on average 42 minutes (min=20, max=60).

Data analysis

We collected usage log data (the personalization scripts and task completion), the screen recordings, the interview data, and the notes of the participants' actions and comments. The screen recordings and the transcriptions of the think aloud were coded against the number and type of mistakes, and whether or not they reused their created personalization in subsequent tasks. The semi-structured interviews were transcribed and coded against the strengths and challenges of different parts of the design. We ran mixed-model regression to analyze participants' success in completing the tasks without mistakes and the number of mistakes. In the

Task groups	Tasks
TG1	**T11:** Change the due date of the following tasks to tomorrow: "finish paper review", "learn javascript", "do yoga"
	T1P: Imagine that you'd like to do the previous task for a whole bunch of tasks and that you might need to do this again in the future. For this situation, you decide to create a button (called 'postpone to tomorrow') that when you click on, the system modifies the due dates of the to-dos that you have selected to tomorrow.
TG2	**T21:** Find tasks that are overdue (i.e., due before today) and change their color to red so that you won't miss them.
	T2P: To save time on the previous task in the future, you decide to create a button called "Highlight overdues" that when you click on, it automatically turns the overdue tasks into red.
TG3	**T31:** Mark the following tasks as done to indicate that you are done with them: (Code, do yoga)
	T32: What did the system do when you pressed the 'Mark as Done' button? (please explain)
	T3P: Imagine that you would like the system to move your tasks to the bottom of the list when you are done with them, in addition to crossing them off. So, make the system do that.
TG4	**T41:** Create a list called "tomorrow".
	T42: Add the following new tasks to the tomorrow list and set their due dates to tomorrow: "buy bread", "register", "return book"
	T43: What does the system do when you add a task to a list? (please explain orally)
	T4P: In addition to adding the task to the bottom of the list, make the system set the tasks' due date to tomorrow by default when you enter a task in this list.
TG5	**T51:** You do not want to disturb your sleep by the automated task reminders that are sent when you are asleep. So, find tasks that their reminders are set to be sent out between 10 pm and 7 am and postpone them to 7 am.
	T5P: Imagine that there are other times that you you'd like to define quiet hours so that you tell the system a time period in which you don't want to receive any reminders and the system postpones sending the reminders to the end of that period. In this situation, you decide to create a button called 'mute reminders' that when you click on, the system asks you to enter the time period and then the system changes the reminders that are supposed to be sent out within that period such that they will be sent out at the end of that period.
TG6	**T61:** Today, you want to focus on the following 3 tasks (Read paper, learn javascript, Finish paper review). Gray out the rest of your tasks so they don't distract you.
	T6P: Imagine that you'd like to do the previous task again in the future. For this situation, you decide to create a button called 'Focus' such that when you select the tasks that you want to focus on and click on the 'Focus' button, it makes the other tasks gray.

Table 4. Tasks used in the experiment. Participants were given one task at a time.

regression models, we included the fixed effects of the number of tasks already attempted, programming expertise, gender, and age, as well as the random effects of the task that was being attempted and participant.

We *only* analyzed personalization tasks (T1P-T6P) in the task groups. Some participants performed personalization even for the non-personalization tasks in which they were not explicitly asked nor expected to personalize, i.e. the first task in TG1, TG2, TG5, and TG6. Some of them skipped the personalization task in the task group as they recognized that they had already performed that task. In these cases, we considered their first tasks as their personalization tasks.

FINDINGS

Task completion and mistakes made

Except for two participants who gave up completing T5P, participants did complete all personalization tasks, albeit some with mistakes (due to the iterative nature of writing a script, only uncorrected mistakes are counted as mistakes). A mistake meant that the solution was either a slightly or completely different personalization than the intended one. Out of the 144 (24x6) trials of the six personalization tasks, only 2 were left incomplete, 94 were completed successfully with no mistake (Fig 5-A). In the remaining 48 trials, 54 mistakes were made in total. The number of mistakes made in a single task ranged from 1 to 3, with only 1 participant ever making 3 mistakes on a task (T5P). Table 5 illustrates the distribution of mistakes across the tasks.

We grouped the similar mistakes, and labeled the 4 emergent groups as: lack of precision, terminology related, mental model mismatch, and wrong trigger. In 41% of the mistakes, a wrong block was chosen due to *lack of precision*, e.g., choosing 'for all tasks' instead of 'for all *selected* tasks' or 'yesterday' instead of 'before today'. 37% of the mistakes, were *terminology related*, e.g., using 'completion date' instead of 'due date' or 'time' instead of 'date'. 18% of the mistakes were due to changing the effect of a *wrong trigger* which was most common in the tasks that required use of the self-disclosing mechanism (T3P, T4P). For example, when performing T4P, which required attributing a new effect to the 'add' button, some participants added the new effect to an irrelevant trigger that was displayed through the self-disclosing mechanism because that irrelevant trigger happened to be their last interaction with the system. Finally, only 4% of the mistakes were due to a *mental model mismatch* such as a mismatch between the functionality of a building block and what users expected it to do. For example, 3 participants made an unnecessary use of the 'show' block in T21—where they were asked to find tasks that were overdue and change their color to red—before using the 'change' block. For example, FN4 created the following two scripts: "Show tasks that their due date is before today on calendar" and "Change tasks' color to Red for tasks that their due date is before today". But all those 3 participants mentioned "*I probably didn't have to use 'show'*" right after using the 'change' block. Fig 5-B illustrates a breakdown of mistake types across programming expertise. Compared to programmers, participants with no to some programming background made disproportionately more mistakes due to lack of precision and choosing a wrong trigger.

To examine if the number of mistakes were associated with programming expertise and other aforementioned factors, we ran a Poisson mixed model regression. Also, to analyze success in completing a task (0 mistakes vs. 1 or more), we ran a logistic mixed model regression. Neither of these analyses identified any significant predictors.

Unexpected personalization behaviors

Personalizing when performing non-personalization tasks: As mentioned, some participants performed personalization in tasks where they were not directly asked to do so (T11, T21, T51, and T61). Out of the 96 trials of these four tasks, 57 were done by creating a personalization. When asked

Task	Correct personalization script	# of mistakes in the script				
		0	**1**	**2**	**3**	**--***
T1P	[When clicked on **"Postpone"],** [Change] [tasks' due date to] [tomorrow] [for all selected tasks]	17	7	0	0	0
T2P	[When clicked on **"Highlight overdues"]**, [Change] [tasks' color to] [Red] [for tasks that] [their due date is] [before] [today/now]	10	12	2	0	0
T3P	[When clicked on **"Mark as done"]**, [Move] [all selected tasks to] [location in the list:] [bottom of the list]	20	4	0	0	0
T4P	When clicked on **"add",** [Change] [tasks' due date to] [tomorrow] [for all tasks in this list]	12	11	1	0	0
T5P	When clicked on **"Mute reminders",** [Ask me for] [a Time called "start"], [a Time called "end"] [Change] [reminders' time to] [*end*] [for reminders that] [their time is] [between] [*start*] and [*end*]	13	7	1	1	2
T6P	When clicked on **"Focus",** [Change] [tasks' color to] [Gray] [for all unselected tasks]	22	2	0	0	0
	Total Number of trials = 144	94	48			2

Table 5. Correct personalization for each personalization task and the number of participants who made 0, 1, 2, 3 errors in their scripts when performing each task. Each correctly selected building block of a personalization is shown within a bracket. * The last column shows the number of participants who left the tasks incomplete.

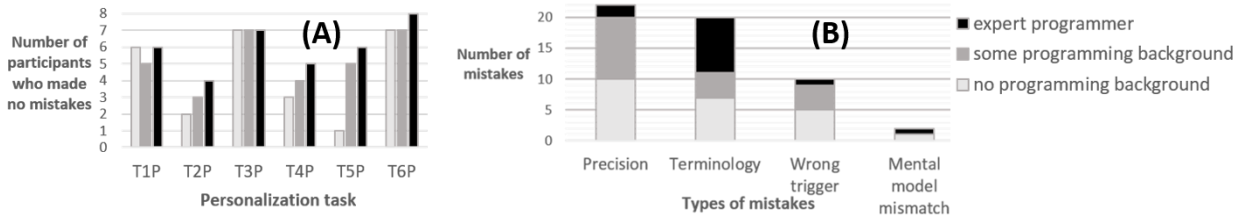

Figure 5: (A) Number of participants who successfully completed each task with no mistake, (B) Breakdown of mistakes across types and programming expertise.

about their choice, participants mentioned one of the following: 1) they were aware of the manual method but thought that the task involved "*too much hassle*" if done manually, 2) they mentioned that they never would have thought that they should do the task manually, or 3) they could not figure out how to do the task without personalizing. In an extreme case, FS4 couldn't find the 'Mark as done' button, when she was asked to mark two tasks as done in T31, and she created a button called 'completed.' A logistic mixed model regression of whether participants personalized in these tasks did not identify any significant predictor.

Creating more generalizable personalization than asked for: Four participants (MP2, MS2, MP3, MS4) created more generalizable personalizations when not asked to personalize; instead of the simpler anticipated solution of "*Change tasks' start date to tomorrow for all selected tasks*" for T11, they combined the following two scripts: "*ask me for a Date called 'dateChange'*" and "*change tasks' start date to *dateChange* for all selected tasks*".

Reusing a personalization created in prior tasks: Some participants reused their personalizations without being instructed to do so. Task T42 asked the participants to add 3 tasks to a list and set their due dates to tomorrow. To save time on this, 5 participants (MP3, FP2, MP5, FN5, FN6) reused the 'postpone to tomorrow' button that they had created in T1P, instead of manually changing the due dates of the three tasks. We expect to see such reuse behavior when users build their own personalization in a real-world setting, and it was reassuring to see this behavior in the lab setting.

ScriPer: strengths and challenges

Flexibility of the system

For each personalization task, we had anticipated a single solution, but participants performed some of the tasks differently. This showed the system's flexibility in supporting different ways of expressing a feature. For example, for T3P where they were asked to add a new effect to the 'Mark as done' button such that it moves the tasks to the bottom of the list, 3 participants performed the task with this script: "change tasks' location to the bottom of the list for all selected tasks" instead of our anticipated solution of "move all selected tasks to location: bottom of the list".

MP5, who used Todoist (a dedicated PTM tool), was eager to provide suggestions for improving the PTM support of our prototype before realizing that he could achieve some of his suggestions through personalization: *"that's the nice thing about the system, you can always edit and do anything you want. So for example, I want 'Mark as done' to move all of this into a 'done' list. I can easily do it with personalization*". Then he went ahead and changed the effect of 'Mark as done' button and said: "*So, it's hard to criticize the system because you can create anything you want. Like if you have anything missing, it's like a plugin, you can just create it."*

Some of the participants speculated about the potential usefulness of ScriPer in other apps: *"I like that you can construct features. Other apps don't do that. The gray out thing [referring to T6P], I have an app similar to this but it doesn't do the gray out…you can prioritize them [your tasks], but you have to do it individually, you can't say all these ones are priority ones together*" [FS1]. MP2 could see ScriPer being used for macro creation in spreadsheets: "*Instead of having to record your macro you can actually do something that's a bit more plain text [as in ScriPer] it's really frustrating to make those macros; they are always very strict…they won't allow any easy process.*"

Overall ease of use

Overall, participants liked the concept of creating their own features, and found ScriPer easy to use for the most part. Even programmers appreciated not having to code for the purpose of personalization: "*I loved the feature construction window. I'm a programmer but I don't like doing it when I don't have to especially for something like personalization"* [MP1]. One participant commented on the value of the save button in ScriPer: "*I liked the fact that when you want to save you need to complete every step. It's not like you complete half the steps and you can save it, that didn't work. It's giving you some feedback that you are right*" [MN2].

Findability of the building blocks among the suggestions

Participants took different approaches to finding their desired block among the suggested blocks: some participants typed a keyword they were looking for to filter the suggestion list, and others visually scanned the list to see what fits. In cases where the list of suggestions was long, participants who filtered seemed to find their desired block faster, based on observation as the time to select a suggestion was not logged. However, the filtering behavior led some participants to either make precision-related mistakes or create less efficient scripts, as they sufficed to the first best-matched block and did not find the correct block, which was filtered out. For example, MP3 inefficiently performed T6P;

while the intention was to gray out the unselected tasks, he created the following two scripts: "change task's color to gray for all tasks" + "change tasks' color to green for all selected tasks." He missed the option of "for all *unselected* tasks" when writing the first script. Some participants preferred scanning the options rather than typing and filtering, because they anticipated a potential mismatch between their own vocabulary and that of the system: *"there wasn't so much stuff that I thought that [the filter] was necessary. I also realized I was scared I'd miss something if I didn't type it as exactly the way it was typed"* [MS4].

Match between the order of blocks and users' expectations

ScriPer imposes an order in which users are expected to express their desired personalization. While the majority of the participants mentioned that the order made sense to them, 7/24 pointed to a mismatch between the order they thought about the personalization and the order the blocks were suggested. All those 7 participants wanted to first identify objects and only then apply an action on them; however, in our prototype, the action building blocks were suggested first. For example, FP3 said: "*I like how it constructs it for you. But sometimes I felt like I have to think about the order of how to construct things in a certain way. First I need to select and then I need to do change whatever it was. I think it's more helpful to have the system help you construct it like this, but at the same time the rigidness made it hard to figure out.*" However, many participants (without prompting) pointed to the learning curve in getting to know the order of ScriPer's suggestions and that by their last tasks, they knew what suggestions they should be expecting and when.

Difficulty in creating composite data types

The 'ask me for' block was designed to be used for instructing the system to ask for data (e.g., a time, a text). Part of T5P involved using this block to instruct the system to ask users to enter a 'time period'—a composite data type. Fig 6 illustrates the steps involved. Following the under-design guideline, we chose not to include composite data types such as 'time period' as a building block, since they could be built using more basic blocks such as time. Thus, in T5P participants were expected to instruct the system to ask them for two time inputs, which proved to be difficult for some of the participants with little to no programming background. Two of the participants gave up completing T5P. For example, when performing this task, FP2 thought aloud "*I want the system to ask me for a range but this is only asking for one time. I don't know how to enter a time period.*"

Self-disclosing mechanism: strengths and challenges

The panel that disclosed the system behavior was available in both the main and the personalization modes. Most participants liked having it in the main mode. MS2 mentioned that he should be able to have it in both modes: "*that's kinda cool because it visually shows you what's being done for whatever is pressed live, whereas in personalization [mode] you have to go and click and see.*"

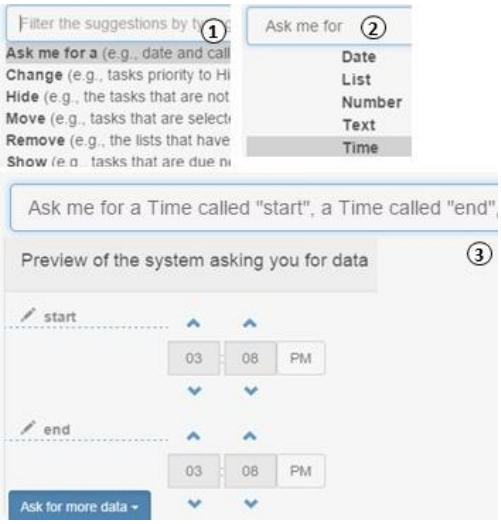

Figure 6: Steps involved in using the 'Ask me for' block to perform part of T5P.

However, as mentioned earlier in the discussion of mistakes, some participants had difficulty finding the right trigger, and made mistakes by attributing a new behavior to a wrong trigger (10/54 mistakes). The triggers shown in the panel are updated on each user action. Thus, to change the effect of an action through the self-disclosing mechanism, a participant has to first perform the action so that the panel displays it as a trigger. If done in the personalization mode, the action and its expected effects are displayed without being executed. However, participants who performed an action in the main mode just to make the panel display the right trigger had to undo the effect of their action. One approach to alleviate this issue is to show a history of user interactions in the panel, instead of only the last interaction, and allow users to choose their desired trigger manually from the panel without going through the process of performing and undoing an action, or having to switch to the personalization mode.

In addition, there was a mismatch in how the system set triggers and some of our participants' mental model of it. In our prototype, triggers are general actions such as "when clicked on the checkbox next to a task." To attribute a new behavior to a trigger such that the behavior is only applied to certain of objects (e.g. tasks that are in the 'shopping' list), participants had to specify those objects when constructing the new behavior, and not when setting the trigger. However, some participants expected to first define a more specific trigger such as "when clicked on the checkbox next to a task in the 'shopping' list", and then to add a behavior to it. Therefore, they avoided changing the effect of a general trigger: "*I was a bit scared of using the panel because it applies to very general actions… If I click on a task and then add an action I say oh my God I'm gonna screw up every time I click on that very generic action. I'll leave that for really general behaviors unless there are some sort of filtering built into that*" [MP3]. To resolve this issue, triggers need to be editable so that users can add conditions to them.

DISCUSSION AND CONCLUSION

The theoretical guidelines on how to design personalizable tools have been rarely put into practice. To our knowledge, our work is first in following such guidelines to design a personalizable tool that is not a design environment as in [9,16]. Our work paves the way for designing personalizable tools by revealing its detailed design process and putting the theoretical guidelines into practice, specifically by providing insights into *how* to identify the building blocks of a system and *how* to under-design, as well as providing an example of a mechanism (ScriPer) for combining building blocks to create some types of advanced personalizations; ScriPer is a proof of concept and was not intended to offer complete language expressivity; it covers trigger-action rules where the trigger is a user interaction with an interface element.

To identify the building blocks of a PTM system we reviewed user needs and feature requests of an existing PTM application. We think that our high level categories—UI elements, actions, interactions, external events, entities, entities' attributes, and attributes' values—provide a more practical starting point for designers of personalizable tools to identify building blocks of a system. We found the under-design guideline helpful in deciding whether to add a new feature. It led us to include more basic building blocks instead of adding more features or composite blocks. This approach increases the number of possible features users could build. However, our decision about not including the composite data type of 'time period'—since it could be built using two times—did not work out for some participants. Thus, under-design decisions need to be tested carefully to ensure that composite blocks can be built intuitively, especially for people with no programming background.

ScriPer is one possible design of a mechanism for combining building blocks to create advanced personalizations, and our preliminary evaluation shows that it is promising. One of the most encouraging results is that many participants intuitively personalized even when not required. Out of the 96 trials of four of the non-personalization tasks, 57 were done by creating a personalization. Further, most users were able to create advanced personalizations when instructed to. On the downside, however, ScriPer does not prevent the user from making mistakes, and indeed about one third of the created personalizations were either slightly or completely different than the intended ones. Part of the issue is the ability for the user to easily spot a mistake. Our evaluation did not include having participants using our tool with their own tasks, something that would have likely highlighted any mistake quickly. Further investigation is needed to see how well users are able to recover from their mistakes. Beyond recovery, it is important for a personalization mechanism to limit the possibility of making a mistake in the first place. Mistakes due to choosing a wrong trigger in the self-disclosing mechanism were related to not noticing the trigger part in the panel. To avoid such mistakes, the self-disclosing mechanism should emphasize the trigger part and ask users—once they are done with creating a personalization—to confirm that their personalization is attributed to the right trigger. Compared to programmers, participants with no to some programming background made disproportionately more mistakes due to lack of precision, i.e. mistakes such as choosing 'yesterday' instead of 'before today'. One approach to reduce the possibility of such error is to suggest other conceptually similar options to what they have selected or are about to select. This can be done, for example, by highlighting those similar options when a user is about to select an option. This approach requires designers to determine clusters of conceptually similar building blocks which might be an added step to the design. Alternatively, a data-driven approach may be adopted by tracking how errors are made and then corrected.

One usability issue with ScriPer was related to the order it imposed on using the blocks. Action building blocks such as change, move, etc. were so basic (i.e., low-level) that they did not necessarily correspond to any user interaction or interface element in the system. Therefore, ScriPer had to be able to provide specific suggestions for each step of composing a script to compensate for users' lack of familiarity with the actions and their parameters. Thus, we chose to impose an order for combining building blocks so the number of suggestions at each step would be manageable for users. The order allowed the personalization scripts to form an English sentence and provided the benefit of knowing what building block should be selected at each step. However, it did not match some participants' preferred order. Part of the problem was due to lack of visibility of the next steps, which was partly due to their dependence on user's prior selections. An alternative approach is to show all the steps to users and let users choose the order in which they want to complete each step. However, that might make the interface crowded and confusing. Also, systems that use higher level building blocks that are more familiar to users can support a flexible order. Inky is an example of such a system, since it replaces a GUI with a command line interface, where users are likely to be familiar with the available commands and their parameters.

Although the personalization tasks in our study were ecologically valid, they were not derived from our particular participants. Longitudinal studies are needed to assess if users can translate their *own* needs into personalizations and whether they can reuse those personalizations effectively. Similar to our prototype, even a fully developed PTM tool can be limited in its coverage of primitive building blocks. To overcome this, users should be able to add building blocks to the system; but this could be challenging. For example, adding an *action* building block such as 'hide' requires determining all the possible building blocks that can be combined with it, a template to represent arrangement of the building blocks, and the underlying functionality associated with the block. Although the first two can be achieved by people with no to some programming background, defining the underlying functionality associated with an action block perhaps needs to be left to programmers.

REFERENCES

1. Victoria Bellotti, Brinda Dalal, Nathaniel Good, Peter Flynn, Daniel G Bobrow, and Nicolas Ducheneaut. 2004. What a to-do: studies of task management towards the design of a personal task list manager. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM, 735–742. http://doi.org/http://doi.acm.org.proxy.lib.sfu.ca/10.1145/985692.985785
2. Richard Bentley and Paul Dourish. 1995. Medium versus mechanism: supporting collaboration through customisation. *Proceedings of the fourth conference on European Conference on Computer-Supported Cooperative Work*, Kluwer Academic Publishers, 133–148. Retrieved October 31, 2012 from http://dl.acm.org/citation.cfm?id=1241958.1241967
3. Abraham Bernstein and Esther Kaufmann. 2006. GINO–a guided input natural language ontology editor. In *The Semantic Web-ISWC 2006*. Springer, 144–157. Retrieved November 30, 2015 from http://link.springer.com/chapter/10.1007/11926078_11
4. Maria Francesca Costabile, Daniela Fogli, Piero Mussio, and Antonio Piccinno. 2004. Software environments for end-user development and tailoring. *Psychnology* 2: 99–122.
5. Allen Cypher, Mira Dontcheva, Tessa Lau, and Jeffrey Nichols. 2010. *No Code Required: Giving Users Tools to Transform the Web*. Morgan Kaufmann. Retrieved April 12, 2015 from http://books.google.ca/books?hl=en&lr=&id=-bJKhQWYesoC&oi=fnd&pg=PP2&dq=no+code+required&ots=aHQ20HID3N&sig=pFdjOWG3bmxD86QPlZo9LAS_DGg
6. Chris DiGiano and Mike Eisenberg. 1995. Self-disclosing design tools: a gentle introduction to end-user programming. *Proceedings of the 1st conference on Designing interactive systems: processes, practices, methods, & techniques*, ACM, 189–197. Retrieved April 13, 2015 from http://dl.acm.org/citation.cfm?id=225455
7. Gerhard Fischer and Thomas Herrmann. 2011. Socio-technical systems: a meta-design perspective. *International Journal of Sociotechnology and Knowledge Development (IJSKD)* 3, 1: 1–33.
8. Gerhard Fischer and E. Scharff. 2000. Meta-design: design for designers. *Proceedings of the 3rd conference on Designing interactive systems: processes, practices, methods, and techniques*, 396–405. Retrieved October 26, 2012 from http://dl.acm.org/citation.cfm?id=347798
9. G. Fischer, R. McCall, and A. Morch. 1989. JANUS: Integrating Hypertext with a Knowledge-based Design Environment. *Proceedings of the Second Annual ACM Conference on Hypertext*, ACM, 105–117. http://doi.org/10.1145/74224.74233
10. Norbert E. Fuchs, Kaarel Kaljurand, and Tobias Kuhn. 2008. Attempto Controlled English for knowledge representation. In *Reasoning Web*. Springer, 104–124. Retrieved November 30, 2015 from http://link.springer.com/chapter/10.1007/978-3-540-85658-0_3
11. Adam Funk, Valentin Tablan, Kalina Bontcheva, Hamish Cunningham, Brian Davis, and Siegfried Handschuh. 2007. *Clone: Controlled language for ontology editing*. Springer. Retrieved November 30, 2015 from http://link.springer.com/chapter/10.1007/978-3-540-76298-0_11
12. Mona Haraty, Joanna McGrenere, and Charlotte Tang. 2015. How and Why Personal Task Management Behaviors Change Over Time. *Proceedings of the 2015 Graphics Interface Conference*, Canadian Information Processing Society, GI'15.
13. Mona Haraty, Diane Tam, Shathel Haddad, Joanna McGrenere, and Charlotte Tang. 2012. Individual differences in personal task management: a field study in an academic setting. *Proceedings of the 2012 Graphics Interface Conference*, Canadian Information Processing Society, 35–44. Retrieved October 4, 2012 from http://dl.acm.org/citation.cfm?id=2305276.2305284
14. Austin Henderson and Morten Kyng. 1991. There's no place like home: Continuing Design in Use. *1991) Design at Work: Cooperative Design of Computer Systems. Lawrence Erlbaum Associates, Hillsdale, NJ*: 219–240.
15. Max Van Kleek, Brennan Moore, David R. Karger, Paul André, and m.c. schraefel. 2010. Atomate It! End-user Context-sensitive Automation Using Heterogeneous Information Sources on the Web. *Proceedings of the 19th International Conference on World Wide Web*, ACM, 951–960. http://doi.org/10.1145/1772690.1772787
16. Andreas C. Lemke and Gerhard Fischer. 1990. A cooperative problem solving system for user interface. *AAAI*, 479–484. Retrieved December 28, 2015 from http://www.aaai.org/Papers/AAAI/1990/AAAI90-072.pdf
17. Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: automating & sharing how-to knowledge in the enterprise. *Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, ACM, 1719–1728. http://doi.org/10.1145/1357054.1357323
18. H. Lieberman, F. Paternò, and V. Wulf. 2006. *End user development*. Springer. Retrieved October 2, 2012 from http://books.google.ca/books?hl=en&lr=&id=9bTgzKA1fkYC&oi=fnd&pg=PR7&dq=component-based+approaches+to+tailorable+systems&ots=ujhXsoIR-e&sig=atGOBVjZktFJw1SeVEzbVLG2V3I
19. Greg Little, R. Miller, V. Chou, Michael Bernstein, Tessa Lau, and Allen Cypher. 2010. *Sloppy programming*. Morgan Kaufmann. Retrieved April 12, 2015 from http://books.google.ca/books?hl=en&lr=&id=-bJKhQWYesoC&oi=fnd&pg=PA289&dq=sloppy+programming&ots=aHQ20HNGaK&sig=ECLvPwTV1wtc4A_jk0DU6JMleL8
20. R. C. Miller, V. H. Chou, M. Bernstein, G. Little, M. Van Kleek, and D. Karger. 2008. Inky: a sloppy command line for the web with rich visual feedback. Retrieved November 19, 2012 from http://dspace.mit.edu/handle/1721.1/51696
21. Thomas Moran. 2002. Everyday Adaptive Design (keynote). Retrieved from http://www.sigchi.org/dis2002/
22. Anders Mørch. 1995. Application units: Basic building blocks of tailorable applications. In *Human-Computer Interaction*. Springer, 45–62. Retrieved October 10, 2013 from http://link.springer.com/chapter/10.1007/3-540-60614-9_4

23. Brad A. Myers, John F. Pane, and Andy Ko. 2004. Natural Programming Languages and Environments. *Commun. ACM* 47, 9: 47–52. http://doi.org/10.1145/1015864.1015888

24. R. Oppermann and H. Simm. 1994. Adaptability: User-initiated individualization. *Adaptive User Support–Ergonomic Design of Manually and Automatically Adaptable Software. Hillsdale, New Jersey*. Retrieved November 5, 2012 from http://books.google.ca/books?hl=en&lr=&id=0N9xdnrutSoC&oi=fnd&pg=PA14&dq=Adaptability:+User-Initiated+Individualization&ots=ptwQxyR8-j&sig=BWDthV-2warCk0Ko4oO3KQ0IYmI

25. *Alfred*. Retrieved from https://www.alfredapp.com/

26. Remember The Milk - Forums / Ideas. Retrieved March 30, 2016 from http://www.rememberthemilk.com/forums/ideas/