Information and Software Technology 103 (2018) 55–74

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

What happened to my application? Helping end users comprehend evolution through variation management

Sandeep Kaur Kuttal[*,a], Anita Sarma[b], Gregg Rothermel[c], Zhendong Wang[d]

[a] University of Tulsa, USA
[b] Oregon State University, USA
[c] University of Nebraska-Lincoln, USA
[d] University of California, Irvine

ARTICLE INFO

Keywords:
End-user programming
End-user software engineering
App inventor
Variation management

ABSTRACT

Context: Millions of end users are creating software applications. These end users typically do not have clear requirements in mind; instead, they debug their programs into existence and reuse their own or other persons' code. These behaviors often result in the creation of numerous variants of programs. Current end-user programming environments do not provide support for managing such variants.

Objective: We wish to understand the variant creation behavior of end user programmers. Based on this understanding we wish to develop an automated system to help end user programmers efficiently manage variants.

Method: We conducted an on-line survey to understand when and how end-user programmers create program variants and how they manage them. Our 124 survey respondents were recruited via email from among non-computer science majors who had taken at least one course in the computer science department at our university; the respondents were involved in the Engineering, Sciences, Arts, and Management fields. Based on the results of this survey we identified a set of design requirements for providing variation management support for end users. We implemented variation management support in App Inventor – a drag and drop programming environment for creating mobile applications. Our support, AppInventorHelper, is meant to help end-user programmers visualize the provenance of and relationships among variants. We conducted a think-aloud study with 10 participants to evaluate the usability of AppInventorHelper. The participants were selected on a first-come, first-served basis from those who responded to our recruitment email sent to list-servers. They were all end users majoring in electrical engineering, mechanical engineering, or physics. None had formal training in software engineering methods, but all had some experience with visual programming languages.

Results: Our (user study) results indicate that AppInventorHelper can help end users navigate through variants and find variants that could be utilized cost-effectively as examples or actual code upon which to build new applications. For example, in one of our empirical studies end users explored variants of a paint application in order to find a variant that could easily be extended to incorporate a new feature.

Conclusions: Our survey results show that end users do indeed reuse program variants and suggest that understanding the differences between variants is important. Further, end users prefer running code and looking at outputs, accessing source code and meta information such as filenames, referring to the creation and update dates of programs, and having information on the authors of code. When selecting variants users prefer to look at their major features such as correctness, similarity and authorship information. End users rely primarily on memory to track changes. They seldom make use of online or configuration management tools. Hence, integrated domain-specific variation management tools like AppInventorHelper can significantly help improve users' interactions with the system. A key contribution of our work is a set of design requirements for end-user programming environments that facilitate the management and understanding of the provenance of program variants.

* Corresponding author.
E-mail address: sandeep-kuttal@utulsa.edu (S.K. Kuttal).

https://doi.org/10.1016/j.infsof.2018.06.008
Received 20 September 2017; Received in revised form 15 June 2018; Accepted 19 June 2018
Available online 20 June 2018

1. Introduction

End-user programmers are individuals who create software without any formal training in software engineering methodologies. Millions [60] of them write simulations using LabVIEW [43], design websites using HTML, program web-applications using Yahoo Pipes [72], and design mobile applications using App Inventor [1], to name just a few popular platforms.

It is well known that end-user programmers learn by looking at and using examples [10,47]. In fact, end users are known to opportunistically employ reuse while programming. A case study of more than four thousand mobile apps across five different categories in the Android Market revealed that on an average 61% of the classes appeared in two or more apps through inheritance, libraries, or frameworks [59]. To support such reuse, most end-user programming environments provide online repositories in which program variants are publicly archived. Evidence suggests that end users do make use of these repositories: previous studies [66] have found that more than 56% of the instances of code in repositories of end-user programs are clones. Another mechanism for reuse is to use programs as sub-programs in other applications. For example, in a study of Yahoo Pipes (a web mashup environment)[1] it was found that 27.4% of the programs in the Yahoo Pipes repository had been used as sub-programs [39]. Thus, even though end users may view their programs as "throw away," their code often ends up being long-lived and in many cases is reused by other end-user programmers [32].

End-user programmers' requirements tend to be implicit, and emerge and change over time [32]. Often, requirements are poorly defined and there is no single way in which to satisfy them [68]. Moreover, while creating programs, end users' design decisions and coding activities are typically interleaved [32]. Further, end users program opportunistically; i.e., they tend to select from alternative solutions that are available based on their own assessments of the strengths and weaknesses of those solutions [10]. Finally, end users tend to "debug their programs into existence" [58]; i.e., they investigate alternative strategies and backtrack through changes to arrive at solutions [40,58].

Changing requirements, code reuse ("clone and own" and "accidental" sharing of code), and programming styles (opportunistic programming and debugging programs into existence) tend to create large numbers of program variants. While the provision of programs in repositories helps support end-user programming to an extent, finding appropriate variants remains a challenging task.

End-user programming environments typically do not provide facilities for managing variation. While developing applications, end-user programmers may have difficulties determining which variant of a program to begin with, which variant is the most recent, and which variants may have been created by other programmers in order to improve or extend their code. Professional software engineers use variation management systems to support code reuse, change traceability, and maintenance [21,34]. Therefore, bringing the benefits of variation management into end-user programming environments is likely to empower end users with their programming tasks as well.

When considering variation management from the point of view of end-user programmers, we find it useful to distinguish two different ways in which variation occurs. In the first case, an end-user who has some "ownership" of a program may modify it or enhance it over time, creating new versions that replace prior ones, and essentially viewing the program as having a single, persistent identity. This is akin to developers working along a single line of successive releases of a product, or improving *one* of the products that makes up a product line. In the second case, an end-user may begin with an existing program, and essentially "clone" that program, also with the goal of enhancing or modifying it, but viewing it as the beginning of a new, independent line of programs. This is akin to developers creating a new permanent branch off an existing product, or adding a new product to a product line. In this work, we refer to all of the programs created in either of these fashions as "variants", because they have shared provenance with some particular program, and we refer to programs created in the first manner as "versions". In our earlier work [40], we showed that approaches for managing *versions* can be useful for end-user programmers. In this work we widen our focus to *variants*.

We hypothesize that by providing support for managing variants in addition to versions, we can enable end-user programmers to explore and understand structural and behavioral differences between different instantiations of programs. This, in turn, should help them create programs using variants of their choice as they can select alternative variants, try various combinations of variants, and appropriately mix features from different variants.

In this article,[2] we first summarize the results of a survey that we conducted with the aim of understanding why and how end users create, find, and manage variants. Based on the survey results, we define a set of design requirements for programming environments that support the management of variants in end-user domains. We then describe AppInventorHelper, a prototype supporting variation over space in the App Inventor programming environment. With the help of visualizations provided by AppInventorHelper, end users can: (1) visualize all program variants at once, (2) view relationships between variants, and (3) select appropriate variants based on various parameters, such as code similarity, author, date of creation, and date of update. Finally, we report results of a formative user study that shows that AppInventorHelper can help users select appropriate variants and understand the differences and similarities among variants. We also explore which environment features facilitate or hinder selection of variants by end users.

Our work makes the following contributions:

1. We present the first study focusing on understanding the behavior of end-user programmers with respect to how they create, find, and manage variants of similar programs.
2. We present data showing that a large percentage (88%) of end-user programmers report that they do create many program variants, while an almost equally large percentage (82%) of them do not use tools to track changes or manage variants; instead they rely primarily on memory to manage variants.
3. We present design requirements for end-user programming environments to facilitate the management, and understanding of the provenance of, program variants.
4. We implement our approach within the App Inventor environment. The visualization of program provenance provided by our implementation, however, generalizes to other programming environments because it does not rely on specific languages or tools.
5. We present an evaluation of our environment that shows that it can help end-user programmers find and manage variants.
6. We discuss implications for the design of future tools supporting variation management for end-user programming environments.

The remainder of this article is structured as follows. Section 3 describes our online survey and its results, and the design requirements it leads to. Section 4 describes our AppInventorHelper programming environment. Section 5 describes our empirical study design, and Section 6 describes the results obtained in the study. Section 7 describes the implications of our results for the design of environments

[1] Yahoo Pipes was launched in 2007 and discontinued in 2015. It was popular enough, however, to have spawned recent attempts to revive it. See, for example, http://www.pipes.digital.

[2] Much of the material in this article appeared originally in the first author's Ph.D. dissertation [36].

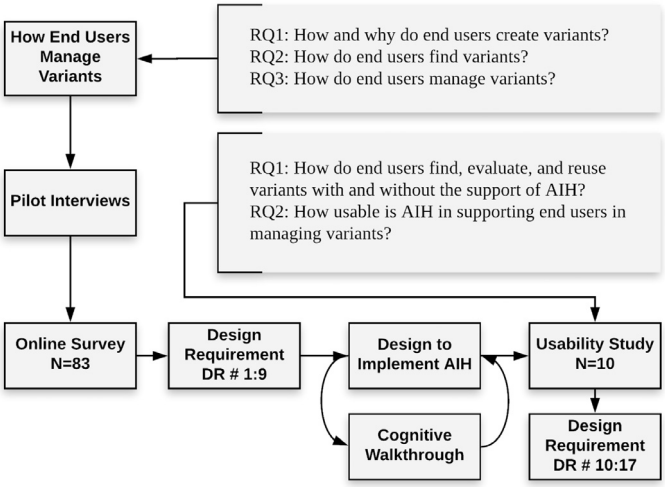


Fig. 1. Summary of overall research methodology.

supportive of variation management. Section 8 discusses related work, and Section 9 concludes.

2. Overall research methodology

Fig. 1summarizes the overall research methodology that we followed. Our first goal was to understand how end users manage their variants; this goal is informed by the three research questions in the box at upper right. We conducted pilot interviews with end users (Labview users) to understand what processes they use to manage their variants and whether they have any difficulties. Based on these interviews we constructed an online survey (following guidelines created by Kitchenham and Pleger [3], which received 83 responses from end-user programmers. We analyzed the survey results to inductively generate a set of nine design requirements (DR#1– DR#9) for tools supporting managing variants, by following guidelines listed in [17,23]. We then iteratively designed our AppInventorHelper environment by using the design requirements and cognitive walkthroughs on our prototypes. Finally, we performed a usability study of AppInventorHelper, informed by the research questions in the box second down from upper right. This study consisted of a with-in subjects evaluation of 10 end-user programmers where they had to modify an app by finding and reusing code from an existing variant (example). We analyzed our usability study results using inductive and abductive reasoning to derive eight additional design requirements (DR#10–DR#17).

3. Online survey

To the best of our knowledge, our work is the first to support end-user programmers' behavior in the presence of variants. Therefore, as a first step we needed to answer the following questions:

- **RQ1:** How and why do end users create and share variants?
- **RQ2:** How do end users find variants?
- **RQ3:** How do end users manage variants?

To answer these questions, we began by conducting interviews with two experienced LabVIEW programmers to understand how and why they created and shared variants. These interviews were semi-structured and focused on understanding how the programmers interacted with variants and learning what features and factors motivate the creation of variants. Some of the questions we asked were:

- how do you create a program?
- how do you share a program?
- do you reuse your programs?
- how do you find the programs you want to reuse?

The interviewees showed the first author their workspaces, and how they created, shared, organized, and managed alternatives (variants). They also discussed the challenges in retrieving and reusing these alternatives. We used the information gathered in our interviews, as well as the results from a prior survey of professional programmers [73], to create a structured online survey. We conducted a trial administration of the survey with four end-user programmers at the University of Nebraska-Lincoln. Given their feedback, we refined the survey questions iteratively to improve their relevance and clarity.

Our survey consisted of 14 background questions related to the demographics of the respondents, and nine survey questions. All survey questions were multiple choice, and asked the respondents to estimate the frequencies of various activities they conducted on a 5-point scale where 1 = All the time, 2 = Frequently, 3 = Sometimes, 4 = Rarely, and 5 = Never. The online survey[3] was designed using Qualtrics.[4] (As the text of the survey shows, we avoided using technical terms, and instead of referring to "variants" and "versions" we used common end-user terminology such as "copies of programs" and "modified code". These terms were selected from verbalizations from the two pilot interviews.)

3.1. Recruitment

We recruited survey respondents by sending email to non-computer science majors who had taken at least one course in the computer science department at our university. To participate in the survey, students were required to have some programming experience (in any language) and to be at least 19 years old (a requirement of our university's institutional review board.) Interested students answered the background and survey questions using the link provided in the email. Respondents were eligible to receive a raffle prize of $25.

3.2. Survey participants' demographics

Of the 124 respondents, 83 matched our criteria. Table 1 presents general demographic data on these 83 respondents. The respondents were from the Engineering (Mechanical, Chemical, Electrical, Civil, Construction), Sciences (Maths, Economics, Actuarial Sciences, Biochemistry, Physics, Agronomy), Arts (Music, BSEN & French, History,

[3] A complete version of the survey can be found in the Appendix.
[4] http://www.qualtrics.com.

Table 1
General demographics (N = 83).

Characteristics	Distribution
Gender	Male: 64 (77%)
	Female: 19 (23%)
Age	19–23: 66 (80%)
	24–29: 16 (19%)
	30–40: 1(1%)
Education	Undergraduate: 73 (88%)
	Graduate: 10 (12%)
Industrial	None: 70 (84%)
Experience	1–6 months: 6 (8%)
	6 months – 1year: 3 (4%)
	1–3 years: 3 (4%)
	> 3 years: 3 (4%)
Programming	< 1 year: 65 (78%)
Experience	2 years: 11 (13.5%)
	3 years: 0 (0%)
	> 3yrs: 7 (8.5%)
Programming	1–2: 67 (81%)
Languages	3–5: 14 (17%)
Known	5–10: 2 (2%)
	Engineering: 49 (59%)
Disciplines	Sciences: 26 (30%)
	Arts: 8 (8.5%)
	Management: 2 (2.5%)

Education Psychology, Psychology, Advertising and public relationships), and Management (Business Administration, Supply Chain Management) fields. All respondents had taken at least one programming language course per the requirements of their majors, and most (81%) claimed knowledge of one or two programming languages. All respondents had worked on group projects as part of their required curricula. No respondents, however, had taken any courses related to software engineering.

The 83 qualifying respondents indicated knowledge of the following programming languages (with numbers of respondents in parentheses): Matlab (45), C (23), C++ (16), Python (14), Java (14), Visual Basic (11), Alice (8), C# (3), LabVIEW (3), HTML5 (3), Ruby (2), Fortran (2), Arduino (2), HTML (2), and others (Objective C, Scheme, Haskell, XML, CSS, Javascript, SQL, A++, ACC, Datalog, Tibasic, R, SAS, SPSS, Mplus, and Stata). Most respondents (89%) had created programs as part of class projects and assignments, and all respondents had taken courses in our computer science department that included group projects. 22% of the respondents programmed to support their research, 8% programmed as part of jobs, and 22% wrote code in relation to hobbies.

The survey data showed that 88% (73 of 83) of the respondents tend to keep copies of their programs. Because we are interested in understanding the processes by which programmers create, find, and manage variants, we consider only these 73 respondents in the analyses that follow.

The survey results allowed us to define several design requirements for supporting variation management (see Table 2). We used a rigorous approach to extract the design requirements. This process was performed by the first and second authors iteratively, who used inductive and abductive reasoning to extract the requirements. (Inductive reasoning is a process by which specific observations with strong evidence of truth are used to make broader generalizations. Abductive reasoning is a process of deriving a plausible explanation or diagnosis from an observation.)

Table 2
Design requirements.

No.	Design requirements emerging from survey
DR#1	*Allow users to understand similarities between variants.*
DR#2	*Allow easy access to author information.*
DR#3	*Keep close-proximity variants together.*
DR#4	*Include program output as a way of representing variants.*
DR#5	*Provide direct access to documentation and source code.*
DR#6	*Include metadata such as filenames, popularity, creation and update date information for variants.*
DR#7	*Include correctness and efficiency information for variants.*
DR#8	*Include information about features supported by variants.*
DR#9	*Allow easy access to variants and their relationships.*

We used these requirements to design our variation management support for end-user programmers. We refer to the *n*th DR in the table by "DR#N" in the following discussion.

3.3. How and why do end users share variants?

We wished to understand how and why end users share variants. Thus, we asked the respondents how frequently they share their code with friends or work in teams. We also asked them how frequently they use online resources or borrow code from friends when programming. Results for these questions are shown in Fig. 2.

Personal copies: Most respondents reported creating variants for personal use; 83% indicated that they reuse their own code at least "sometimes" (mean: 3.41, SD: 0.88). Almost 50% of the participants indicated, however, that they "rarely" or "never" share code with others (mean: 2.70, SD: 0.98). This supports the notion that end users typically create programs for themselves and that they keep variants of their code while opportunistically coding [10].

Code sharing: At least 73% of the respondents indicated that they at least "sometimes" work in teams and share their code with team members (mean: 2.73, SD: 1.08). Almost 50% of the respondents indicated, however, that they "rarely" share code with friends (mean: 3.53, SD: 1.04). The fact that our respondents are university students, and typically discouraged from sharing work products, may play a role here.

Online resources: At least 78% of the respondents indicated that they "sometimes" use online resources such as templates, examples and tutorials when seeking information on program variants (mean: 2.71, SD:1.12).

Improving code: To learn *how* end-user programmers create variants of their programs, we gave participants a set of seven programming situations in which users might keep copies of their code and asked them how often these applied to them.

The results show that at least 82% of end users create variants at least "sometimes" when they reuse or improve their own code. 60% of the respondents indicated that they create variants at least "sometimes" in other situations.

Since end users do create variants, these results indicate that similar programs might exist in online repositories. Past work [66] has shown this to be true. This suggests that providing a way to indicate code similarity may help end-user programmers find variants. This also motivates the provision of similarity information in the design of variation management systems (DR#1). Even though users may not create programs explicitly to share, they do reuse one anothers' code; this suggests the potential utility of tagging programs with author names while keeping variants (DR#2). Finally, the fact that end users often use examples and templates, and numerous similar variants exist in repositories, motivates a design requirement that favors keeping similar variants in close proximity to each other in views provided by variation management systems (DR#3). This is especially important, since recent studies have shown that finding a variant among others is challenging [56,65].)

3.4. How do end users find variants?

We also wished to understand how users find specific variants.

Finding one's own variants: Sophisticated version control systems allow extraction of past versions based on creation dates and tags, but these are often difficult for end users and are rarely used [29]. End users

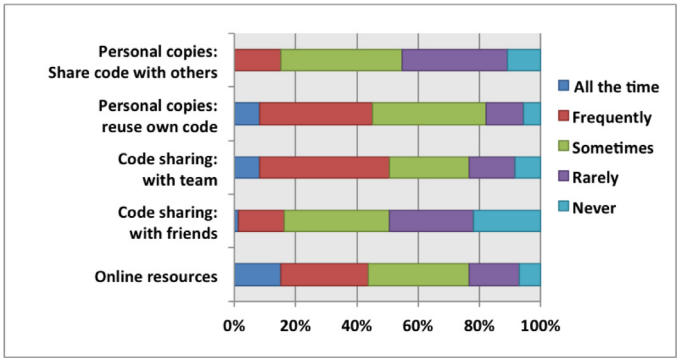


Fig. 2. Factors related to sharing of variants.

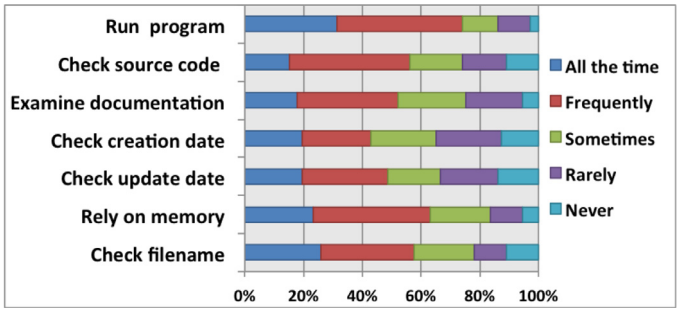


Fig. 3. Finding one's own variants.

program in a way that is broadly characterized as implicit, unplanned and opportunistic [32]. We wished to elicit the attributes of variants that end users actually desire given their programming styles.

Fig. 3 shows the survey responses for different methods respondents used to find their own variants. At least 85% of the respondents "sometimes" resort to executing programs to find a particular variant. Hence, program output is an important attribute to consider in providing access to variants (DR#4). The next most common approach involved relying on memory; at least 82% of the respondents reported at least "sometimes" doing this. This motivates the need for a variation management system that provides easy access to information about variants and their relationships (DR#9). A third approach involves examining documentation (at least 78% of the respondents "sometimes" did this), and a fourth approach involves observing program source code, (at least 77% "sometimes" did this). These motivate the provision of simple, direct code and document access from within a variation management tool (DR#5). Filenames, creation dates, and update dates were reported to be used at least "sometimes" by between 62% and 78% of the respondents; hence these should also be supported (DR#6).

Finding variants in online repositories: We asked similar questions to learn how end users find other users' variants in online repositories; Fig. 4 summarizes the responses. The responses are similar to those in Fig. 3, but with more emphasis on examining documentation and less on checking source code. Additionally, at least 67% of the respondents indicated that they check the popularity of source code (mean: 3.04, SD: 1.79) "sometimes", and at least 60% of the respondents indicated that they "sometimes" consider creation and update dates; hence, these should be provided in a variation management system (DR#6).

Naming conventions: We also asked the respondents what naming conventions they use to keep track of variants. Indicating working or non-working copies (mean: 2.74, SD: 1.37) and including date of creation (mean: 2.90, SD:1.44) were popular conventions; at least 60% of participants use these at least "sometimes" (Fig. 5). Other naming

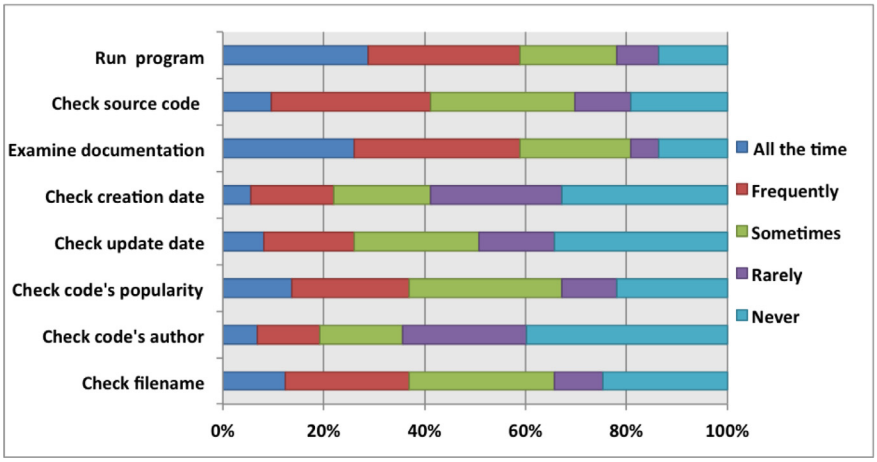


Fig. 4. Finding variants in online code repositories.

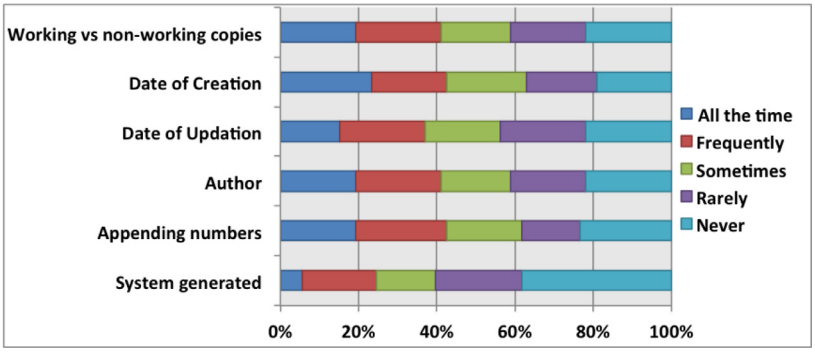


Fig. 5. Factors encouraging naming conventions of variants.

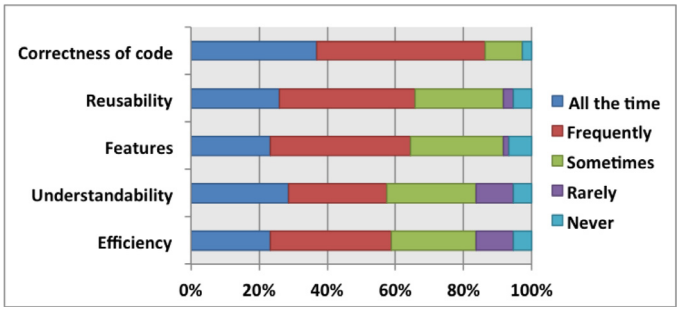


Fig. 6. Factors explaining selection of variants.

conventions such as appending numbers, author names or update dates, or retaining system-generated numbers, were less commonly used.

Selecting variants: Fig. 6 provides data on variant selection. Over 98% of the respondents noted that they at least "sometimes" consider code correctness when selecting variants (mean: 1.82, SD: 0.84); this motivates the inclusion of information on correctness (DR#7). Respondents (92%) indicated that reusability is a factor they consider at least "sometimes" (mean: 2.22, SD: 1.04). Providing information about similarities between variants may help end users identify variants to reuse (DR#1). Further, 92% of the respondents also indicated that they at least "sometimes" evaluate features (mean: 2.27, SD: 1.06), which motivates DR#8. Finally, 82% of the respondents indicated that they consider the understandability (mean: 2.36, SD: 1.17) and efficiency of a variant (mean: 2.40, SD: 1.13) to be useful. Providing direct links to source code and documentation (DR#5) should help with understandability, and DR#7 should enable users to assess efficiency.

3.5. How do end users manage variants?

We also wished to understand how end users manage their variation space.

Existing practices: Fig. 7 shows responses of the respondents regarding the solutions they use to keep track of variants. Most respondents relied on memory (at least 90% indicated at least "sometimes"). The use of configuration management tools is reported by 78% as "rare" or "never". This suggests that domain-specific *version* management systems might help end users; a result noted in prior work relative to scientists [29]. However, since end users emphasize the speed and ease of programming over program robustness and maintainability [10], variation management support must be automated and seamless, so that they need not follow explicit steps to manage variants.

Sources of frustration: We also asked the respondents whether they keep track of changes and whether they find this frustrating. Fig. 8 shows that 82% of the respondents (mean 2.63, SD: 0.94) at least "sometimes" do keep track of changes and that the process is indeed frustrating for them (mean: 2.66, SD: 1).

These findings suggest that a proposed variation management system for end users should make it easy for users to access information on variants and the relationships among them (DR#9).

3.6. Limitations

Our survey drew on a population of college students, so we cannot claim that our results will generalize beyond them. Still, our participants do represent a relevant population, and they had created programs in a wide-variety of programming languages, including many

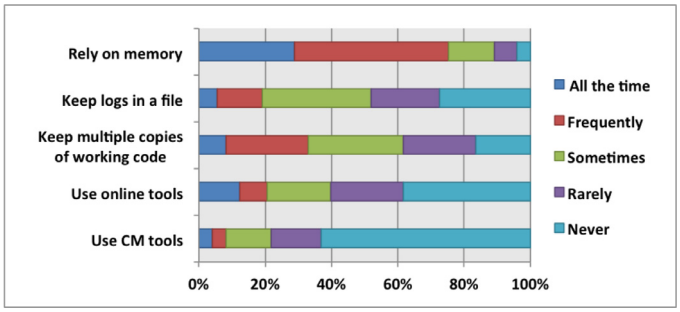


Fig. 7. Solutions for keeping track of changes.

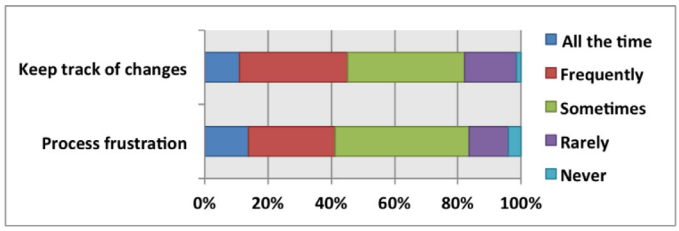
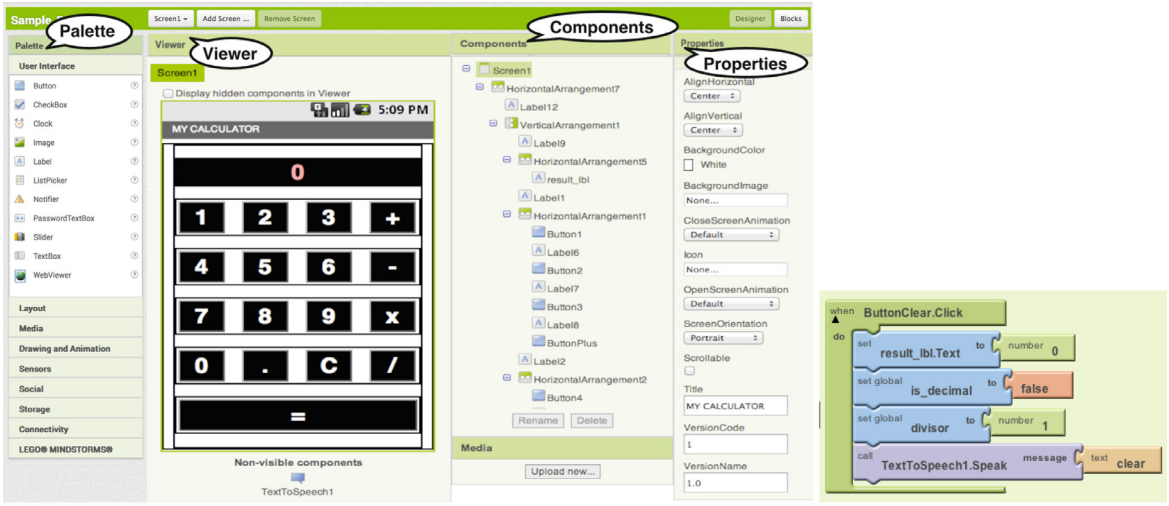


Fig. 8. Sources of frustration.



(a) Design Editor. (b) Block Editor.

Fig. 9. App Inventor Design Editor and Block Editor showing a Calculator app.

that are popular with end users.

Phrasings of survey question can inherently embed researcher bias. To mitigate this, the first author carefully designed the survey by referring to a survey design framework [57] and using terms that end users typically use, which were identified from the pilot interviews. Nonetheless, our survey questions may not fully represent our theoretical concepts and could have been ambiguous to some participants.

Finally, we proposed design requirements by drawing inferences from the survey results and user study feedback; therefore, these may be subjective. However, to mitigate these problems, the first and second author iterated over the design requirements to determine how well they matched the results until they reached consensus.

3.7. Summary

The key results of our survey are that end users do create variants, and they do so frequently, but they do not use tools to track changes or manage their variants. Respondents relied primarily on memory to manage variants. Correctness, author names and similarities among features are attributes used by end users while searching for specific variants. When attempting to find specific variants, respondents considered program output. Therefore, we hypothesize that providing automatic support for variation management can provide features that end users can rely on, and improve their ability to access, assess, and use variants.

4. Approach: AppInventorHelper

Our online survey helped us collect general design requirements for use in supporting a variation management system for end-user programmers. To help us explore the potential benefits of such a system the first and fourth authors implemented a variation support prototype in the mobile app creation environment, App Inventor. In this section we describe the App Inventor environment, and the support we provide.

4.1. App Inventor

The App Inventor environment [1] supports the creation of mobile applications (“apps”) for Android platforms. App Inventor is a popular end-user programming environment. As of 2015, the App Inventor community included nearly three million users from around 195 countries; over 100,000 active weekly users have used App Inventor to build over seven million android apps. The resulting apps are used variously by formal and informal educators, government and civic employees and volunteers, designers and product managers, hobbyists and entrepreneurs, and researchers [54].

We selected App Inventor because it is open source software, and it allows users with any level of programming background to create apps. Moreover, users can share their apps with other App Inventor users on a community site called the App Inventor Gallery. Further, as noted earlier, App Inventor is popular with the sub-community of end user programmers (i.e. students) we used in our survey and user study.

App Inventor provides a graphical interface that allows users to drag and drop components to create software applications (referred to in the App Inventor context as “projects”). Each project consists of a design (interface) and blocks (program logic). App Inventor provides Design and Block Editors to allow users to create these.

The App Inventor Design Editor consists of components for designing the interface of an app. Fig. 9 (left) shows the interface of a calculator app created in App Inventor. The Design Editor window consists of four sub-windows. The Palette window makes components related to user interfaces, layout, media, drawing and animation, sensors, storage, connectivity, and Lego Mindstorms available to users. After a user drags a component into the viewer window, they can modify the component’s look and feel via the Properties window. The Components window contains a list of all components present in the Viewer window.

The App Inventor Block Editor (Fig. 9 (right)) helps users create program logic for apps. It consists of built-in and custom blocks that are

used to design components. Users combine blocks to create program logic. Users can also use parameters to tailor blocks for specific apps.

Users creating apps can change their design or blocks without compiling or running the code. Once an app has been created it can be tested on an Android phone or an emulator. After testing an app, a user can deploy it by packaging it into a file (.apk) or by generating a barcode. The barcode can be scanned or typed in on a phone.

App Inventor users can reuse their own apps by using "save as" or "checkpoint" commands. When using "save as" a user begins with a new project, whereas when using checkpoint the user continues to work on the original project.

4.2. AppInventorHelper

We now discuss the design of AppInventorHelper (AIH), and present a user scenario that illustrates our design elements. We then provide implementation details about AIH.

4.2.1. Design decisions

End-user programming environments largely do not represent prior versions or alternative exploration paths as branching histories; and end users who tend to program opportunistically are forced to rely on their memory to reason about alternatives. While the web interfaces provided in project management systems (e.g., GitHub) are leading to the use of version control systems outside of code management, they still require users to learn specific commands, concepts, and processes. Cloud based applications (Dropbox, Google Drive) are easier to use, however, they do not allow variation management operations of the sort that are needed. Our survey results and pilot interviews revealed that end users typically rely on ad hoc processes and their memory to keep track of the variants landscape.

Therefore, our overall goal was to design a variation management system that renders the process of working with variants seamless and intuitive by making information about variants and their relationships easily accessible (DR#9).

Our survey results helped identify the attributes that end-user programmers assess when selecting variants, from which we created an initial set of requirements (Table 2) for providing variation management support for end users. We then sought feedback from four LabView programmers (senior Ph.D students in Engineering) about the requirements and the visualization support. We used LabView programmers as participants, because LabView also uses a block-based interface and these programmers created and managed variants as part of their day (research) jobs.

We performed four evaluation iterations, in which the first author, assisted by another member of our research staff, conducted cognitive walkthroughs and interviews, with the same set of four end users.

In the first iteration we evaluated paper prototypes of four different visualizations representing the evolution of programs. These visualizations included a network view, a list view, and two tree views. The final tree view (the one on which the final prototype was designed) was based on the ancestry tree view design, but included aspects from the network view.

In the second iteration we identified the information (e.g, filenames, snapshot of output) that would be useful to present. Our third iteration helped us select the features needed for app profiles (meta-data about apps). Our final iteration suggested some esthetic improvements to the prototype. Our application is just one implementation in one particular domain, targeting our population, but similar visualizations could be provided for different domains.

Unit of variation. We wished to select features of variants that are abstract enough to differentiate variants from each another. These features can be based on structure or behavior. App Inventor is a visual programming language in which apps are created by connecting components and defining their properties. This gave us three options to consider as units of variation; namely, components/blocks, properties/parameters, or connection information. Parameter values and connection information do not capture complete information about an app. Parameters capture only configuration information and connections capture data flow information about a program. Components, in contrast, contain functional information related to program behavior that is static and does not change. Moreover, in our previous formative studies, a similar abstraction level (modules) helped participants reuse temporal variants [37,38,40]. Finally, components and blocks are similar to functions in textual languages, and thus, using these as basic units may allow the work to be extended more easily to other languages. For these reasons, we chose components as units of variation. Note that while we keep track of all code changes, we decided to present our visualization based on component changes; then, for each component, the user can drill down and find details about the changes.

Visualization support. Another important choice involves the level of abstraction to use to support visualization of information on variants. We decided to support a coarse-grained view for visualizing variation over space, to help users view all the variants of a program in a repository. We organize the information on variants hierarchically and as a graph view, because hierarchical approaches for organizing information based on context are found useful by users [30]. We also decided to keep all information in close proximity with other variants based on context (DR#3). This may help support "orienteering" approaches for finding information based on context and origin, that are also found useful by users [67]. Figs. 10 and 11 provide examples of the visualization support we provide; these examples are discussed further in the following scenario.

4.2.2. Relating variants to user experience: a scenario

To further illustrate AIH, and to illustrate the type of user experience we intend AIH to support, we consider a scenario in which an end-user programmer interacts with our prototype in order to create, find, and manage variants. In presenting the scenario, we refer to relevant design requirements (Table 2) and relate these to visualization features provided by AIH.

Our scenario involves Shelley, a junior science student at a university, who has a classmate, Ted, who is visually impaired. Ted's Math and Physics classes require the use of a calculator to solve problems. Shelley has some programming background and she has used App Inventor to create Android apps. She decides to create a calculator app that Ted can use on his Android phone.

Creating and managing variants. Shelley's friend Erin has already created an app (named "EasyCalculator") using App Inventor that takes text input and performs basic calculator operations (add, subtract, multiply and divide). Shelley looks at Erin's app, and with AIH she is able to see that her app has already been extended by other people. Fig. 11A shows various app variants extended from Erin's EasyCalculator app; nodes represent apps (variants) and edges represent copy relationships between apps (variants). All copied (app) variants are displayed to the right of the parent app from which they have been derived. Node colors represent authors (DR#2). As can be seen in Fig. 11A, the Graphview shows that another author, Terry, has already created a variant of the calculator app (named "TalkingCalculator") for visually impaired people, and his app converts text to speech. Shelley clicks on the node, which opens Terry's app in App Inventor (DR#5), allowing her to assess the "speech" block(s) (DR#8).

Shelley copies (using "save as") Terry's app and changes the interface design to try three different look-and-feel variants; namely, those of a typical calculator, Android, and Windows 8. Fig. 11B shows the four app variants that are thus created ("PocketCalculator", "WindroidCalculator","VisualCalculator", and "VoiceCommandCalc") from "TalkingCalculator". The Graphview allows Shelley to see these designs in relation to their parent and in close proximity (DR#3).

Shelley prefers the user interface with the Android look and feel, and gives it to Ted who begins to use it to solve problems in his classes. In this manner, AIH has helped Shelley opportunistically program.

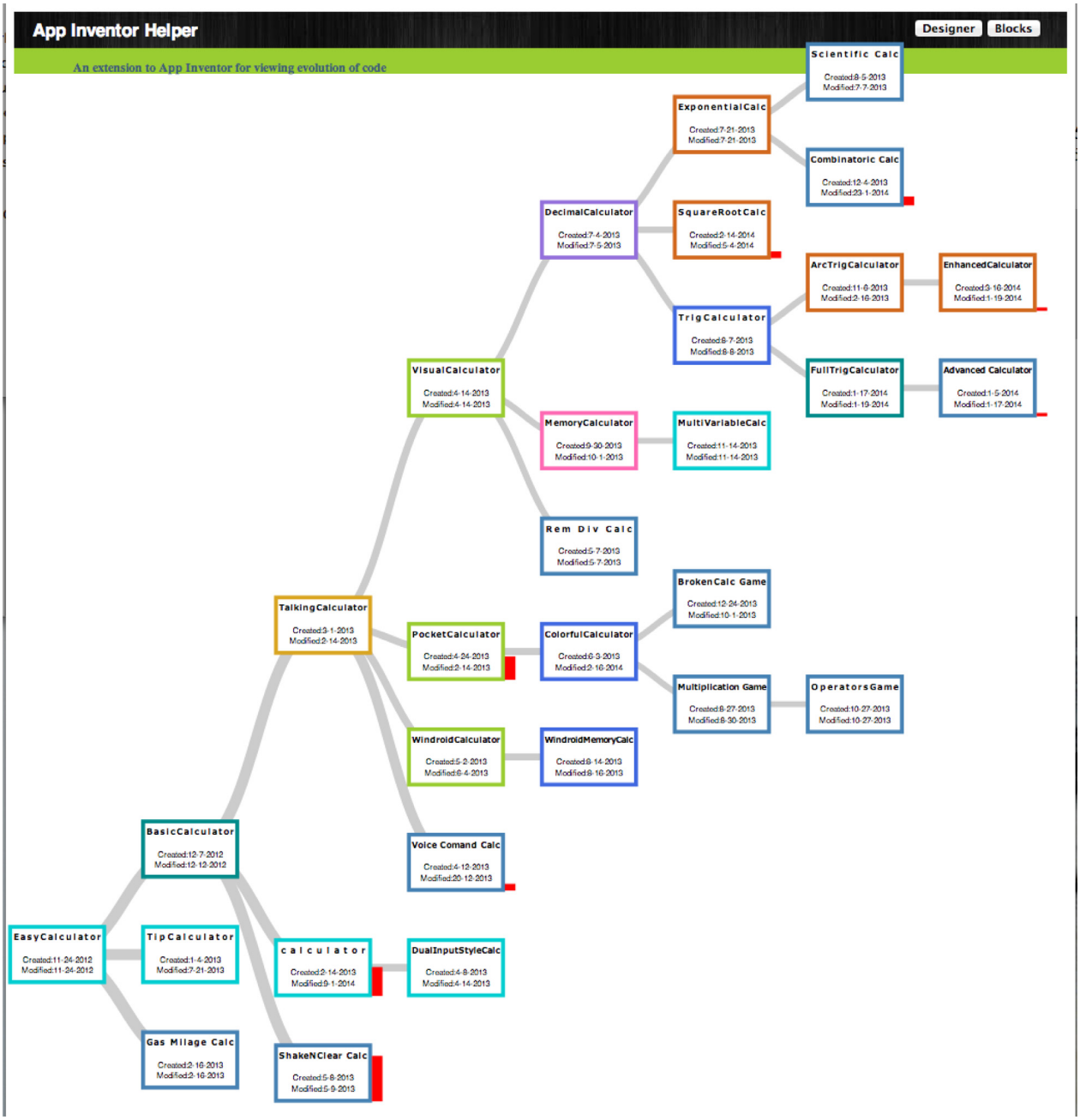


Fig. 10. AppInventorHelper (AIH) interface for providing variation management support for end-user programmers. Nodes in the graph represents variants of an app and edges represents the copy relationship between two variant. Apps created by the same author have the same boundary colors. The red bars next to some of the nodes indicate error status (percentage of errors in a project). The thickness of the gray edges denotes the amount of similarity of source code between two nodes.

While creating the app (variants), whenever she reaches a state in which she finds errors, she can remove the portions of code she has newly added, and AIH keeps track of the intervening states of her program. As such, AIH also helps her debug her programs into existence. Both of these programming styles are encouraged because AIH retains copies of her variants automatically, so that she need not rely on memory (DR#9) or other tools to manage them.

Seeking variants. After several weeks, Ted's math course becomes more advanced and involves trigonometric calculations. Shelley offers to extend her app to include trigonometric functions. Using AIH, she discovers that her app has already been copied and extended by several other authors creating more variants. AIH helps her understand the directions in which her app has evolved (Fig. 10).

Shelley can also determine the similarity of her code to that of other app (variants) by looking at the thickness of edges in the Graphview (DR#1). When Shelley hovers her mouse over the projects, she can assess the similarity of her code to that of parent and child projects. For example, Fig. 11C provides a view of the similarity of "TrigCalculator" to its parent and other variants. Thicker edges signal greater similarities between project variants, and the maximum thickness of an edge can be equal to the height of its incident nodes, indicating 100% similarity between projects.

Additional sources of information are also available. Nodes also contain project names, and project creation and update times, and Shelley can see which authors have copied code by considering node colors. Shelley can also retrieve design and block information by selecting the Designer or Blocks buttons in the AIH interface.

AIH also helps Shelley find variants of the calculator app that are

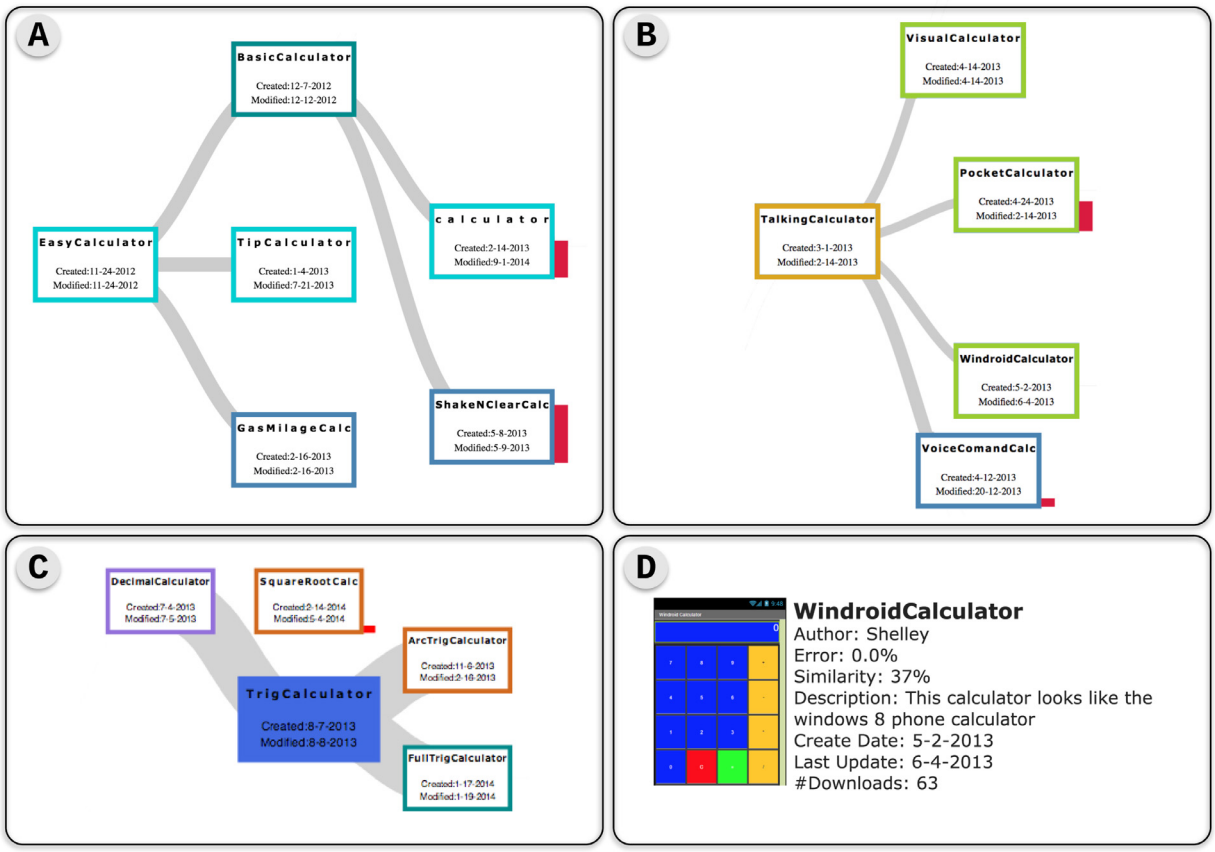


Fig. 11. AIH Graphviews. (A) A snippet of the Graphview as seen by an end user. (B) The four variants—"PocketCalculator", "WindroidCalculator","VisualCalculator", and "VoiceCommandCalc"—that were created from "TalkingCalculator" app. (C) Hovering over "TrigCalculator" shows the similarity between it, its parent ("DecimalCalculator"), and its variants ("ArcTrigCalculator" and "FullTrigCalculator"). (D) An app profile for the "WindroidCalculator" app variant.

not known to contain errors, by considering the red (error) bars (DR#7). The height of the red bar beside a node represents the percentage of errors known to exist in the project, where error percentages are calculated in terms of the number of errors or messages occurring in blocks, divided by the number of blocks. Shelley can see which variants are known to be faulty, and subsequent cloned/extended variants into which the faults may have been propagated if the child project was created before the parent project was updated.

Selecting specific variants using app profiles. Shelley can view complete app profiles by letting her mouse hover over specific projects. Fig. 11D shows an example of such a profile for the "WindroidCalculator". App profiles provide complete information on the project, including name, description, output, percentage of similarity with copied apps, percentage of errors, popularity, author of the app, and when the app was created and updated (DR#4 and DR#6). Profile information helps Shelley obtain a finer-grained view of each project, and she can search for a desired variant by using that information. Ultimately, she can select the most desired variant by a mouse click, open it in a separate window, and modify it (DR#5).

4.2.3. System architecture and usage

Fig. 12 presents the system architecture for AIH. The architecture consists of a Server with a variation plug-in, and a Client with a visualization plug-in. The Server and Client interact with a database to save and retrieve information. The Client interacts with the Server to run and render information about an app. The data exchanged is XML or JSON data.

Variation plug-in. To create variation management support we modified the source code of App Inventor. The contents of all variants

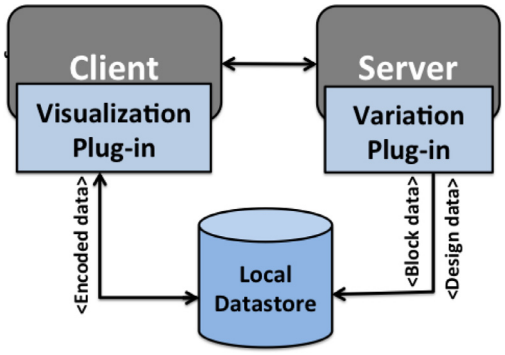


Fig. 12. System architecture for AppInventorHelper (AIH).

are kept locally in the database, where we retain both design (UI) and block (program logic) data for each app. JSON (for design) and XML (for blocks) contents are captured and kept in the database to save information on variants, and each variant's information includes the set of components/blocks added to the canvas by the user. We keep track of variants by intercepting copy events ("save as" or "checkpoint") and recording them in the database. Working and non-working variants are identified by intercepting warnings and error messages captured by App Inventor.[5]

Visualization plug-in. Our visualization plug-in is initialized from

[5] The visualization for AIH is available at http://www.cse.unl.edu/~ressarma/Visualization/performancetree.php.

the client side. The client requests encoded data (JSON) from the server in order to display the visualization in AIH. The visualization code uses the d3.js libraries, which are written primarily in Javascript and SVG. The server side plug-in is written primarily in PHP. The Graphview utilizes the child and parent relationships obtained from the database.

We consider similarity based on how much an application resembles another application based on code – including features and the user interface. Hence similarities between parent and child nodes are calculated differently for block views and designer views. We calculate the similarities by "diffing" the programs of the two variants. The similarities are calculated by "diffing" XML files for the block view, and JSON files for the designer view.

Usage Scenario. To further illustrate the way in which end users and the system interact we return to our usage scenario. Any time Shelley creates, selects, or deletes an app from the AIH Graphview, this triggers the AIH visualization plugin to interact with the variation plugin. Given information on Shelley's action, the variation plugin saves, accesses, or updates the information related to the app in the local database. Each save action also causes the plugin to save versions of the code in form of XML data (for the design) and JSON data (for the blocks in the local database). When Shelley selects an app profile from AIH, the visualization plugin accesses the information about the app directly from the local database to render it on the screen.

5. User study design

To better understand end users' usability experiences with AIH and discover how we can improve it, we performed a formative lab study[6], guided by the following research questions:

RQ1: how do end users find, evaluate, and reuse variants with and without the support of AIH?

RQ2: how usable is AIH in supporting end users in managing variants?

5.1. Participants

To recruit participants we sent an email to several departmental mailing lists at our university; 21 students responded. We screened these responses in order of arrival, excluding those respondents who had less than three months experience with LabView, and those who had any formal training in software engineering methods (course work or professional software development experience). We required LabView experience for several reasons: 1) being a visual programming language it is similar to App Inventor, 2) it is the only visual programming language used in the curriculum of our University, and 3) it has the advantage of being popular with engineers and scientists for creating sophisticated programs. Our participants were majoring in electrical engineering, mechanical engineering, and physics, and their ages ranged from 19 to 40. None of the participants had formal training in software engineering methods, but all had some experience (ranging from three months to four years) with visual programming languages. None of the participants had prior experience with App Inventor. Finally, since gender differences have been shown to influence end-users' interactions with software, [5,11,12] we also ensured that we had selected an equal number of male and female participants (five each). All participants were compensated with $20.

5.2. Study setup

We used a think aloud protocol [46,62], asking participants to vocalize their thoughts and feelings as they performed their tasks. This helped us gain insights into the participants' thought processes, and barriers they faced while exploring, understanding and selecting variations. This approach required us to administer the study to participants on an individual basis with an observer – in this case, the first author.

We performed the study in the Usability Lab of the Computer Science Department at the University of Nebraska-Lincoln. Participants were first asked to complete a brief background questionnaire; this was followed by a tutorial of approximately ten minutes on App Inventor. The tutorial also included a short video of a think-aloud study so that users could understand the process. After participants completed the tutorial, we asked them to create a small sample project with App Inventor to give them hands-on training in creating a mobile app and familiarity with App Inventor in general. The participants were encouraged to ask questions and ask for clarification during this phase of the study. We also provided them with web links for App Inventor documentation.

Next, to provide additional experience we asked participants to complete an initial task in the App Inventor environment itself, without the use of AIH. This *base case* task involved using keywords to search the App Inventor gallery for apps for painting, and extending one such app by adding a new feature. The base case helped us understand the difficulties faced when participants do not have variation management support in this particular domain. This experience also allowed participants to provide more insightful feedback on AIH and its features, because they could compare the two types of experiences.

We next moved on to the experiment tasks, which were designed to help us understand how participants employed AIH to identify relevant variants and what environment features affected their selections. In AIH, participants used the visualization support to look for variants (unlike the base case which involved keyword searches).

We set out initially to find existing families of variants in online repositories for our study, because we did not wish to create entire variation families artificially. We had discovered that most popular App Inventor apps occurred in families containing 15–30 members; thus, we decided to utilize base case and experiment tasks that involved 30 apps. We ultimately chose two families of apps (one family of calculator apps and one family of paint apps) that had 16 and 32 variants available, respectively. We downloaded these variants, and then asked the fourth author (an undergraduate research assistant) to create additional new variants for each of the families until 30 of each were available. The fourth author performed this task by observing similarities in variants' code, and then based on those, extending existing variants to include additional features. The resulting sets of variants were of diverse complexity and some of them were large.

Participants used the UI (Fig. 11D), provided by AIH, to view variants of the apps. In the base case, users had to search over the entire repository (the 30 apps appeared when appropriate keywords were used), whereas in AIH the variation subset was presented. (This is not an artificial device: the very presence of variation management support allows sets of related apps to be automatically identified and considered in isolation.)

Each participant completed two tasks (see Section 5.3). Since our primary goal was to understand the users' system usage behavior rather than their task correctness or the time required to complete tasks, we provided hints to the participants if they became "stuck" while adding features to the app. We did this because we did not want to penalize our study participants for reasons of their unfamiliarity with App Inventor programming. After the participants completed each task we followed up with an interview to understand why they selected specific variants and what environmental features helped or hindered their search and encouraged their selection.

We audio recorded each session and logged the users' on-screen interactions using Morae,[7] a screen capture system. The total time

[6] All supplementary documents related to this study can be found in the Appendix of [36].

[7] Morae: http://www.techsmith.com/morae.asp.

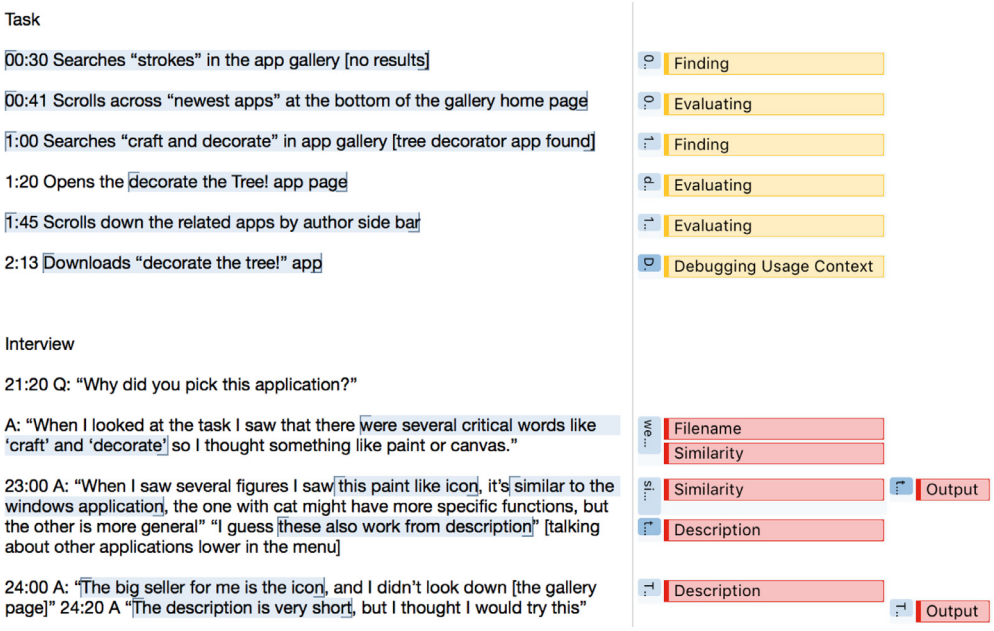


Fig. 13. Coding examples for analyzing a task and an interview transcript.

required for completion of the study per participant was approximately 1.5 h, which included an average of 60 min for task completion.

After participants completed all tasks, we administered an exit survey to obtain additional feedback. The survey consisted of both closed and open-ended questions about AIH. We used the Microsoft Desirability Kit to measure participants' opinions of AIH [6].

5.3. Tasks

As just explained, our two tasks involved a paint app (base case task) and a calculator app (experiment task). The primary goals of the first task were to provide additional experience (after the 10 minute tutorial), serve as a base case (without the use of AIH), and allow participants to later compare experiences with and without AIH (and thereby provide insightful feedback on AIH). The primary goals of the second task (calculator) were to understand how users search and select variants, understand users' system behavior (not task correctness or efficiency), and understand the strengths and weaknesses of AIH in helping end users select and reuse existing examples. In both cases, our participants were asked to search for similar apps and then add features to the apps.

Paint task: In this task, participants were asked to do the following: “Create an app to craft and decorate using mouse strokes. The app should also allow users to select strokes of different sizes. Add a feature to this app for erasing the canvas when the phone is shaken.” Participants were asked to search for such apps using the App Inventor Gallery.

Calculator task: In this task, participants were asked to do the following: “Create a talking calculator, which speaks aloud all numbers and the operations being performed on the numbers. In order to do this task, you can search for calculator apps that perform basic operations, e.g., add, subtract, divide and multiply. Extend this app to include the log function.” To perform this task, participants had to use the functionality provided by AIH.

5.4. Analysis methodology

We transcribed all verbalizations and actions performed by the participants. We analyzed transcripts by coding instances of reuse (finding, evaluating and debugging a usage context), navigation order (forward, backward), exploration (backtracking, alternate choices), and variant attributes (author, filenames, errors, similarity, output, and date). Two researchers collaborated on the coding and an inter-rater reliability was calculated between the two coders. Once the coders achieved more than an 80% inter-rater reliability agreement (the final round reached 87% agreement) on 20% of the data, each researcher independently coded the remaining transcripts. Note that according to Landis and Koch [44], reliability coefficients of 80–100 are considered “Almost Perfect” for determining the rigorousness of coding. The semi-structured interviews with participants were used to analyze behaviors observed and elicit feedback about how users viewed App Inventor and AIH. These interviews were transcribed and coded similarly. Fig. 13 provides an example of how the coding was done on the transcripts in the cases of (1) analyzing a task and (2) analyzing a retrospective interview.

5.5. Threats to validity

All empirical studies have threats to validity that should be considered when interpreting results. Threats to external validity arise from the fact that we considered only two tasks based on only two types of projects. Moreover, we considered only a small number of university students in the survey and study. Our study was intended to be a formative study, however, and therefore we used convenience sampling. Moreover, App Inventor has a large user base of K–12 and university students [71]; thus, our survey and study do address a significant population of actual App Inventor users. Furthermore, prior studies have found that students and professionals can be equivalent “when their knowledge, skills and experiences fit within the tool's intended user population” [33].

Other external validity threats arise from the fact that the variants given to the participants represent only a small sample of possible variants. We limited the numbers of variants considered to 30 since this number was typical of variants found in some of the most popular projects in the AppInventor Gallery. Moreover, some of these variants were created by ourselves. Further studies with different populations, tasks, and projects is needed to mitigate this threat.

Threats to internal validity arise from the following factors. Our study design helped us obtain better feedback about the usability of AIH, but it could have led to learning effects as participants moved from

the base case to the experiment task. Our study is meant to be formative (not summative), however, and to focus on the feasibility and usability of variation management support. The base-case task was needed to train participants in App Inventor and to observe the current conditions in which they operate. Since AIH provides additional features (visualization and provenance) an AB comparison would not be fair. We did not want to create a situation in which participants looked for AppInventor Helper features in the base case. Therefore, we did not counterbalance the tasks.

Threats to construct validity include the possibilities that the complexity of our projects was not high enough to allow measurements of effects, and that the projects used for base and experiment tasks may not be comparable in complexity. We controlled for this by performing initial pilot studies on two non-participants and used their feedback to adjust the tasks' complexity.

6. Results

We wish to examine the strengths and weaknesses of AIH in helping end users select and reuse variants in programming tasks. To do so, we analyze the behavior of participants with and without AIH with respect to (1) how they find variants and (2) how they evaluate variants. We also evaluate the usability of AIH through our exit survey and interviews.

6.1. Reuse behavior of users with and without AIH

Reuse is a primary mechanism by which end users create new programs [15]. Rosson et al. [58] identify three primary activities that users perform when they attempt to reuse programs: (1) finding a usage context, (2) evaluating a usage context, and (3) debugging a usage context (see Fig. 14). That is, users first need to find an example that suits their current task context, then they need to evaluate this example to determine whether it matches their requirements within their task context, and then they may need to edit, tweak, and debug the example to perfectly meet their requirements.

These reuse activities occur sequentially, but if any of them are unsuccessful (as per the user), users tend to backtrack to their previous activity. This process of performing activities and backtracking is repeated until the user has completed the task. We use the steps of this process to organize the discussion of our results.

We observed that our participants, when performing both base case and experiment tasks, also performed the three aforementioned activities to find appropriate variants. However, since our focus is on searching and navigating to find variants, we focus primarily on the first two tasks.

In presenting results, we first describe how participants performed an activity in the warm up task, as it serves as a base case task. We then describe how they performed the same task using AIH.

6.1.1. Finding a usage context

Our results show that finding a usage context can be seen as a composition of: (1) understanding the variation space, and (2) navigating through that space.

a. Understanding the landscape of the variation space. To find an appropriate variant a user first needs to know what variants exist that may match their current requirements. In the base case task, participants used the search feature provided by App Inventor. Participants had to rely on the keyword search functionality, which required them to identify appropriate keywords – a difficult process in general [19]. Not surprisingly, half of the participants (five) had difficulties identifying appropriate keywords to use. For example, when participant P2

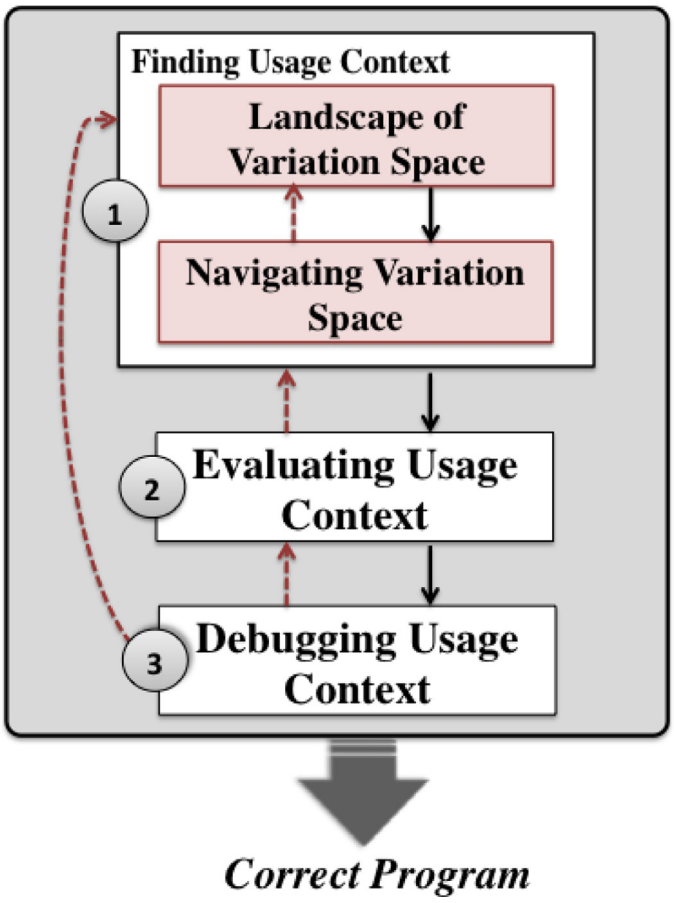


Fig. 14. Reuse activities process flow adapted from Rosson et al. [58].

Table 3
Number of Times an App was selected from the list of results returned with the keyword search. The * shows the Final App selected by the participant to perform the Task. P4 selected from the list of the latest Apps while P5 did not select any App during the study.

Participant	1st app	2nd - 3rd app	4th or more apps
P1	1*	0	0
P2	1	1*	0
P3	3*	0	0
P4	0	0	0
P5	3	3	0
P6	2	1	1*
P7	0	0	1*
P8	2	1	2*
P9	1*	0	0
P10	0	0	1*

searched for "mouse strokes", she received results related to a rodent. These five participants used dictionaries or searched the internet for the meaning of the phrases in the task description to better understand its semantics.

A key step in finding the right usage context involves participants "orienteering" their search; that is, following a series of small steps to reduce the search space [67]. In the base case task, participants performed iterative searches and relied on their analysis of the results to do so. Seven participants used multiple combinations of keywords before arriving at a final list. One participant (P5) who had difficulty in the task asked, *"How do I find applications that are similar [in the list presented]?"*. After evaluating the search results for 7.5 minutes he modified the search keywords to reorienteer his search. Another participant (P4) needed to orienteer his search keywords 14 times before he was satisfied. Eight participants used on average 5.6 keywords while orienteering their search (this does not include participants who selected apps without using keywords). In summary, participants in the base case task spent effort creating queries, evaluating the search context, and orienteering their search.

In the experiment task all participants used AIH, and hence short circuited the search process. They were able to rely on the visualization to understand the usage context of the variation space. Seven participants mentioned that AIH helped them organize information (as per their responses to the Microsoft desirability kit). The graph view of AIH served two purposes. First, it allowed participants to obtain an overview of the variation space and how variants were related to each other through the hierarchical views, which are known to be useful in representing relationships between information [30]. For example, P3 commented: *"It explains more about the applications that have been done and the relationships among them"*.

Second, the graph view helped participants understand the landscape of the variation space by providing information about the provenance of variants, where provenance is defined as an understanding of the chronology of the ownership, custody or location of an object. In exit interviews, eight of ten participants mentioned that AIH was useful for understanding the provenance of variants. The graph view allowed participants to track the origin and development of variants in terms of functionality and changes in authorship. Each participant referred to one or more of these sources of provenance information in their exit interview. For example, P2 commented: *"It helps in knowing the origin of the app... also the authors and similarity"*.

b. Navigating the variation space. To navigate the variation space users browsed through the list of apps and reviewed app names and features. In the base case task, participants browsed through the list of apps returned by their search in the order in which they were presented. Half of the participants tended to remain within the first three search results (Table 3). One reason for this could be that the App Inventor Gallery display page shows only the first three results (as shown in Fig. 15) and users have to scroll to see other results. (When asked why she didn't look at other apps, P2 commented *"Oh I didn't see the scroll thing"*.) Four participants browsed to other apps in the list by scrolling.

In contrast to the base case, when using AIH, participants navigated through the variation space easily using the graph view that was provided. The hierarchical layout and similarity information on edges fostered exploration among variants before participants settled on a variant that matched their usage context. Participants visited on average 9.6 apps (nodes in the graph) before selecting a specific app.

Participants used information on similarity and degree of similarity between apps (edges and edge weights in the graph view, respectively) to focus their navigation and to select an appropriate app. P3 commented, *"I followed the tree..."*. Participants also explored branches and considered the provenance of the variants. For example, P10 commented: *"if neither of the other branches would work [were appropriate] I went back to the Talking Calculator [app name] as the base one"*.

Two participants used information on how many apps an author had used to inform their navigation. For example, P10 selected an app based on its author, commenting: *"I picked it because the guy [author] who created this app [pointing to the selected app] made another five [apps], which is a high number. So I stayed in this area"*.

The ease of navigating forward (toward more functionality) or backward (toward less functionality) between variants, along with provenance information, gave participants flexibility in navigating. Often, instead of tracing the development of a variant (additional functionalities) from the beginning, participants started from leaf nodes

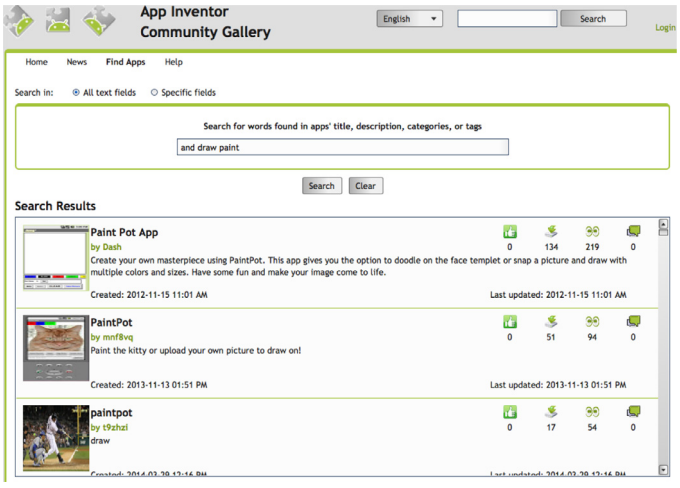


Fig. 15. Linear list of search results in App Inventor Gallery as viewed by participants.

Table 4
Parameter sources.

Parameters	Base case task	Experiment task
Name	search results	node name, app profile
Code	block editor	block editor
Output	emulator	emulator
Snapshot	gif in search result	gif in app profile
Description	search result	app profile
Author	search result	app profile
Popularity	search result	app profile
Similarity	none	graph view
Error	none	graph view

because they deduced that these variants would have the most functionality, and then worked their way backwards. Two participants used forward navigation (parent to child), five participants used backward navigation (starting at leaf nodes), and three used both.

6.1.2. Evaluating a usage context

The second step in reuse activities is to evaluate the appropriateness of an example to the task at hand. After evaluation, if a selected app turns out to be inappropriate then users backtrack, i.e., revert back to finding a new variant by a new search (base case task) or exploring the AIH visualization.

Models of attention investment [7] consider the costs, benefits, and risks that users weigh when deciding how to complete a task. The participants performed such implicit cost benefit analyses when making decisions about whether to backtrack to identify a more appropriate variant or modify a selected app to meet the task requirements.

a. Application characteristics investigated. Participants investigated the following parameters when considering activities in both the base case and experiment tasks:

- the features of an app as inferred from its `name`
- the `code` inspected by opening the block editor
- the `output` when the app is executed in the emulator
- the `snapshot` of the output from the .gif image provided either by the author (base case task) or by AIH
- the `description` of the app
- the `author` of the app
- the `popularity` of the app based on number of downloads
- the `similarity` between apps
- `errors` in an app

Table 4 provides details on how each of these parameters was obtained by the participants. From our observations and corroborated by our exit interviews, we identified the extent to which each parameter was used. Fig. 16 shows the numbers of participants who used each of the parameters. As the figure shows, when using AIH users have a wider variety of characteristics that help them select an application.

The three parameters that all participants in both cases investigated were `name`, `code`, and `output`. The name of the app was the first thing participants considered, and it played a key role in the selection strategy. When asked about his selection strategy, P6 (in the base case task) commented: *"I was looking at the name of the project and the functionality of the project. So I looked for paint programs"*. In a similar vein, in the experiment task, P8 commented: *"enhancedCalculator [app] belonged to the talking calculator [app] so it should speak aloud, and the advanced [app] one should have a function like power similar to the log function"*. Therefore, it is likely that the name of the app is the "least costly" cue available to the user and it was used as a primary factor in evaluating the usage context. Once a participant considered the app to be a close enough match to what they desired, they opened the code and checked its output.

Two other parameters, `snapshot` and `description`, were used to evaluate an app's functionality in a light-weight manner before investigating it in greater detail. The `snapshots` were visible in the base case task as icons next to the app name in the search results. Seven participants viewed this information. In many cases, however, such snapshots could not be expanded or were not present in an app's page. In AIH, in contrast, a snapshot of the output was always available through the app profile, which was available via a mouse hover over the node (variant) in the graph. Four participants used this information frequently. This feature was generally appreciated by participants; for example, P4 commented: *"I like that you can see what the application looks like instead of a logo"*.

Participants in both groups (five in the base case task and six in the experiment task) read the `description` of the apps to understand how they worked, and were frustrated when there were not enough details available. When asked what things he would like to see improved in the App Inventor Gallery, P6 (in the base case task) commented: *"First I would add a better description, people need to see a proper description"*.

We were surprised that other parameters such as `author` or `popularity` were not used often. Participants may have ignored information about authors given that they are not part of the community and not aware of authors' reputations. We did find that the number of similar apps created by an author was used to some extent by participants in the experiment task (6). For example, P2 commented *"Good choices to start off"*, and while explaining his selection he mentioned, *"I saw the basic calculator, I saw that if this person modified these [pointing to*

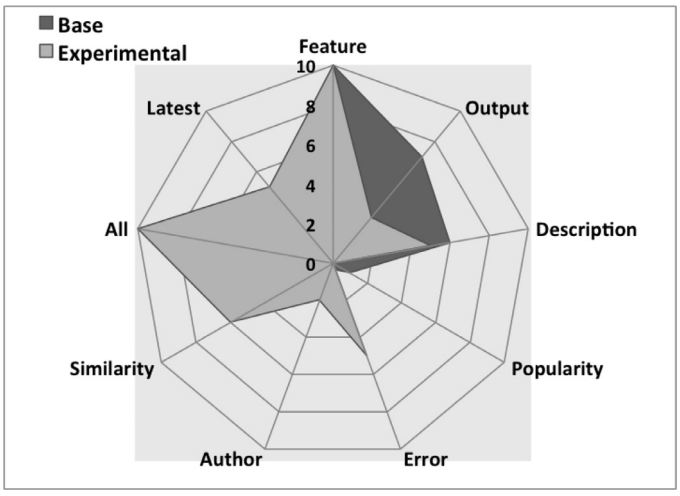


Fig. 16. Distribution of parameters used during experiment and base case tasks. The selection choices increased with AIH.

other children of the application] then they must have added something [here]".

Information on `errors` was used moderately in the experiment task: five participants accessed the error status of an app before selecting it. P9 commented: *"It [graph view] increases the efficiency because others know which part is wrong / how much error"*. It is likely that the error information allowed participants to avoid selecting erroneous apps; P3 commented: *"I don't remember if I checked the error of the code, but I should as I want to do as little programming as possible"*.

b. Backtracking

Participants evaluated the app code and if it was relevant to their task context, they then worked on the code; otherwise, they backtracked to find a new usage context.

Seven out of ten participants backtracked at least once in the base case task. Participants found this task particularly difficult, because the cost of finding an appropriate starting point was high. The participants who backtracked ended up selecting imperfect or erroneous apps, because they began working on the first app that seemed relevant to the task. Selecting imperfect or erroneous apps cost them time and effort. For example, participant P8 selected an erroneous app, and spent 25 min trying to make it work, which in turn introduced further errors. In contrast, AIH helped participants by providing an overview of the "landscape" of the variation space (the degree of similarity among variants), while also providing app profiles (available when allowing the mouse to hover over a variant). These profiles presented the majority of the app parameters (see Table 4) that participants referred to. Since participants spent more effort navigating and selecting a "good candidate" initially, fewer of them (three of ten) performed backtracking. For example, P3 commented: *"I could discard faster the ones that were not useful [during initial selection]"*. Two of the participants who did backtrack, however (P1 and P5), did so multiple times, because they wanted to find an app closer to the one required by the task. These participants initially opened multiple apps and selected among them rather than settling for the first app that they located.

6.1.3. Debugging a usage context

The final step in reuse involves users tweaking reusable components to fit the context of their current task. While reusing code, if users introduced errors or found that the selected app was not appropriate they backtracked, i.e., resumed looking for new usage contexts or evaluating usage contexts.

While not the focus of our experiment, we found that AIH allowed participants to select more appropriate apps as starting points. This helped them save time while editing the variants. We observed that once participants selected an app and began modifying it, they preferred to debug the selected app and make it work instead of going back to navigation. Only one participant (in the base case task) and two participants (in the experiment task) sought to look for a new app when they ran into errors due to their modifications.

6.2. Usability of AIH

We evaluated the *perceived* usability of AIH using the Microsoft Desirability Toolkit and the exit survey.

6.2.1. Microsoft desirability toolkit

We used the Microsoft Desirability Toolkit, developed by Benedek and Miner, to measure users' emotional response and the desirability of AIH [6]. Participants were given a list of 118 adjectives and were asked to select those that most closely applied to their experience with AIH. The participants selected 28 words (on average 2.8 apiece), of which 23 were positive and 5 were negative. The most popular positive words (based on the number of times they were selected by participants) were: Organized (7), Easy To Use (6), Accessible (6), Useful (5), and Time Saving (4). The negative words were mentioned by four of the eight participants. The most common negative word was Simplistic (4). The other negative words were: Too Technical (1), Unpredictable (1), Intimidating (1) and Overwhelming (1).

Table 5
Participant feedback about AIH (1 Indicating "Strongly Believe" and 5 indicating "Don't Believe").

Variable	Mean (SD)
Helps understand app evolution	1.9 (0.99)
Improves app reusability	1.8 (1.03)
Helps in selecting correct app	2.0 (0.81)

As already discussed, the GraphView helped organize variants and their similarities in an easy to navigate form. The app profiles and edges depicting similarity across variants made information about variants easily accessible. The similarity information along with comprehensive app profiles helped participants select appropriate variants and thereby spend less time creating, removing, modifying or debugging variants. It is possible that the AIH visualization is what four participants found to be too simple.

6.2.2. Exit survey

Three of the questions in the exit survey asked participants to provide ratings on a Likert scale from 1 to 5 (1 indicating "strongly believe" and 5 indicating "don't believe") on the usability of AIH. The questions were related to whether users believed that AIH: (1) helped them understand the evolution of apps, (2) improved the reusability of the apps, and (3) helped them choose correct apps. Table 5 summarizes the results, showing the mean and standard deviation of the responses of our participants. The data indicates relatively strong beliefs that AIH helped in all three categories.

7. Discussion and implications

Here we discuss our study results in the context of end-users' behavior while using AIH. We then discuss implications for the design of environments to support variation management.

7.1. End-user behavior

7.1.1. Evaluating the variation space

Laying out variants in the graph view based on parent-child relationships and aggregation of functionally similar variants provided a hierarchical view. Hierarchical views help users create mental maps of relationships [30]. In the absence of a hierarchical view, participants (in the base case) with linear lists explored only small portions of the variation space. Therefore, hierarchical views that present information on relationships and similarities across variants can complement the linear search results that are typical in large online repositories. In the future, we can use existing techniques like ECCO (Extraction and Composition for Clone-and-Own) to select desired features, find software artifacts to reuse, and then provide hints to users completing the task manually regarding which software artifacts may need adaptation [22].

7.1.2. Using provenance data

The participants largely used provenance information when navigating the variation landscape and choosing variants to reuse. The participants used information such as creation date to identify the latest variant, author information to identify similar variants, and the parent of the variant when making their selection. The similarity relationship between variables was the most popular. This suggests that in a variation space, the key thing that users look for is how variants emerged and their similarity with others in the space. Such provenance information and visualization for variations can potentially be also used to represent information common among other projects.

7.1.3. Assessing quality

A key step when evaluating a variant involves assessing its quality. Five of our participants heavily used the error indicator provided by AIH. However, in the presence of the indicator, they did not bother to assess the code itself. They also did not consider the number of downloads as a metric–a possible signal of the quality of an app (popularity)– perhaps because it was the last piece of information provided and lacked explicit visual markers on the graph view.

Finally, in addition to provenance and quality information, participants used the activity of an author to determine the quality of apps. Some participants focused on the set of apps made by the most prolific authors. This indicates that quantity and frequency of contributions are treated as a proxy for quality, a fact that has been observed in other online peer production sites such as GitHub [70].

7.1.4. Risk aversion and exploration

All participants performed implicit cost-benefit analyses when deciding how much of the variation space they would explore and which variant they would use as their starting point. AIH made it easy to access information on similarities between and details of variants. Further visual cues allowed participants to quickly evaluate whether the features they needed were available in a particular app. Finally, accessing the source code of an app was possible by simply clicking on a node. Lower cost access to information and ease of navigation meant that when using AIH, participants were less risk averse, and they: (1) explored a large portion of the variation space, (2) evaluated many apps, and (3) if dissatisfied with a selected variant went back to exploring variants. In contrast, all individuals explored less in the base condition.

7.1.5. Selection strategies

Participants used different strategies when navigating and evaluating variants. Some individuals began their explorations from leaf nodes, believing that removing features would be easier than adding new ones. However, removing features and logic from existing code can be difficult as there may be dependencies involved, and the impact of removing a section of code could be felt in other modules. Given that the language involved in our case is visual and modular, this was not an enormous problem; however, participants did run into trouble when they removed blocks or changed parameters inappropriately. This behavior indicates that environments might need to provide users with "selective" undo – that is, automated support through which a user can remove a feature, with the relevant pieces of code then being automatically removed or marked for removal by the user.

In addition to avoiding faulty variants, participants also avoided "child" variants that were derived from faulty ones, even though they were not marked erroneous. For example, participant P10 commented *"If a branch had errors on it I stopped [at that app] thinking it couldn't help me out, because I can't create something from an error"*. This indicates the need for explicit "good" as well as "bad" quality markers. An implication for end users is that erroneous apps left in public repositories are going to affect how users perceive more recent, correct apps.

7.2. Implications for design

Based on observations and feedback from participants, we identified several additional design guidelines (Table 6).

7.2.1. Handling errors

As noted, while most participants were able to use the error annotations in AIH, they tended to bypass apps marked as erroneous. Including markers inside erroneous apps to direct attention to sources of errors (in the app) may help. Also as noted, some participants incorrectly assumed that apps developed from erroneous apps would have errors propagated to them too. Including percentages denoting total correctness may further help with DR#7 *(Include correctness and efficiency information for variants)* by reducing such assumptions. Hence, if a child node of an erroneous app has been corrected by another author and is annotated with a green color bar, this may be helpful.

Similarly, if an author of a child app has found and fixed an error that also affects the parent app, propagating the error notification to its parent app would be helpful. Error propagation mechanisms may thus help reduce errors in apps (DR#10).

Table 6
Design requirements (Extended).

No.	Design requirements emerging from formal study
DR#10	*Automated support for error propagation and mitigation*
DR#11	*Associate features of the app automatically*
DR#12	*Annotate features that are added or removed*
DR#13	*Recommend apps to users*
DR#14	*Include a search facility within AIH*
DR#15	*Display differences between two variants*
DR#16	*Allow apps to be combined/merged*
DR#17	*Provide a roadmap of the app with ability to zoom in and out*

7.2.2. Automatic feature abstraction

When selecting a variant, participants (1) looked for apps that had the most features similar to those listed in the requirements, and (2) evaluated the features and the context of the apps. Support for automatic feature extraction from program code (DR#11) can help users associate an app with its features. Here, by feature extraction, we mean extracting certain features (a list of feature names) provided by an application. Feature abstraction can also help in creating recommendation systems. Further, if automatic feature abstraction is supported, then semantic search mechanisms within AIH may help users find apps with desired features. Many feature location techniques like dynamic analysis, static analysis, textual, and historical analysis based on software repositories can be used to automate some or all of this process [18]. Also, given that many participants initially focused on leaf nodes in the graph view and then moved toward parent nodes, providing lists of features added and removed could help users evaluate and select more appropriate variants (DR#12).

Techniques like the Model Variants Comparison approach [49] (MoVaC) can be extended for EUP variation management tools to support feature abstraction and diff/merge functionalities. MoVaC identifies both feature commonality and variability by comparing a set of model variants. Each feature consists of a set of atomic model-elements. MoVaC also visualizes the identified features using a graphical representation where common and variable features are explicitly presented to users.

Feature abstraction can help create more sophisticated recommendation systems (DR#13). For example, as noted earlier, 50% of the participants in the base case searched on language semantics; if automatic feature abstraction were supported, then semantic search mechanisms within the AIH environment could help users find apps with desired features (DR#14).

7.2.3. Providing diff/merge functionality

Several participants looked for functionality in an app and wanted to copy or reuse only that functionality. For example, P1 commented, *"The task asked me for a feature that would say the operators and numbers, so [I needed] voice talk [referring to application]. The other thing I was looking for was if one calculator had a log button so I could combine these two"*. If automatic feature abstraction were supported, it could help users differentiate between variants at the feature level (DR#15) and enable the merging of variants (DR#16). Since AIH does not provide combine/merge capabilities, its current hierarchal visualization works fine but in the presence of merge capabilities graph based visualization over time may work better.

7.2.4. Improving scalability

The size of an app tree will increase as the number of apps increases. This will make labels difficult to read. Thus, there is a need to provide a roadmap of the variation space (e.g., a radar view) that allows users to zoom in and out in order to focus on a specific area of the space (DR#17).

8. Related work

In this section we discuss related work in the area of representing and transforming variations.

8.1. Variation management for professionals

The artifacts produced in software development evolve over time because of changing requirements and the need to adapt to technological progress. Software configuration management (SCM) systems [48,69] help manage software evolution and provide change control for software products. Examples of SCM systems are CVS, SVN, and Git. Professional software development all use some form of SCM system.

Software product lines (SPLs) (e.g., [14,53] – for surveys and systematic literature reviews see [16,24,41,61])– constitute a sub-area of software engineering in which variants are tracked using a (formalized) feature model, and created by "configuring" the model. SPLs tend to be used for large products, where the product architecture is rigorously engineered and different products need to exist concurrently.

One of the more commonly used paradigms in SPL is feature-oriented software development (FOSD) [2]. FOSD allows software to be assembled semi-automatically from features. Features are a specific way of expressing variation in software. Approaches for implementing features can be categorized as (1) annotative approaches [8,25,31], (2) compositional approaches [9,50], (3) metaprogramming approaches [63,64] and (4) choice calculus [20,45]. Feature models are also used to describe the structure of software product lines [52]. At a high level, features and their relationships are described with the help of feature models, which can be expressed in forms such as diagrams [13], algebras [28], and propositional formulas [4].

The foregoing approaches tend to target professional programmers (with some exceptions noted below) and are unlikely to be adopted by end users (as is evident from our survey results). Our approach provides more lightweight, seamless support for end-user programmers.

8.2. Variation management for end user programmers

Programming styles and reuse tend to create large numbers of variations. Users seeking reuse solutions must explore these variations. Stolee et al. [66] conducted a study on how end-user programmers program in Yahoo! Pipes – a web mashup programming environment. They found that authors create highly similar programs. They found that 60% of the programs in the repository had the same structure as other programs, and 73% were within a distance of one from other programs (where they defined distance in terms of additions, deletions, and substitutions of modules (nodes) and wires (edges) to obtain one program from another). They found that 43% of the programs were created by tweaking and were structurally similar to programs they created previously. This work emphasized the occurrence of variants in end-user repositories. They found that exploration of the repository was slowed by the substantial manual effort needed to create, compare and understand alternatives. This has been addressed by prior research in three primary ways, as follows.

8.2.1. Product configuration support for end users

Some researchers in the SPL community have developed tools meant to help end users in product configuration tasks. Rosa et al. [42] developed a toolset to capture configurable processes for film production by studying system variability based on a questionnaire modeled to include order dependencies and domain constraints. Rabiser et al.[55] developed a configuration tool, DOPLER CW, by analyzing existing configuration tools to identify key capabilities using the cognitive dimensions of a notations framework for guiding end users.

8.2.2. Exploring variations

Researchers have investigated the advantages of supporting variations for interface design practices. Terry et al. [68] present "Side Views", a model for interface designers that lets them compare multiple graphic designs by varying parameters. Later they created "Parallel Paths" to generate, manipulate and compare alternative solutions [51]. Parallel Paths was designed by considering what-if tools, augmented histories, and enhanced previewing mechanisms. Hartmann et al. [27] propose Juxtapose to generate alternative solutions and move between them by changing application logic and interface parameters. Another tool, d.note [26], helps designers selectively edit and execute source code. Kumar et al. [35] present Bricolage [35] another tool for web interface designers that allows users to transfer the style and layout of one page to another; this allows interface designers to transfer designs across different websites.

Our approach differs from these approaches as we are targeting end-user programmers, not interface designers. We are also managing variants at the code level as well as the design level, and for visual programming languages. Moreover, we allow users to create variants without changing their work process or employing extra effort.

8.2.3. Managing and analyzing variations through visualizations

Various mechanisms for automatically managing variations without affecting the workflow or behavior of users have been studied. Karlson et al. [30] introduce the concept of copy-aware computing ecosystems. They keep all digital artifacts as a single entity and call this a "version set". They show that such personal information management systems can help users keep track of changes and of the semantics behind copy operations. Our approach differs from theirs in that we focus on end-user programmers, and on program variations rather than document variations. Moreover, we consider more than just the copying habits of users.

In our own earlier work we created a *version* (not *variant*) management system for end users, Pipes Plumber, which tracks versions of Yahoo Pipes programs [40]. In Pipes Plumber, versions are represented with abstract list views of features. Pipes Plumber also denotes baselines and uses tags to represents versions that are erroneous, tested or untested. Controlled studies have shown that Pipes Plumber can help end users with exploration and backtracking through versions of code.

The approach presented in this article is supplementary to Pipes Plumber as we focus on visualizing variants. Pipes Plumber keeps track of minor changes in the code, i.e., at the workspace level, while AIH keep track of final changes in the code. AIH also provides additional provenance information on variants and shows them hierarchically and not as a linear list. AIH also keeps all apps in close proximity with other variations based on context; this helps support "orienteering" approaches for finding information based on context and origin. Information related to similarity, error status and project profiles helps users reuse and explore variants to find relevant variants for a particular context.

9. Conclusions and future work

This article is the first to present an investigation of the usage of variants by end-user programmers. Our survey allowed us to pose answers to three questions.

- *RQ1: How and why do end users create variants?*Our survey results revealed that end users do indeed reuse programs they create and code shared with teams. This suggests that being able to understand

the differences between variants is important. Further, when working in teams, having information about the authors of code is important.

- *RQ2: How do end users find variants?*While looking for variants, end users prefer to run code and look at outputs, to access source code and meta information such as filenames, and to refer to the creation and update dates of programs. When selecting variants users prefer to look at their major features such as correctness, similarity and authorship information.
- *RQ3: How do end users manage variants?*End users rely primarily on memory to track changes. They seldom make use of online or configuration management tools.

The results of our survey motivated the need for a variation management system by which end users can access and assess variants. Therefore, we developed AppInventorHelper (AIH), a variation aware system for end users working in the App Inventor environment. Our formative user study showed that AIH helps organize information on variants. We also found that AIH helps users explore and reuse variants, and understand code evolution.

Our study of AIH uncovered several potential new design requirements. Participants wished to see explicit information on whether code was correct or not. They wished to have features abstracted to help them find apps and determine the differences and similarities between variants, and they wanted to have a roadmap with the ability to zoom in and out. Our research indicates that incorporating these new requirements into AIH may significantly help improve users' interactions with the system.

Acknowledgments

We thank David Montz for helping with parts of our task implementation and empirical study. We also thank our pilot study and usability study participants.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.infsof.2018.06.008.

References

[1] AI, App inventor website, 2016.
[2] S. Apel, C. Kastner, An Overview of Feature-Oriented Software Development, (2009).
[3] S.L. P. B. A. Kitchenham, Personal opinion surveys, Guide to Advanced Empirical Software Engineering, 2008, pp. 63–92.
[4] D. Batory, Feature models, grammars, and propositional formulas, Proceedings of the International Conference on Software Product Lines, (2005), pp. 7–20.
[5] L. Beckwith, M. Burnett, V. Grigoreanu, S. Wiedenbeck, Gender HCI: what about the software? Computer 39 (11) (2006) 97–101.
[6] J. Benedek, T. Miner, Measuring desirability: new methods for evaluating desirability in a usability lab setting, Proceedings of Usability Professionals Association, (2002), pp. 8–12.
[7] A. Blackwell, First steps in programming: a rationale for attention investment models, Proceedings of the IEEE Symposium on Human Centric Computing Languages and Environments, (2002), pp. 2–10.
[8] Q. Boucher, A. Classen, P. Heymans, A. Bourdoux, L. Demonceau, Tag and prune: a pragmatic approach to software product line implementation, Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, (2010), pp. 333–336.
[9] G. Bracha, W. Cook, Mixin-based inheritance, SIGPLAN Notices 25 (10) (1990) 303–311.
[10] J. Brandt, P.J. Guo, J. Lewenstein, M. Dontcheva, S.R. Klemmer, Opportunistic programming: writing code to prototype, ideate, and discover, IEEE Softw. 26 (5) (2009) 18–24.
[11] M. Burnett, L. Beckwith, S. Wiedenbeck, S.D. Fleming, J. Cao, T.H. Park, V. Grigoreanu, K. Rector, Gender pluralism in problem-solving software, Interact. Comput. 23 (5) (2011) 450–460.
[12] M. Burnett, S.D. Fleming, S. Iqbal, G. Venolia, V. Rajaram, U. Farooq, V. Grigoreanu, M. Czerwinski, Gender differences and programming environments: across programming populations, Proceedings of the International Symposium on Empirical Software Engineering and Measurement, (2010), pp. 28:1–28:10.
[13] F. Cao, B.R. Bryant, C.C. Burt, Z. Huang, R.R. Raje, A.M. Olson, Automating feature-oriented domain analysis, Proceedings of the 2003 International Conference on Software Engineering Research and Practice, (2003), pp. 944–949.
[14] P. Clements, Software Product Lines: Practices and Patterns, Addison-Wesley Longman Publishing Co., Inc., 2001.
[15] A. Cypher, M. Dontcheva, T. Lau, J. Nichols, No Code Required: Giving Users Tools to Transform the Web, Morgan Kaufmann, 2010.
[16] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, A. Wasowski, Cool features and tough decisions: a comparison of variability modeling approaches, Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems, (2012), pp. 173–182.
[17] B.C.D.A.J.E.H.D.I.K. Sjøberg, Tore Dybå, building theories in software engineering, Guide to Advanced Empirical Software Engineering, (2008), pp. 312–336.
[18] B. Dit, M. Revelle, M. Gethers, D. Poshyvanyk, Feature location in source code: a taxonomy and survey, J. Softw. 25 (1) (2013) 53–95.
[19] K. El-Arini, C. Guestrin, Beyond keyword search: Discovering relevant scientific literature, Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining, (2011), pp. 439–447.
[20] M. Erwig, E. Walkingshaw, The choice calculus: a representation for software variation, ACM Trans. Softw. Eng. Method. 21 (1) (2011) 6:1–6:27.
[21] J. Estublier, D. Leblang, G. Clemm, R. Conradi, W. Tichy, A. van der Hoek, D. Wiborg-Weber, Impact of the research community on the field of software configuration management: summary of an impact project report, ACM SEN 27 (5) (2002) 31–39.
[22] S. Fischer, L. Linsbauer, R.E. Lopez-Herrejon, A. Egyed, Enhancing clone-and-own with systematic reuse for developing software variants, Proceedings of the International Conference on Software Maintenance and Evolution, (2014), pp. 391–400.
[23] R.L.F. Forrest Shull, building theories from multiple evidence sources, Guide to Advanced Empirical Software Engineering, (2008), pp. 337–364.
[24] M. Galster, D. Weyns, D. Tofan, B. Michalik, P. Avgeriou, Variability in software systems – a systematic literature review, IEEE Trans. Softw. Eng. 40 (3) (2014) 282–306.
[25] GNU, GNU Project. The C Preprocessor, Free Software Foundation, 2009.
[26] B. Hartmann, S. Follmer, A. Ricciardi, T. Cardenas, S.R. Klemmer, d.note:revising user interfaces through change tracking, annotations, and alternatives, Proceedings of the ACM Conference on Human Factors in Computing Systems, (2010), pp. 493–502.
[27] B. Hartmann, L. Yu, A. Allison, Y. Yang, S.R. Klemmer, Design as exploration: creating interface alternatives through parallel authoring and runtime tuning, Proceedings of the ACM Symposium on User Interface Software and Technology, (2008), pp. 91–100.
[28] P. Höfner, R. Khedri, B. Möller, Feature Algebra, in: Formal Methods, volume 4085 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2006, pp. 300315.
[29] M. Jones, S. Scaffidi, Obstacles and opportunities with using visual and domain-specific languages in scientific programming, Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, (2011), pp. 9–16.
[30] A.K. Karlson, G. Smith, B. Lee, Which version is this?: Improving the desktop experience within a copy-aware computing ecosystem, Proceedings of the ACM Conference on Human Factors in Computing Systems, (2011).
[31] C. Kästner, S. Apel, M. Kuhlemann, Granularity in software product lines, Proceedings of the International Conference on Software Engineering, (2008), pp. 311–320.
[32] A.J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M.B. Rosson, G. Rothermel, M. Shaw, S. Wiedenbeck, The state of the art in end-user software engineering, ACM Comput. Surv. 43 (3) (2011) 21:1–21:44.
[33] A.J. Ko, T.D. LaToza, M.M. Burnett, A practical guide to controlled experiments of software engineering tools with human participants, Emp. Softw. Eng. 20 (1) (2015) 110–141.
[34] C.W. Krueger, Variation management for software production lines, Proceedings of the International Conference on Software Product Lines, (2002), pp. 37–48.
[35] R. Kumar, J.O. Talton, S. Ahmad, S.R. Klemmer, Bricolage: Example-based retargeting for web design, Proceedings of the ACM Conference on Human Factors in Computing Systems, (2011), pp. 2197–2206.
[36] S.K. Kuttal, Leveraging variation management to enhance end users' programming experience, Ph.D. dissertation. University of Nebraska, 2014.
[37] S.K. Kuttal, A. Sarma, A. Swearngin, G. Rothermel, Versioning for mashups – an exploratory study, Proceedings of the International Conference on End-User Development, (2011), pp. 25–41.
[38] S.K. Kuttal, S. Sarma, G. Rothermel, History repeats itself more easily when you log it: Versioning for mashups, Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, (2011), pp. 69–72.
[39] S.K. Kuttal, S. Sarma, G. Rothermel, Debugging support for end-user mashup programming, Proceedings of the ACM Conference on Human Factors in Computing Systems, (2013), pp. 1609–1618.
[40] S.K. Kuttal, S. Sarma, G. Rothermel, On the benefits of providing versioning support for end-users: an empirical study, Trans. Comput. Hum. Interact. 21 (2) (2014) 9:1–9:43.
[41] C. L., M. Ali Babar, A systematic review of evaluation of variability management approaches in software product lines, Inf. Softw. Technol. 53 (4) (2011) 344–362.
[42] M. La Rosa, W.M.P. van der Aalst, M. Dumas, A.H.M. ter Hofstede, Questionnaire-based variability modeling for system configuration, Softw. Syst. Model. 8 (2) (2009) 251–274.
[43] LabVIEW, Labview website, 2016.

[44] J.R. Landis, G.G. Koch, The measurement of observer agreement for categorical data, Biometrics 33 (1) (1977).
[45] D. Le, E. Walkingshaw, M. Erwig, #ifdef confirmed harmful: promoting understandable software variation, Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, (2011), pp. 143–150.
[46] C.H. Lewis, Using the "Thinking Aloud" method in cognitive interface design, RC 9265, IBM, 1982.
[47] H. Lieberman, F. Paternó, V. Wulf, End User Development, Springer, first ed. 2006.
[48] S.A. MacKay, The state of the art in concurrent, distributed configuration management, Selected Papers from the ICSE SCM-4 and SCM-5 Workshops, on Software Configuration Management, (1995), pp. 180–193.
[49] J. Martinez, T. Ziadi, J. Klein, Y. Traon, Identifying and visualising commonality and variability in model variants, Proceedings of the European Conference on Modelling Foundations and Applications, (2014), pp. 117–131.
[50] M. Mezini, K. Ostermann, Variability management with feature-oriented programming and aspects, SIGSOFT Softw. Eng. Notes 29 (6) (2004) 127–136.
[51] T. Michael, E.D. Mynatt, K. Nakakoji, Y. Yamamoto, Variation in element and action: supporting simultaneous development of alternative solutions, Proceedings of the ACM Conference on Human Factors in Computing Systems, (2004), pp. 711–718.
[52] D.L. Parnas, On the design and development of program families, IEEE Trans. Softw. Eng. SE-2 (1) (1976) 1–9.
[53] K. Pohl, G. Böckle, F.J. Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer-Verlag New York, Inc., 2005.
[54] Popularity of App Inventor, Popularity of app inventor, 2017.
[55] R. Rabiser, P. Grünbacher, M. Lehofer, A qualitative study on user guidance capabilities in product configuration tools, Proceedings of the International Conference on Automated Software Engineering, (2012), pp. 110–119.
[56] S.S. Ragavan, B. Pandya, D. Piorkowski, C. Hill, S.K. Kuttal, A. Sarma, M. Burnett, PFIS-V: Modeling foraging behavior in the presence of variants, Proceedings of the CHI Conference on Human Factors in Computing Systems, (2017), pp. 6232–6244.
[57] L.M. Rea, R.A. Parke, Designing and Conducting Survey Research: A Comprehensive Guide, (2015).
[58] M.B. Rosson, J.M. Carroll, The reuse of uses in smalltalk programming, Trans. Comput. Hum. Interact. 3 (1996) 219–253.
[59] I.J.M. Ruiz, M. Nagappan, B. Adams, A.E. Hassan, Understanding reuse in the android market, 20th IEEE International Conference on Program Comprehension (ICPC), (2012), pp. 113–122.
[60] C. Scaffidi, M. Shaw, B. Myers, Estimating the numbers of end users and end user programmers, Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, (2005), pp. 207–214.
[61] P.Y. Schobbens, P. Heymans, J.C. Trigaux, Feature diagrams: a survey and a formal semantics, Proceedings of the IEEE International Requirements Engineering Conference, (2006), pp. 139–148.
[62] C.B. Seaman, Qualitative methods in empirical studies of software engineering.
[63] T. Sheard, Accomplishments and research challenges in meta-programming, Proceedings of the 2Nd International Conference on Semantics, Applications, and Implementation of Program Generation, (2001), pp. 2–44.
[64] T. Sheard, A Taxonomy of Meta-Programming Systems, (2016).
[65] S. Srinivasa Ragavan, S.K. Kuttal, C. Hill, A. Sarma, D. Piorkowski, M. Burnett, Foraging among an overabundance of similar variants, Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, (2016), pp. 3509–3521.
[66] K.T. Stolee, S. Elbaum, A. Sarma, Discovering how end-user programmers and their communities use public repositories: a study on yahoo pipes, Inf. Softw. Technol. 55 (2013) 1289–1303.
[67] J. Teevan, C. Alvarado, M.S. Ackerman, D.R. Karger, The perfect search engine is not enough: a study of orienteering behavior in directed search, Proceedings of the ACM Conference on Human Factors in Computing Systems, (2004), pp. 415–422.
[68] M. Terry, E.D. Mynatt, Side views: Persistent, on-demand previews for open-ended tasks, Proceedings of the ACM Symposium on User Interface Software and Technology, (2002), pp. 71–80.
[69] W.F. Tichy, RCS-A system for version control, Software – Practice & Experience, (1985), pp. 637–654.
[70] J. Tsay, L. Dabbish, J. Herbsleb, Influence of social and technical factors for evaluating contribution in GitHub, Proceedings of the International Conference on Software Engineering, (2014), pp. 356–366.
[71] D. Wolber, App Inventor and real-world motivation, Proceedings of the ACM Technical Symposium on Computer Science Education, (2011), pp. 601–606.
[72] Y. Pipes, Yahoo Pipes, 2014.
[73] Y. Yoon, B. Myers, An exploratory study of backtracking strategies used by developers, Proceedings of the International Workshop on Cooperative and Human Aspects of Software Engineering, (2012), pp. 138–144.