Towards helping end-user programmers' information foraging by manipulating information features in a patch

AN ABSTRACT OF THE PROJECT OF

Balaji Athreya_for the degree of Master of Science in Computer Science_presented on December 16, 2013.

Title: Towards helping end-user programmers' information foraging by manipulating information features in a patch

Abstract approved:

____

Christopher Scaffidi

Software maintenance tasks often require finding information within existing code, which is time-consuming and difficult even for professional programmers [1,55]. For example, programmers may need to know what code implements certain functionality or what is the purpose of certain code [3,2]. In response, researchers have developed tools to help programmers find information during programming tasks [5,6,7,8]. The empirical success of these tools can be explained by Information Foraging Theory (IFT) [9], which predicts how people will seek information by navigating through virtual patches in an information system. In the case of programming, these patches are often chunks of code (e.g., functions), with navigable links for moving among methods. IFT predicts people will perceive cues (such as words or symbols) associated with navigable links, select links that seem relevant to their information needs, and attempt to obtain the needed information by maximizing the rate of information gained relative to the cost of navigating and understanding

patches. Many existing tools accelerate foraging by decreasing the cost associated with navigating from one patch to another.

IFT suggests that the visual weight of the information features in a patch can have a strong effect on a predator's foraging choices and, consequently, on how well the predator succeeds in maximizing the rate of information gain. In an ideal situation, visual weight will efficiently lead the predator to the needed information; on the other hand, if visual weight leads the predator astray, then this could lead the predator to process more patches than necessary (increasing cost and reducing the rate of information gain). Therefore, it is anticipated that increasing the relative weight of important information features with respect to unimportant information features will aid an end-user programmer's foraging effort. Towards this end, two prototypes were implemented: each of these uses an existing algorithm [10] to identify the most important lines of code in a function. One prototype increases the relative weight of important information features by highlighting important lines of code; the other prototype decreases the relative weight of unimportant information features by hiding unimportant lines of code. This research's focus is end-user programmers, who have received minimal attention in prior work.

An empirical study evaluated the effectiveness of the prototypes relative to the baseline (no information feature modification). These results indicate that increasing the relative weight of important information features by highlighting important statements had a significant effect on the amount of information foraged and the rate of information gained; on the other hand, decreasing the relative weight of unimportant information features by hiding unimportant statements had a significant effect on the rate of information gained, but not on the amount of information foraged. Neither approaches seemed to have any effect on the amount of time spent on information foraging or patch-to-patch navigation.

Towards helping end-user programmers' information foraging by manipulating information features in a patch

by
Balaji Athreya

A PROJECT

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented December 16, 2013
Commencement June 2014

Master of Science project of Balaji Athreya presented on December 16, 2013

APPROVED:

Major Professor, representing Electrical and Computer Engineering

Director of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

Balaji Athreya, Author

ACKNOWLEDGEMENTS

First and foremost, I'm very grateful to my advisor, Dr. Christopher Scaffidi for his constant motivation, guidance and patience. Working with him has helped me learn a lot about how to approach and solve problems. His course on mobile and cloud computing not only helped prepare well for industry, but also opened up several new opportunities.

I'd like to thank my committee member, Dr. Margaret Burnett, whose classes were the most interesting and challenging. Her courses were very helpful in learning the HCI aspects of programming and in writing this work. Interacting with both of them have inspired me and taught me so much, both inside the classroom and outside it. I'd like to thank my committee member, Dr.Pankaj Jaiswal for his support, guidance and the opportunity to hone my programming skills through his lab.

I'd like to thank Chris Chambers for helping me write my project report. I'd like to thank Amin, Iftekar, Faezah and Sheela for providing moral support through the years. I'd like to thank Dr. Bella Bose, Colisse, Justin Elser, Justin Preece, Joseph Jess, Laurel, Nicole, Padma, Palitha, Rajani, Tina, Tracey, Vindhya and the members of the EECS office who were so kind to me during the past two years. Finally, I'd like to thank my parents and my family for their love and support.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF TABLES

LIST OF FIGURES

1 Introduction

The number of end-user programmers is increasing every year and has long since surpassed the number of professional programmers [12]. End-user programmers come from a variety of backgrounds ranging from secretaries, accountants, secondary school students [13], teachers [14] to scientists [15]. These varied users write programs on many types of platforms including spreadsheets, web mash-ups [16,17], web scripting [18] and animations [21,22]. Recently, end-user programmers have begun programming on smart-phones [19, 20].

Many of these end-user programming environments provide a central code repository to allow users to share code [18,19,22,23,17]. In some cases, certain groups of end-user programmers write programs specifically to be used by others [24]. In the absence of repositories, end-user programmers tend to share code informally by passing on the source file [14,15,24]. Reusing existing code is crucial for end-user programmers because it enables them to save time by making use of code that already works, potentially reducing the risk of writing new code that might have new bugs. However, this also poses a new challenge: in order to be able to reuse, maintain or modify existing code, the end-user programmer needs to understand a program that may not have been written with reuse in mind. Thus, program comprehension is important for reusing and maintaining existing code.

There have been several studies on program comprehension over the years. Studies have discovered three strategies used by programmers during program comprehension - top-down model [25], bottom-up model [26] and integrated meta-model [27]. All of these program-comprehension strategies, in turn, require finding information within the code that the end-user programmer can combine into a working comprehension of the code.

To better understand how people look for information in code, new models of program comprehension are being developed based on information foraging theory (IFT), which predicts and explains how people navigate through an information system, by maximizing the ratio of information gained to interaction cost [9]. Models based on IFT (e.g., [1,41]) are more powerful than the older models [25,26,27] because they can predict and explain the lower level navigation actions of a programmer.

Separately, apart from efforts at modeling foraging, several tools [4,5,6] have been developed to help programmers effectively search, relate and collect relevant and useful information during programming tasks. For example, Hipikat [5] helps a programmer by providing navigable links to relevant non-code artifacts (bug reports, emails, version control logs) based on lexical similarity with the search term. Whyline [4] helps a programmer by enabling him/her to ask questions about a program's runtime behavior and providing navigable links that map the program's output with the piece of code that is responsible for that output. Codefinder [6] helps a programmer by providing navigable links to directly reusable code based on the search term and by suggesting suitable alternate search terms. A recent literature survey argued that the success of these tools can be explained in terms of IFT by noting that the tools essentially reduce the cost of patch-to-patch navigation [30].

To date, only a smaller number of tools have been aimed at reducing the other main cost of foraging, which is the cost of extracting useful information efficiently from a patch of code after a programmer has navigated to it. There have been a few attempts at summarizing the source code in order to reduce the cost of reading and understanding it. For example, Haiduc et al [47] used text retrieval techniques [49,50] and structural information to summarize source code files. Sridhara et al developed a technique [10] to identify important statements within a Java method and generate natural language summaries for Java methods. Rastkar et al developed a technique [48] that produces natural language summaries to help programmers understand code (that are relevant to the task at hand) that crosscut multiple modules of a source code. All of these efforts are targeted towards professional programmers.

One approach to reduce the cost of understanding (and subsequently the cost of foraging) is highlighting important information features within a patch of code so that the programmers can focus on those important parts [30]. For example, syntax highlighting in Integrated Development Environments (IDEs) decreases the cost of reading and understanding a chunk of code. Fleming et al reviewed other more sophisticated forms of code highlighting (that emphasize cues) used in tools for professional programmers in [30]. For example, Jigsaw [40] uses code highlighting with colors to alert professional programmers about overlooked chunks of code and illustrate how reusable code was integrated with existing source code. Duplac [43], a tool to identify code clones during refactoring tasks, uses code highlighting with colors to indicate similarities and differences between code clones.

Highlighting does not appear to be used as widely among end-user programmers. Spreadsheets are often used as reporting tools (for storage rather than computation) [51] and contain a significant amount (~ 40%) [52] of non-numeric and textual data. A plug-in for Microsoft Excel based on Topes [54] by Scaffidi et al uses color-highlighting (red colored triangles) to help end-user programmers find and fix typo errors in textual data. A fault localization technique [44] by Ruthruff et al uses color-highlighting to aid end-user programmers in finding incorrect formulas in spreadsheets during debugging tasks. Both the two tools highlight areas of the program that appear to contain problems - while this information can be very useful during debugging, they do not identify important lines of code which can be very useful during other software maintenance tasks such as adding new features, refactoring and code re-use.

No empirical research appears to have been done to investigate whether end-user programmers working on maintenance tasks obtain benefits from highlighting inside of code patches or from the removal of unimportant information features . Also, it is not clear that increasing relative weight of important information features will be as helpful for end-user programmers as for professional programmers. One reason is that end-user programmers’ code may not be designed for reuse and is often hard to understand during maintenance tasks—Nardi even writes, “It is not clear whether users who modify existing example programs could ever really come to understand the programs they modify" [24]. So maybe such code does not have many information features that would be important enough to highlight, or maybe the important information features are not as identifiable in end-user code as they are in professional programmers' code, meaning that highlighting might not help end-user programmers very much. On the other hand though, end-user programmers only write code as a secondary task in their work, rather than their main task [12], so they might be relatively unfamiliar with programming and/or with the programming tool, This could make them all the more dependent on tool assistance with understanding patches of code, which could make highlighting important statements or hiding unimportant statements even more beneficial to end-user programmers than to professionals. Further investigation is needed of the relationship between the visual weights of information features and foraging costs for end-user programmers.

Therefore, this work investigates the effects of visual weights of important and unimportant information features on an end-user programmer's foraging cost. Specifically, this work investigates if visual weights of important and unimportant information features affect the amount of information foraged, foraging time and the rate of information gained of an end-user programmer working on real-world maintenance tasks using TouchDevelop [31] - a mobile application creation environment. Using an existing algorithm [10] that identifies important lines of code within a patch (function), two prototypes were developed - one prototype highlights these important lines of code at the cost of unimportant lines; the other prototype hides the unimportant lines of code.

An empirical study was conducted to evaluate the effectiveness of the prototypes relative to the baseline and analyzed the data using non-parametric statistical tools.

The results of this study will be useful for understanding how best to design tools that help end-user programmers to understand patches of code. Especially, the study results will reveal whether highlighting or eliminating lines of code can help people to understand code more effectively during maintenance tasks. The study results will discover opportunities for further research, such as investigations about when different kinds of cue enhancement do or do not help end-user programmers.

This document is organized as follows: In Section 2, the existing literature on program comprehension strategies, information foraging theory and existing software engineering tools for information-intensive activities are reviewed; Section 3 reviews the prototypes and how important lines within a function are identified. Section 4 details the experiment and evaluation procedure. Section 5 discusses the results of the empirical study and possible explanations to the results. Finally, Section 6 concludes by stating the contributions of this study and discuss future research opportunities.

2 Literature Review

This section reviews previous research in the field of program comprehension strategies, concepts in IFT, and existing software engineering tools that help programmers with information foraging during programming tasks. Reviewing previous work shows that existing tools offer little help to professional and end-user programmers to efficiently extract useful information from a patch of code during information foraging and that end-user programmers face impassable barriers during information foraging. So, efforts aimed at reducing the cost of extracting information from a patch could be very beneficial to end-user programmers.

2.1 Program comprehension strategies

Program comprehension is the process of assigning meaning to program text; it involves understanding the meaning of each program statement, control flow, data flow and the purpose of groups of statements [33]. Studies have discovered three common strategies used by programmers during program comprehension - top-down model [25], bottom-up model [26] and integrated meta-model [27]. In the top-down model, program comprehension is a hypothesis-driven process, in which a programmer begins with a vague hypothesis about the entire program. Then, the programmer refines it to a tree of secondary hypotheses which are verified or rejected [27]. Top-down model requires the programmer to be familiar with the program domain and/or to have programming expertise - hence this model is mostly common among experienced programmers.

On the other hand, the bottom-up model is common among novice programmers. In this model, the novice programmer first understands the control flow of the program by grouping pieces of code into higher-level abstractions known as “procedural episodes” [28]. Then the novice programmer investigates data objects and functions that connect these procedural episodes to gain an understanding about the data flow of the program [27]. Finally, the integrated meta-model which was proposed by von Maryhauser and Vans suggests that programmers often choose one of these two models as their dominant strategy depending on their domain knowledge and switch between the two models as more information is presented to them while they try to understand a program. Programmers may switch their strategy to adapt to external stimuli, becoming what has been referred to as “opportunistic processors” [29].

In all of these strategies, programmers rely on beacons, “cues that index into knowledge, [which] can be text or a component of other knowledge. For example, a swap statement inside a loop or a procedure can be a beacon for a sorting function; so can the procedure name Sort” [27]. So a beacon is a kind of cue that helps a programmer to associate code with meaning.

Recently, a new model of program comprehension has emerged based on IFT, which offers a more complete perspective not only on the role of just beacon cues, but also the relationship between cues and programmer navigation throughout the code base. While being consistent with the earlier program understanding research above, this new line of work can also explain and predict lower level actions of a programmer. From this

IFT-based standpoint, program comprehension is a "process of *searching*, *relating*, and *collecting* relevant information in a graph and forming perceptions of relevance from cues in the programming environment" [1]. In the aforementioned graph, each node is an individual patch that appears on-screen (such as a chunk of code, documentation, comments, other metadata, etc.) connected by an edge between the nodes (such as calls, declaration, definition, or any other relationship represented as a navigable link between patches). This new model of program comprehension can be illustrated in the following figure below. (taken from [1]).

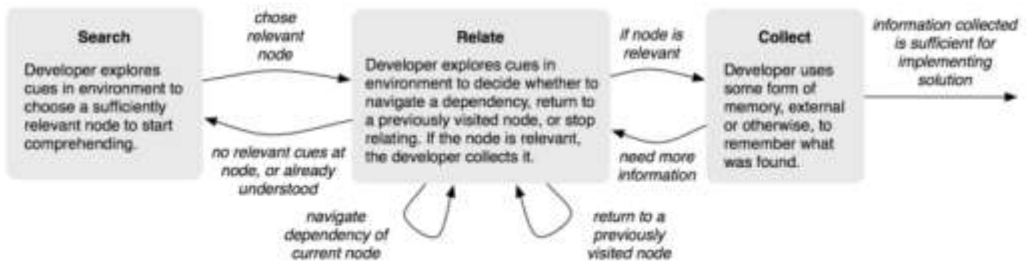

Search
Developer explores cues in environment to choose a sufficiently relevant node to start comprehending.
chose relevant node
no relevant cues at node, or already understood
Relate
Developer explores cues in environment to decide whether to navigate a dependency, return to a previously visited node, or stop relating. If the node is relevant, the developer collects it.
navigate dependency of current node
return to a previously visited node
if node is relevant
need more information
Collect
Developer uses some form of memory, external or otherwise, to remember what was found.
information collected is sufficient for implementing solution

Figure 1. A new understanding of program understanding [1] based on Information Foraging Theory

As the figure illustrates, a programmer begins by looking (*searching*) for a node in the graph that is relevant to the task at hand. After finding a relevant (perceived) node, potentially based on the presence of certain cues, he/she tries to understand (*relate*) the node. The programmer understands the node by processing the information features located within the node or by investigating the sub-graph it is a part of, choosing the most relevant (perceived) neighbor node in the sub-graph, navigate to it and understand the new node and so on. If the programmer deems the new node not useful, he/she may trace back to the previous node to pick another relevant node or may completely abandon the *relate* phase returning back to the *search* phase. This searching and relating will continue until the programmer decides that he has *collected* all relevant information needed for the task at hand. At this point, he/she stops the program comprehension process and starts focusing on implementing a solution, which may further warrant more search, relate and collect activities.

The new model suggests that there are three important factors that determine the success of a programmer - first, the programming environment must provide adequate and representative cues to guide searches; secondly, the programming environment must provide useful cues so that a programmer can determine the relevance/usefulness of a node in the graph; thirdly, the environment must help the programmer in effectively collecting relevant information within the graph.

Given the aforementioned factors relevant to helping programmers, several software engineering tools assist programmers in effectively searching, relating and collecting relevant information while working on programming tasks. They are discussed in section 2.3. In order to explain why these tools appear to help professional programmers, it is first necessary to explain more of the details about IFT, which are covered in section 2.2.

2.2 Concepts in Information Foraging Theory

This work is informed by the new model of program comprehension described above and information foraging theory (IFT). This section describes IFT and its terminology.

IFT is a theory about how people navigate through an information system, foraging for information while maximizing the value of information gained and minimizing the interaction cost. It can "explain and predict how people best shape themselves for their information environments and how information environments can be best shaped for people" [34].

IFT likens a person looking for information in an information environment to a predator looking for its prey. The information environment consists of a *topology* which is made up of *information patches.* Each information patch is made up of several units of *information features (*words, phrases, figures) and the predator is in search of its *information goal,* a specific set of information features (each of which is a *pre*y*)*. The predator moves from one information patch to another by processing special information features known as *links (menus, hyperlinks).* Each link possesses *cues*, which are indicators of the information available at the other end of the link. In addition, cues embedded within a patch may carry additional information. The predator processes these *cues* to determine the likelihood (known as *information scent*) of the presence of some *prey* at the other end of the link. The cost associated with processing cues is the *cost of* *the link.* The effort the predator spends on processing the information features in a patch is known as the *cost of the information patch*. The definitions are tabulated below.

IFT Term	Definition
Predator	The person looking for information
Information patch	A set of information features
Information features	Units of information in a patch that the predator can process
Links	Information features connecting two information patch
Topology	Collection of information patches and the links between them
Information goal	A set of information features that the predator is looking for
Prey	Elements of information goal
Cue	Indicators that signal information present at the other end of a link
Cost of an information patch	Measure of the effort required to process the information in an information patch
Cost of a link	Measure of the effort required to process the cues of a link
Information scent associated with a link	The predator's estimation of the likelihood of some prey at the other of a link

Table 1. IFT Terminology (adapted from [30])

IFT suggests that the predator may perform one of the following three actions while searching for its information goal:

- process the information features present in the current patch
- navigate to a nearby patch
- add new patches to the topology (known as *enrichment*)

IFT predicts that the predator will choose the action that has the highest *expected* benefit-to-cost ratio. That is, it will choose the action that maximizes the expected value of information gained per expected processing cost. This is given by the formula, where the argmax expression iterates over all the available choices:

$$\text{Predator's choice of action} = ArgMax\left(\frac{Exp(V)}{Exp(C)}\right), \text{ where}$$

Exp (V) = expected value of the information that can be gained through an action

Exp (C) = expected processing cost associated with the action

IFT also suggests that given an information patch containing many links each of which leading to different information patches, the predator will choose the link with the highest factor given by the formula, [30]

$\sum_{i} \sum_{j} W_j S_{ji}$ where W_j is the amount of attention the predator pays to the cue j and S_{ji} is the information scent the predator associates with the link because of the cue j. In prior work [57], W_j has been treated as a direct function of a cue's visual weight. In contrast to efforts aimed at clarifying how predators choose links for navigation (above), there has been much less effort aimed at clarifying how predators evaluate the other terms above, particularly expected cost of processing cues within a patch.

2.3 An IFT perspective on existing SE tools

Research in end-user software engineering [3] identifies six learning barriers faced by end-user programmers trying to learn a new programming system. One of the six learning barriers is information barrier. The study [3] by Ko et al defines information barrier as "properties of an environment that make it difficult to acquire information about a program's internal behavior (i.e., a variable's value, what calls what). " According to this study, information barriers occur when an end-user programmer is not able to verify his/her hypothesis about the program's internal behavior. Ko et al reported that end-user programmers did not overcome 71% of the information barriers they faced. The remaining were overcome by assuming *something* about the program's internal behavior. Ko et al also observed that, the remaining barriers (design, selection, use, co-ordination and understanding) often led to information barriers. Clearly, end-user programmers often struggle with information barriers!

In IFT terms, information barriers can be viewed as instances where the programmer has navigated to a particular information patch, but he/she

- is unable to extract useful information about the patch and form/confirm/reject a hypothesis, or
- is unable to pick a relevant link and navigate to another information patch to further his/her information foraging.

Thus, end-user programmers can be helped to overcome information barriers by providing them

- tools to aid in efficiently extract useful information *within* an information patch
- tools to aid in efficiently navigating *between* information patches in the neighborhood

Prior research on tools to efficiently extract useful information from a patch of code is very minimal(the tools for professional programmers mentioned in section 1). However, several tools have been developed to help both end-user programmers and professional programmers identify and navigate between relevant patches. Fleming et al, in their comprehensive study [30] of software engineering tools from an IFT-perspective also arrive at the same conclusion. Some of these tools are reviewed here.

Hipikat [5] is a plug-in for Eclipse integrated development environment platform. It helps a professional programmer find relevant non-code artifacts (such as bug reports, emails and version control logs) for a given search query. The plug-in interface provides clickable links to non-code artifacts annotated with the reason they were selected and a vote of confidence. Non-code artifacts can be particularly useful because they contain information that is not available in the source code. For example, bug reports often contain instructions to reproduce a particular bug; emails between developers might contain documentation about special cases in the code that is not available elsewhere [5]; version control logs can pinpoint to exact changes that introduced new bugs. In essence, Hipikat identifies relevant information patches that was previously hidden to the programmer.

Whyline [4] allows end-user and professional programmers to ask "why did" and "why didn't" questions about a program's output. It allows an end-user programmer to reason about some error in the program's output by providing links to all lines of code responsible for the program's incorrect output. It annotates the links with "why" questions that are answered by the corresponding line of code. Study [4] revealed that programmers using Whyline spent considerable very less time (by a factor of 8) than programmers without Whyline. Programmers with Whyline also had a higher success rate than programmers without Whyline during debugging tasks.

One common approach to help programmers find reusable code is by reducing the cost of "*searching*" for relevant code. For example, Contextual Search tool [39] automatically generates natural language phrases from the source code and uses it to identify new code that can augment the existing source code. Other tools are based on monitoring program behavior to develop a model about the relationship among different code patches [37]. The relationships among pieces of code can also be clarified through specifications and tools to aid in creating and using formal specifications [36], such as by using contracts, security constraints and test-cases to search for a particular program behavior for re-use [38]. Integrating reusable code with an existing codebase can be an expensive process, because the programmer has to identify dependencies of the reusable code and decide if the dependencies are required. In some cases, the dependencies can be removed with some modifications and in some cases the dependencies are required (which might require further foraging). Gilligan [7] generates a list of links to the dependencies of a reusable code during the search phase and allows programmers to label each link with information about their decision to add/remove each dependency. It also automatically color-codes each link so that programmers can focus only on the most relevant links during integration phase. From an IFT perspective, this serves to enhance cues and help programmers decide which links to navigate.

A common motif among all these tools is that they enable a programmer to enrich their topology with new relevant information patches, and/or to navigate more effectively among existing patches. Since the vast majority of these tools are developed for professional programmers, the applicability and usefulness of the principles behind these tools to end-user programmers should be further investigated. Having showed that end-user programmers face barriers while finding information about program behavior and existing tools offer little help in overcoming these information barriers, the next section describes a new approach that is intended to help end-user programmers extract useful information from an information patch (which in turn lowers foraging cost).

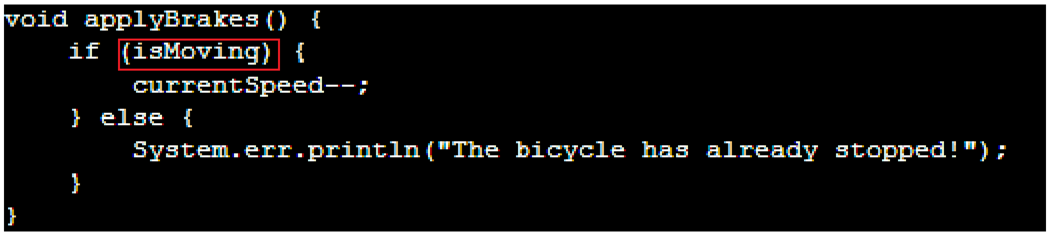
3 Approach

This section details a solution that is intended to help end-user programmers efficiently extract useful information from information patches. Section 3.1 describes the algorithm from related work that identifies important statements within a Java method, as well as the manner in which this existing algorithm has been adapted in order to apply it to finding important statements in the TouchDevelop programming language. Section 3.2 describes the new tool and the html output generated by this tool, which can highlight important statements in TouchDevelop and/or hide unimportant statements. Later sections will discuss this tool was used to investigate how the highlighting of important information features or hiding of supposedly unimportant information features were able to affect how end-user programmers foraged for information in TouchDevelop code.

3.1 Identifying important statements within a function

Previous work [10] by Sridhara et al presents a technique to generate summary comments for a Java method. Summary comments can be described as “descriptive comments that summarize major algorithmic actions of a method ” [10]. One contribution of that work is a set of rules for identifying when a particular Java *s_unit* (statement) within a method is “important” enough that it should be included in the method’s summary comments. That work also describes a procedure that applies these set of rules to identify important *s_units* (statements) within a function. These *s_units* are then converted into natural language text and the text is concatenated together to form a summary.

An *s_unit* is "*a Java statement, except when the statement is a control flow statement; then, the s_unit is the control flow expression with one of the if, while, for or switch keywords*" [10]. That is, in control statements, s*_unit* refers to the conditional expression within the control statement. For example, in the Java code snippet in Figure 2 the control s_unit is highlighted.



```
void applyBrakes() {
    if (isMoving) {
        currentSpeed--;
    } else {
        System.err.println("The bicycle has already stopped!");
    }
}
```

Figure 2. control s_unit example in Java (code taken from Java Oracle docs [53])

Sridhara et al developed the rules for identifying *s_units* that ought to be included in a summary by studying comments from popular open source Java programs and surveying experienced Java programmers about which statements they felt should be included in a method's summary comments [10]. They identified five types of *s_units* that should be included in a Java method's summary comment.

Three terms introduced by Sridhara et al - *action*, *theme* and *secondary arguments* of a method signature – will help with explaining the rules. All method signatures contain an '*action*' term, consisting of the method name. The '*theme*' consists of the parameters, and the '*secondary arguments*' are the object(s) operated upon. Consider the Java code snippet in Figure 3 that appends an object - 'item' to the end of a list referred by the variable ‘list’. In this example, item is the *theme*, append is the *action* and list is the *secondary argument*.

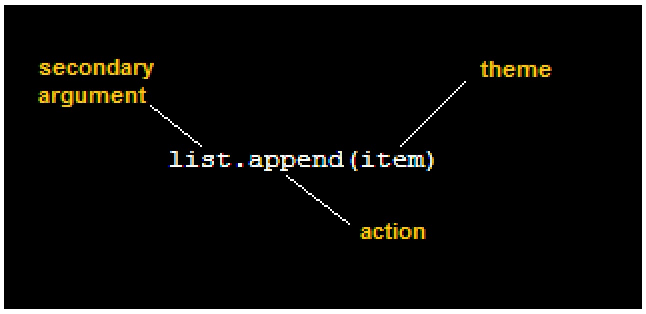
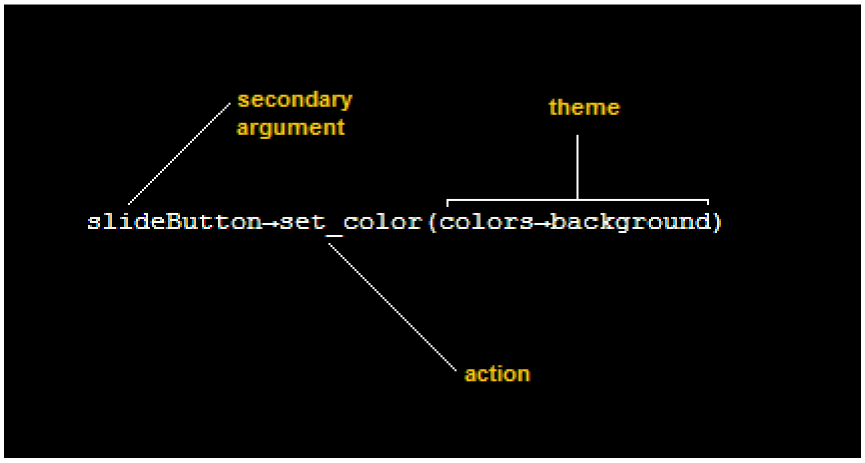


Figure 3. A simple Java s_unit (adapted from [10])

While the concept of *action*, *theme* and *secondary arguments* and the set of rules for identifying *s_units* were originally defined for Java, they can also be applied for TouchDevelop language.

The TouchDevelop language is a mix of imperative, object-oriented, and functional features and is statically typed[31]. It doesn’t allow defining custom data-types or user interface (UI) elements [19]. The language is primarily textual with a few non-ASCII graphical characters to represent some elements of the syntax [19]. For example, a → indicates a dereference of an object member (which might be object property or a method). A TouchDevelop script may contain several actions (functions), event-listeners (e.g., button clicked, text changed), global variables and UI elements. Global variables are truly global - that is, these variables are stored on the cloud and are accessible by programs on other phones [19]. The UI elements can have various properties (e.g., position, color, content, gravity) [31]. Consider the simple TouchDevelop statement in Figure 4, that picks the color of the application background and applies it to the color of a button element referred by the variable slideButton.



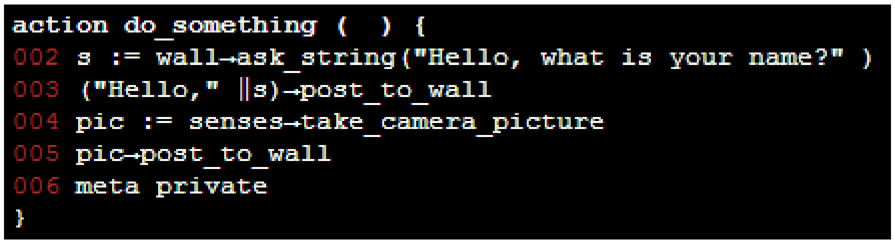
```
slideButton→set_color(colors→background)
```

Figure 4. A simple TouchDevelop s_unit

In this example, set_color is the *action*, colors→background is the *theme* and slideButton is the *secondary argument*. Theme and secondary arguments themselves can be individual *s_units*. Below, the five *s_unit* types identified by [10] are discussed. In each case, although the previous work introduced these *s_unit* types in the situation of Java programming, the discussion below explains these concepts were used for TouchDevelop programming.

3.1.1 Void-return *s_units*

An *s_unit* containing a call to another method but does not return a value or whose return value is not saved in a variable is a void-return *s_unit*. The rationale behind why such *s_units* are important is that a method that doesn't return any value must be purely invoked for its side effects; in contrast, methods returning values act as data facilitators [10]. In the code snippet in Fig 5, lines 003 and 005 are void-return *s_units*.

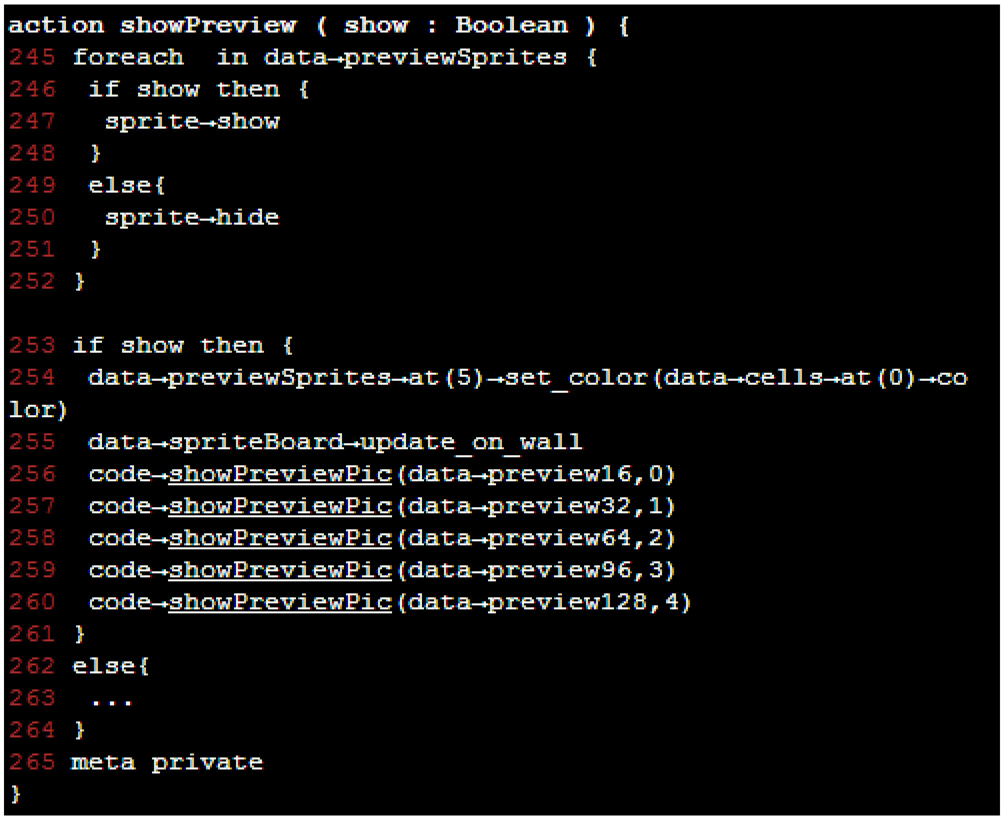


```
action do_something (  ) {
002 s := wall→ask_string("Hello, what is your name?" )
003 ("Hello," ‖s)→post_to_wall
004 pic := senses→take_camera_picture
005 pic→post_to_wall
006 meta private
}
```

Figure 5. void-return s_unit example

3.1.2 Same-action *s_units*

Same-action *s_units* are those *s_units* that contain a method invocation whose *action* term is lexically similar to the *action* term of the method's signature. In the TouchDevelop code snippet in Figure 6, lines 256 to 260 are same-action *s_units* because the *action* of the method invoked in these lines - showPreviewPic has camel case words in common with the method's *action* - showPreview. The rationale behind why same-action *s_units* are important is that the similarity of words implies that the code's purpose may be similar to the overall purpose of the method, as reflected in its name.



```
action showPreview ( show : Boolean ) {
foreach  in data→previewSprites {
 if show then {
  sprite→show
 }
 else{
  sprite→hide
 }
}

if show then {
 data→previewSprites→at(5)→set_color(data→cells→at(0)→co
lor)
 data→spriteBoard→update_on_wall
 code→showPreviewPic(data→preview16,0)
 code→showPreviewPic(data→preview32,1)
 code→showPreviewPic(data→preview64,2)
 code→showPreviewPic(data→preview96,3)
 code→showPreviewPic(data→preview128,4)
}
else{
 ...
}
meta private
}
```

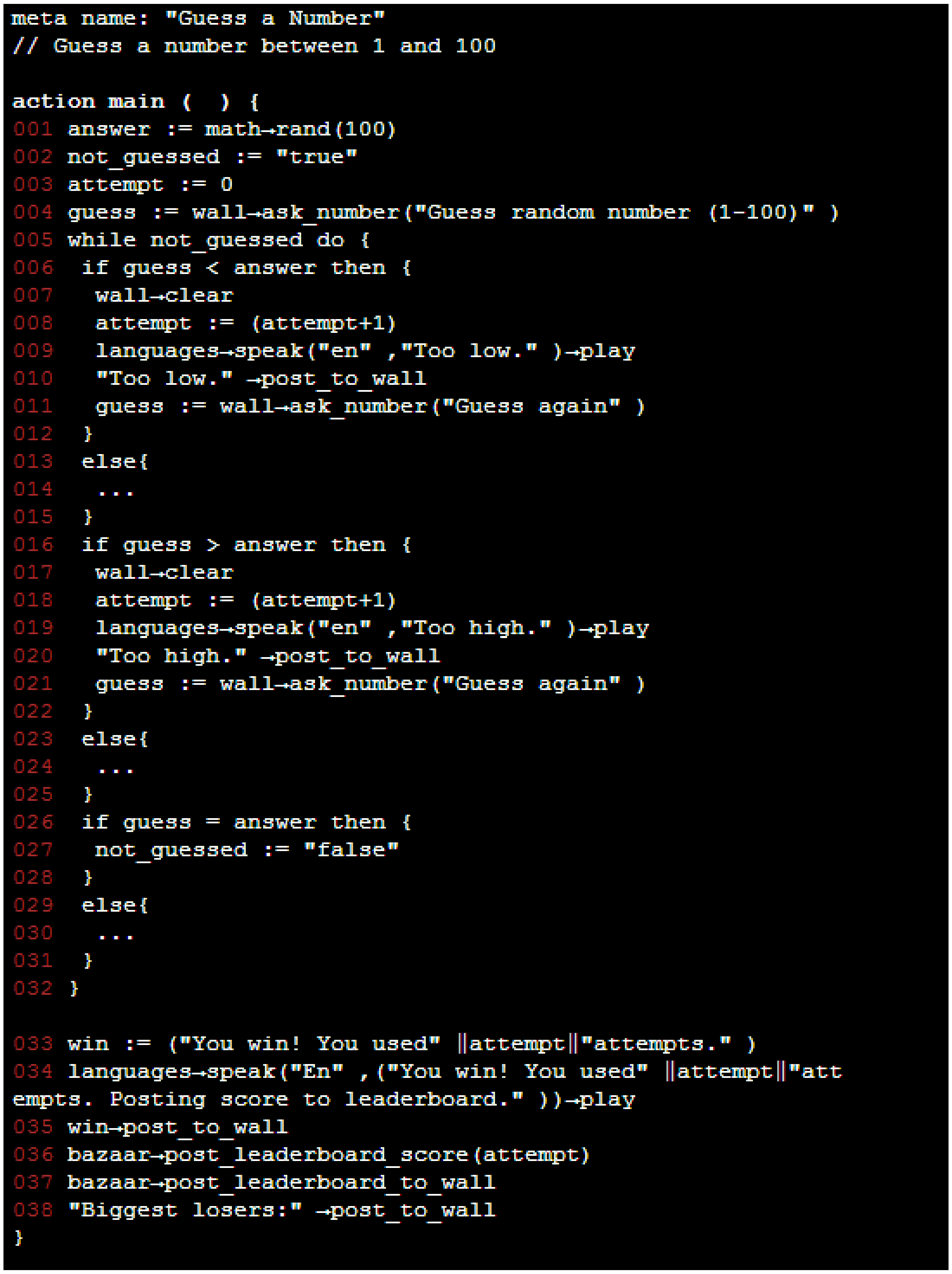
Figure 6. same-action s_unit example

3.1.3 Ending *s_units*

Ending *s_units* of a method refer to those *s_units* after which control exits a method. In case of methods that return some value, ending *s_units* refer to the return statements themselves. In case of methods that don't return any value, ending *s_unit* refers to the line of code that was executed just before the control exited the method. The rationale behind why such *s_units* are important is the observation "that methods often perform a set of actions to accomplish a final action, which is often the main purpose of the method" [10].

3.1.4 Data-facilitating *s_units*

Data facilitating *s_units* are those *s_units* that assign or update the variables used in the previously identified *s_units*. In the TouchDevelop script in Figure 7, lines 34 - 38 are void-return *s_units*. Particularly, in line 36 the variable ‘attempt’ is the *theme*. Hence, those *s_units* that assign value to the variable ‘attempt’ are good candidates for the method’s summary. In this case, lines 8 and 18 are data-facilitating *s_units*.



```
meta name: "Guess a Number"
// Guess a number between 1 and 100

action main (  ) {
answer := math→rand(100)
not_guessed := "true"
attempt := 0
guess := wall→ask_number("Guess random number (1-100)" )
while not_guessed do {
 if guess < answer then {
  wall→clear
  attempt := (attempt+1)
  languages→speak("en" ,"Too low." )→play
  "Too low." →post_to_wall
  guess := wall→ask_number("Guess again" )
 }
 else{
  ...
 }
 if guess > answer then {
  wall→clear
  attempt := (attempt+1)
  languages→speak("en" ,"Too high." )→play
  "Too high." →post_to_wall
  guess := wall→ask_number("Guess again" )
 }
 else{
  ...
 }
 if guess = answer then {
  not_guessed := "false"
 }
 else{
  ...
 }
}

win := ("You win! You used" ‖attempt‖"attempts." )
languages→speak("En" ,("You win! You used" ‖attempt‖"att
empts. Posting score to leaderboard." ))→play
win→post_to_wall
bazaar→post_leaderboard_score(attempt)
bazaar→post_leaderboard_to_wall
"Biggest losers:" →post_to_wall
}
```

Figure 7. data-facilitating s_unit example

3.1.5 Controlling *s_units*

Finally, a controlling *s_unit* is a control statement, recognizable in TouchDevelop as an *s_unit* with one of the following keyword: if, for, while or foreach. The rationale behind why controlling *s_units* are important is that these *s_units* often contain important information about *when* a major action occurs [10]. In particular, a controlling *s_unit* is identified as important only if any of the variables used in the control statement is also used in a previously identified *s_unit* within the block. For example, the controlling *s_unit* in line 005 in the Figure 7 would be identified as important, only if the variable "not_guessed" was used in any of the previously identified *s_unit* in lines 006 - 032.

3.1.6 Procedure to identify important statements

The procedure to identify important statements in a method contains three phases. In the first phase, same-action, ending and void-return *s_units* are identified; in the second phase, data-facilitating *s_units* that correspond to the variables used in the previously identified *s_units* are identified; in the final phase, the controlling *s_units* are identified. At the end of each phase, a few *s_units* are filtered out, according to the procedure specified in [10]. This filtering removes *s_units* responsible for exception handling, object creation, variable initialization, and controlling *s_units* that contain an empty 'else' part from the final set of important *s_units* for a given function.

For example, assume that the controlling *s_units* in lines 006, 016 and 026 in Figure 7 are identified as important at the end of third phase (and before final filtering). The filtering operation following the thrid phase would filter out these controlling *s_units* (lines 006, 016 and 026) from the final set because the “else” part of these controlling *s_units* are empty.

3.2 Tool design

This section reviews the design of the tool for analyzing TouchDevelop scripts. The tool outputs two html files for each input TouchDevelop script. The first html file (Prototype-1) contains a modified version of the code of the input TouchDevelop script with important statements within each function highlighted using different colors; the second html file (Prototype-2) contains a modified version of the same code with unimportant statements within each function hidden.

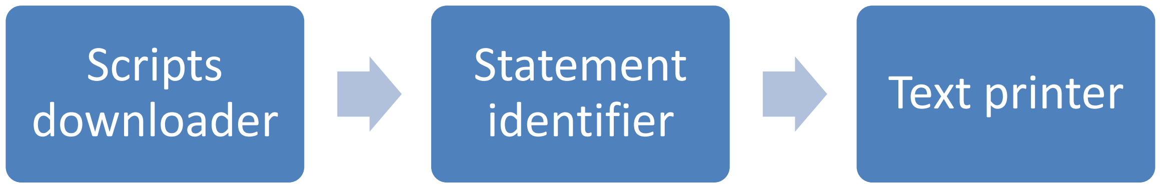
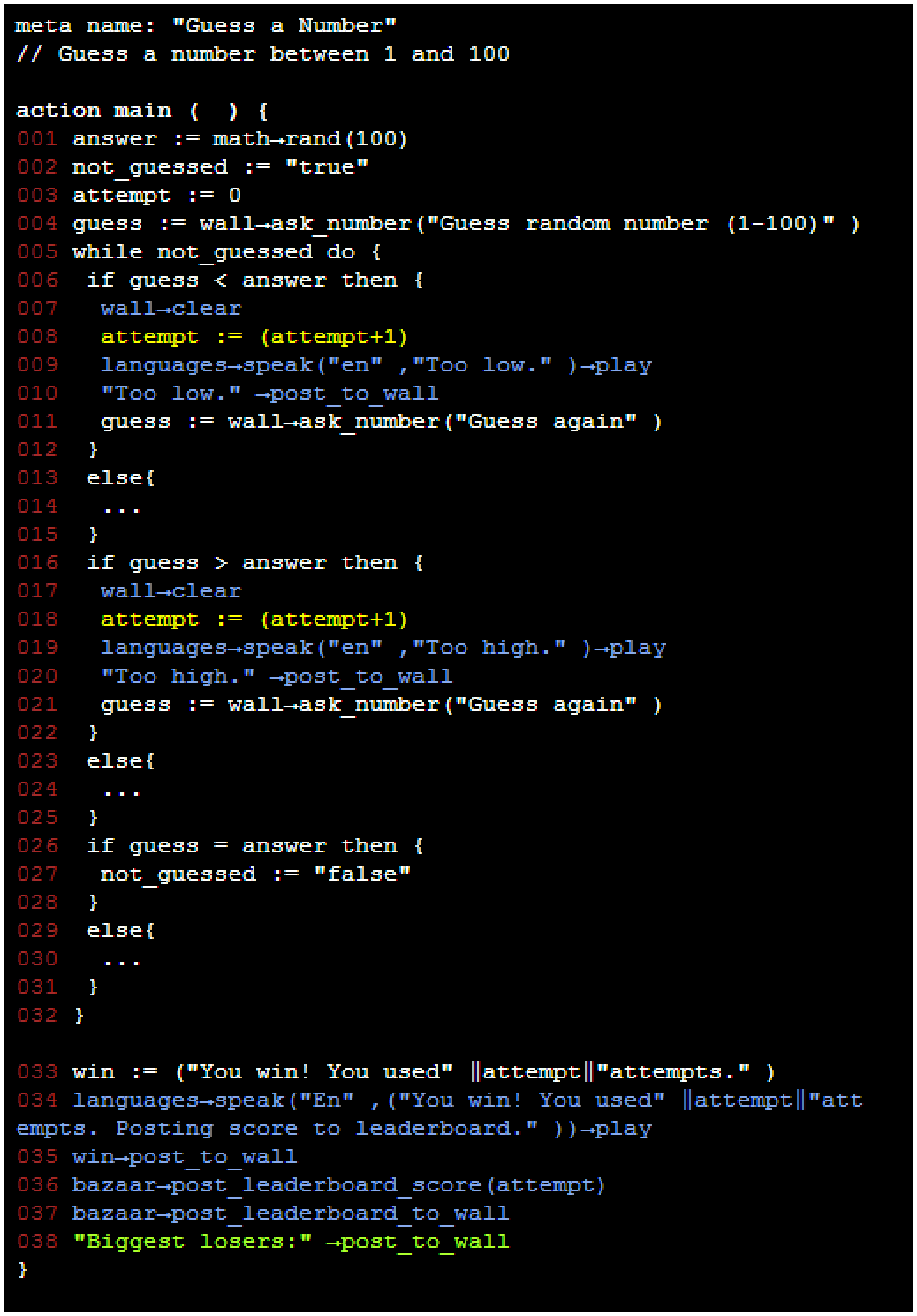


Figure 8. Design of the tool

The tool, as Figure 8 illustrates consists of three separate modules. The scripts downloader module downloads abstract syntax trees representation of TouchDevelop scripts from TouchDevelop’s central repository through a REST API [46] and saves them as text files on the local disk. The second module, “Statement identifier” is basically an implementation of the procedure described in section 3.1.6. This module takes the downloaded text files from the previous stage as input, loops through each function inside a script, and identifies important statements using the rules described in sections 3.1.1 - 3.1.5. The third module, “Text Printer” takes the output of “Statement identifier” as input and generates two html files. The first file is the concatenation of all lines in each method, with highlighting on important lines of code. The second file is the concatenation of all lines of code, along with JavaScript and CSS to hide the unimportant lines of code unless the user toggles them visible (as discussed below).

Figures 9 and 10 show sample html outputs for the TouchDevelop script shown in Figure 7.

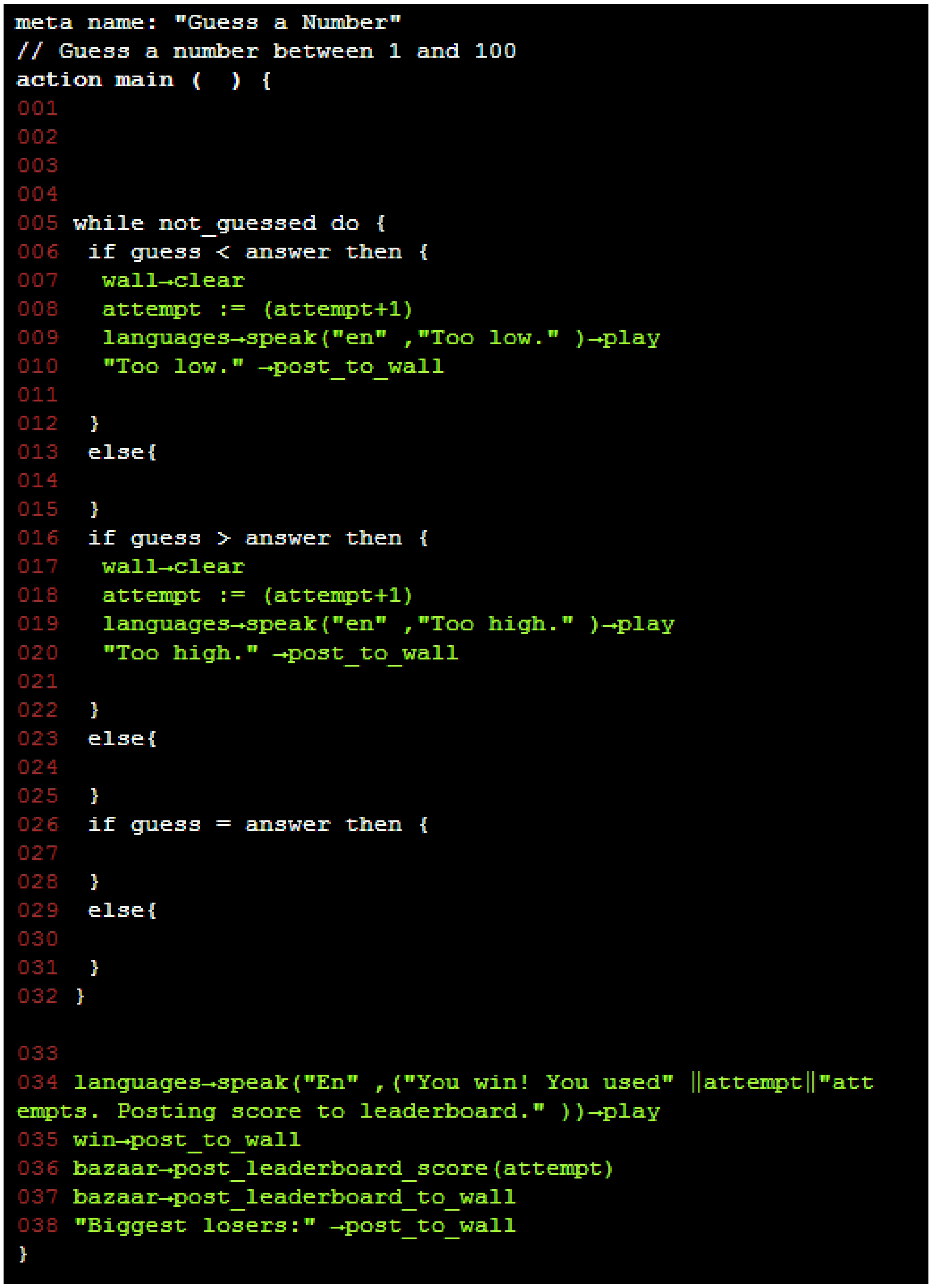


```
meta name: "Guess a Number"
// Guess a number between 1 and 100

action main (  ) {
answer := math→rand(100)
not_guessed := "true"
attempt := 0
guess := wall→ask_number("Guess random number (1-100)" )
while not_guessed do {
 if guess < answer then {
  wall→clear
  attempt := (attempt+1)
  languages→speak("en" ,"Too low." )→play
  "Too low." →post_to_wall
  guess := wall→ask_number("Guess again" )
 }
 else{
  ...
 }
 if guess > answer then {
  wall→clear
  attempt := (attempt+1)
  languages→speak("en" ,"Too high." )→play
  "Too high." →post_to_wall
  guess := wall→ask_number("Guess again" )
 }
 else{
  ...
 }
 if guess = answer then {
  not_guessed := "false"
 }
 else{
  ...
 }
}

win := ("You win! You used" ||attempt||"attempts." )
languages→speak("En" ,("You win! You used" ||attempt||"att
empts. Posting score to leaderboard." ))→play
win→post_to_wall
bazaar→post_leaderboard_score(attempt)
bazaar→post_leaderboard_to_wall
"Biggest losers:" →post_to_wall
}
```

Figure 9. Prototype 1



```
meta name: "Guess a Number"
// Guess a number between 1 and 100
action main (  ) {




while not_guessed do {
 if guess < answer then {
  wall→clear
  attempt := (attempt+1)
  languages→speak("en" ,"Too low." )→play
  "Too low." →post_to_wall

 }
 else{

 }
 if guess > answer then {
  wall→clear
  attempt := (attempt+1)
  languages→speak("en" ,"Too high." )→play
  "Too high." →post_to_wall

 }
 else{

 }
 if guess = answer then {

 }
 else{

 }
}


languages→speak("En" ,("You win! You used" ||attempt||"attempts. Posting score to leaderboard." ))→play
win→post_to_wall
bazaar→post_leaderboard_score(attempt)
bazaar→post_leaderboard_to_wall
"Biggest losers:" →post_to_wall
}
```

Figure 10. Prototype 2

The following color scheme was used for prototype 1.

Type of *s_unit*	**Color used**
Ending s_units	Green-yellow
Void-return s_units	Blue
Same-action s_units	Red
Data-facilitating s_units	Yellow

Table 2. Color scheme used in prototype 1

The line numbers in prototype 2 act as toggle switches - the subjects can see/hide unimportant lines by clicking on them; by default, the unimportant lines are hidden. During the experiment, discussed by Section 4, subjects were explained that they could recognize a hidden line by a line number with no text after it. Future versions of the prototype could include a special icon of some type to show that a line can be clicked to toggle more information.

4 Experiment

This section focuses on the controlled experiment conducted in a laboratory setting to evaluate the performance of the prototypes relative to the baseline (without information features modification). This section reviews the research questions, the design of the experiment, how the subjects were recruited, the tutorial used in the study, how the TouchDevelop scripts used in the study were selected, the design of the program comprehension questions used in the study, and how performance was measured.

4.1 Research questions

The main objective of this research is to investigate the effect of manipulating visual weight of information features in a patch on an end-user programmer’s information foraging and patch comprehension during maintenance tasks. It is not clear if increasing relative weight of important information features will be as helpful for end-user programmers as for professional programmers. On one hand, end-user programmers write code as a secondary task in their work, rather than their main task [12] - as a result, they might be relatively unfamiliar with programming and programming tools making them heavily dependent on tool assistance with information foraging and understanding patches of code. On the other hand, end-user code is often not written with reuse in mind and may not have many information features important enough to highlight or easily identifiable - as a result, highlighting important information features may not help end-user programmers very much.

Modifying the presentation of information features not only affects how the information patch is processed, but also navigation among patches. So, in addition to investigating the effects of manipulating relative weights of information features on within-patch comprehension, it is also important to investigate its effect on an end-user programmer's patch-to-patch navigation. A tool that helps end-user programmers to effectively extract information from a patch, but hinders patch-to-patch navigation is not really useful!

With these objectives in mind, the following research questions were formulated:

RQ1: Does *increasing* the relative weight of *important* information features affect *how much information* an end-user programmer could find during a maintenance task?

RQ2: Does *increasing* the relative weight of *important* information features affect *how quickly* an end-user programmer could find information during a maintenance task?

RQ3: Does *increasing* the relative weight of *important* information features affect *how efficiently* an end-user programmer could find information during a maintenance task?

RQ4: Does *decreasing* the relative weight of *unimportant* information features affect *how much information* an end-user programmer could find for a maintenance task?

RQ5: Does *decreasing* the relative weight of *unimportant* information features affect *how quickly* an end-user programmer could find information during a maintenance task?

RQ6: Does *decreasing* the relative weight of *unimportant* information features affect *how efficiently* an end-user programmer could find information during a maintenance task?

4.2 Design Overview

The experiment was a random-assignment, between-subject user study consisting of 3 distinct groups - one test group for each prototype and one control group as the baseline. All three groups were given the same TouchDevelop scripts and were asked the same set of program comprehension questions. The order of the scripts and program comprehension questions were randomized to account for learning effect. In order to ensure that the groups were reasonably well balanced, tickets were placed into a bowl, and participants randomly chose a ticket to indicate group assignment and task ordering.

During the tutorial, subjects were taught about the APIs in TouchDevelop, how to run scripts in the TouchDevelop app on a Windows smart-phone and how to use the web tool that contained the prototype-generated code and program comprehension questions. The subjects then studied a sample TouchDevelop script and answered two program comprehension questions which were not used in the evaluation. The purpose of this activity was to give the subjects some familiarity with the language, the web tool and the Windows smart-phone.

Following the sample task, subjects studied 3 TouchDevelop scripts and answered program comprehension questions about each script. The web tool recorded the subject’s response to the comprehension questions in the background. Each subject received $10 for their participation. Figure 11 illustrates the experimental design.

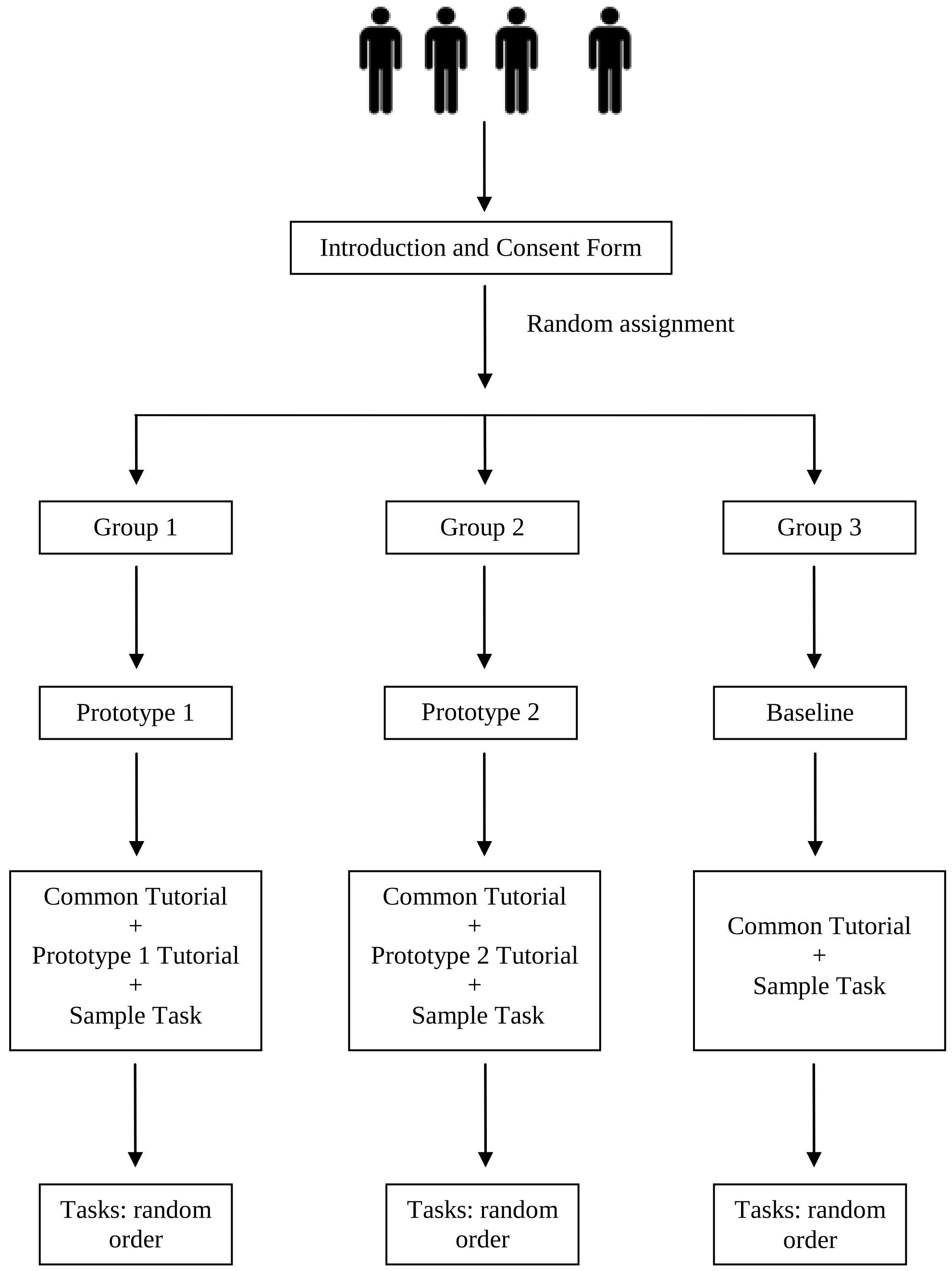

Introduction and Consent Form
Random assignment
Group 1
Group 2
Group 3
Prototype 1
Prototype 2
Baseline
Common Tutorial + Prototype 1 Tutorial + Sample Task
Common Tutorial + Prototype 2 Tutorial + Sample Task
Common Tutorial + Sample Task
Tasks: random order
Tasks: random order
Tasks: random order

Figure 11. Experiment design

4.3 Participants and recruitment

A total of 31 Oregon State University undergraduate and masters' students who take programming classes were recruited in two ways: by sending emails to teachers of the college's programming classes, inviting them to forward the recruitment text to their students, and by emailing the recruitment text directly to relevant people who had previously indicated interest in being notified when an experiment is run. The recruitment email pointed students to a website where they were able to read the consent form, including the inclusion criteria, and to sign up for a timeslot to visit the laboratory. The inclusion criteria were that participants must be able to understand the consent form written in English, must indicate that they are adults, and must indicate that they are not professional programmers. One participant ended up not actually knowing English as well as he thought that he did, so his data was removed from further consideration. This left 30 participants evenly divided among the 3 groups. The subjects were predominantly male (87% of the subjects) and were pre-CS majors. We chose subjects from this population in order to avoid teaching how to program.

4.4 Tutorial

The tutorial used in this study consisted of 3 segments - a segment on TouchDevelop's language features, a segment on how to use the TouchDevelop application to run scripts on the Windows smart-phones and a segment on how to use the web tool which contained both the prototype-generated TouchDevelop script code as well as the program comprehension questions.

While the tutorial on language features and TouchDevelop application was the same for all subjects, the segment on the web tool was different based on the group the subject belonged to. The tutorial on TouchDevelop's language included the following:

- the concept of sprites
- APIs provided by TouchDevelop
- Event listeners provided by TouchDevelop
- Global variables
- organization of code in TouchDevelop scripts
- data-types provided by TouchDevelop

The tutorial on how to use the TouchDevelop app dealt with how to open and access scripts in TouchDevelop app and how to run a script in TouchDevelop app.

The tutorial on how to use the web tool dealt with how to read the code generated by the prototypes. The subjects who belonged to group-1 were taught about the difference between the highlighted and normal lines of code and the meaning of different colors of text in the prototype-generated frame. The subjects who belonged to group-2 were taught about the difference between hidden and unhidden lines of code and how to view the hidden lines of code within the prototype-generated frame. The interface of the web tool was kept uniform for all three treatments except for the frame that contained the prototype-generated script code.

4.5 Experiment scripts

In order to cover a range of patch comprehension and patch-to-patch navigation situations, 3 different TouchDevelop scripts with different sizes were selected. The three scripts were chosen from a pool of hundred scripts that were previously studied [19]. Script-1 (https://www.touchdevelop.com/somi) , which was small, contained only one function and 38 lines of code - hence all information foraging was within one patch. Script-3 (https://www.touchdevelop.com/ujqx), which was large, contained 18 functions, 6 event-listeners and 326 lines of code that included 35 function calls – hence, understanding Script-3 required much more patch-to-patch navigation compared to Script-1. The other script, Script-2 (https://www.touchdevelop.com/ujqx), achieved a middle-ground between Script-1 and Script-3. It contained 4 functions, 1 event-listener and 122 lines of code.

The size of these scripts were fairly representative of the range of script sizes in TouchDevelop repository. According to a recent study on TouchDevelop scripts [32], 72.6% of the scripts in TouchDevelop repository are less than 100 lines of code (Script-1) and 24.8% of the scripts are between 100 and 500 lines of code (Script-2 and Script-3)

The scripts' functionality was also representative several important functional categories demonstrated by scripts in the repository. Script-1 was a game where the user guesses a number between 1 and 100 randomly chosen by the application. Script-2 was a non-entertainment utility application that acted as a timer. Script-3 was an application that lets the user build and save image sprites on a 16*16 grid. According to previous research [19], scripts related to games, utility functions, and image manipulation are fairly common in the TouchDevelop platform.

4.6 Program comprehension questions

The objective of this study is to investigate the effect of manipulating the presentation of information features in a patch on real-world maintenance tasks. Hence, the program comprehension questions were designed around actual changes made by TouchDevelop users over the life of the scripts in the repository. Two different researchers studied and agreed upon what these changes involved in previous research [19]. Previous research on questions that programmers ask during software evolution tasks [2] was used to inform the design of the program comprehension questions. All of the program comprehension questions required the subjects to find specific information (often a line of code) in the script as would be the case prior to accomplishing the maintenance changes that was investigated in the previous study [19]. The list of questions used in the study can be found in Appendix A. (There was 1 question for the small script, 2 for the medium script, and 6 for the large script.) The questions were reviewed and it was informally verified that the users theoretically could find the correct answers primarily by looking at the “important” statements identified by the prototypes, indicating that the existing algorithm that was adapted for the prototypes did have a plausible chance of helping participants. Note that, the program comprehension questions did not test the control flow of the tasks.

4.7 Measures and analysis

To compare the performance of the subjects from the 3 different groups, three measures were used – program comprehension correctness, foraging time, and information gained per total time. Program comprehension correctness was determined by comparing subjects' answer with the answer key. One point was given to a correct answer and zero to an incorrect answer. Some answers contained two parts and if a subject got only one of them correct, 0.5 points was rewarded. The time taken for the tasks was also considered as a measure of success. For the measure of foraging time, the web tool recorded the time taken for the tasks in the background in seconds. The ratio of these two measures, correctness and time taken was used as the third measure of success – rate of information gained per time. (This third measure was scaled by 10000 so that it generally fell in the range of 100-10000.)

Due to the fairly small sample and likelihood that data would not be normally distributed, a Kruskal-Wallis nonparametric statistical test was implemented with a two-tailed ANOVA of each measure rank versus two factors: treatment and task (script size). Even though ANOVA is a parametric test, rank transformation is shown to add robustness against non-normality, outliers and unequal variance to ANOVA [58]. This analysis was used to see if there were any statistically significant differences among the 3 treatment groups while controlling for script size. (The sample size was insufficient to test for an interaction term between treatment and script size.) For each measure and each combination of treatment and script size, the mean of the measurements was computed so that these averages could be reported if statistically significant differences are found. Table 3 summarizes the research questions and measures.

Research Question	Measure used to answer the research question
RQ1: Does increasing the relative weight of important information features affect *how much information* an end-user programmer could find during a maintenance task?	Program comprehension correctness
RQ4: Does decreasing the relative weight of unimportant information features affect *how much information* an end-user programmer could find for a maintenance task?	
RQ2: Does increasing the relative weight of important information features affect *how quickly* an end-user programmer could find information during a maintenance task?	Foraging time
RQ5: Does decreasing the relative weight of unimportant information features affect *how quickly* an end-user programmer could find information during a maintenance task?	
RQ3: Does decreasing the relative weight of unimportant information features affect *how efficiently* an end-user programmer could find information during a maintenance task?	Rate of information gain
RQ6: Does decreasing the relative weight of unimportant information features affect *how efficiently* an end-user programmer could find information during a maintenance task?	

Table 3. Summary of measures used to answer the research questions

5 Results and Discussion

This section discusses the results for each question, possible explanations for the results, and threats to validity.

5.1 Results

RQ1: Does increasing the relative weight of important information features affect how much information an end-user programmer could find during a maintenance task?

There was a strong evidence (p-value = 0.01, F-value = 7.35) for a significant difference between Group 1 (highlighting important statements) and Group 3 (baseline) on program comprehension correctness. There was no evidence (p-value = 0.89, F-value = 0.1217) that script size had a statistically significant effect. This indicates that, across the range of script sizes that was investigated, increasing the relative weight of important information features *did influence* the amount of information an end-user programmer could find. Below, Figure 12 summarizes the mean and range of scores that subjects got for the small, medium, and large scripts.

Summing across all script sizes, the subjects in Group 1, who had the highlighting version of the prototype, scored a total of 5.1 points (with subscores of 0.9/1.1/3.1 for small/medium/large scripts, respectively). In contrast, subjects in Group 3, who had the baseline prototype, scored a total of only 3.45 points (with subscores of 0.65/0.7/2.1).

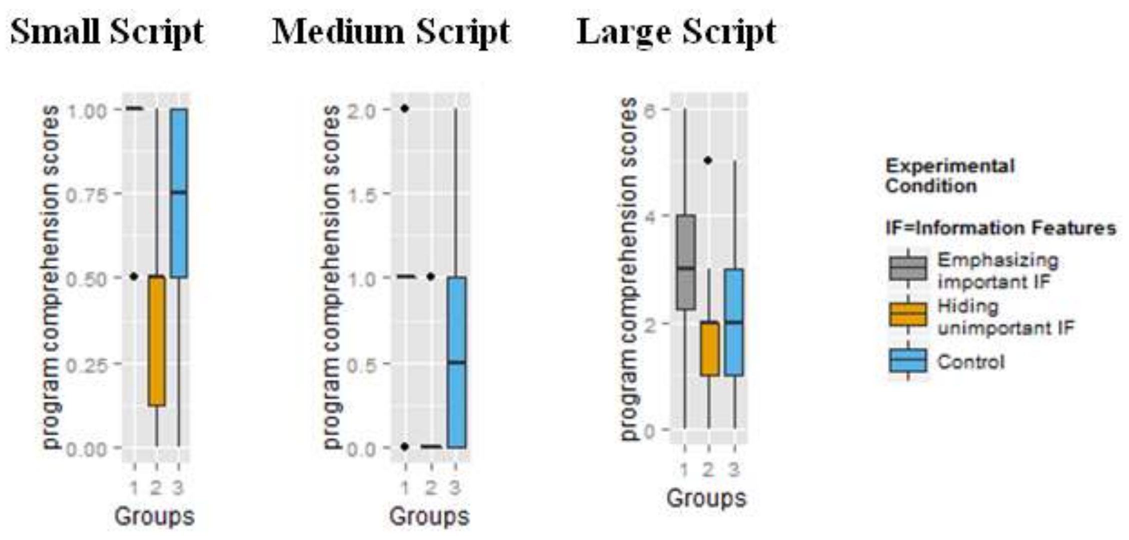


Figure 12. Comparing groups on program comprehension score for Script 1, 2 and 3 respectively

RQ2: Does increasing the relative weight of important information features affect how quickly an end-user programmer could find information during a maintenance task?

There was no evidence (p-value = 0.7, F-value = 0.12) for a significant difference between Group 1 (highlighting important statements) and Group 3 (baseline) on foraging time. There was also no evidence (p-value = 0.98, F-value = 0.0182) for a significant difference between the two groups across script size. This indicates that these two factors *did not influence* an end-user programmer's foraging time.

RQ 3: Does decreasing the relative weight of unimportant information features affect how efficiently an end-user programmer could find information during a maintenance task?

There was some evidence (p-value = 0.03, F-value = 5.06) for a significant difference between Group 1 (highlighting important statements) and Group 3 (baseline) on rate of information gain. This indicates that increasing the relative weight of important information features did influence the efficiency of an end-user programmer's information foraging. As Figure 13 shows, the mean rate of information gain for subjects from Group 1 (859.4/842.9/444.78 for small/medium/large scripts) was greater than the mean rate of information gain for Group 3 (637.2/361.7/248) for scripts 1,2 and 3 respectively. There was no evidence (p-value = 0.88, F-value = 0.13) for a significant difference between the two groups across script size.

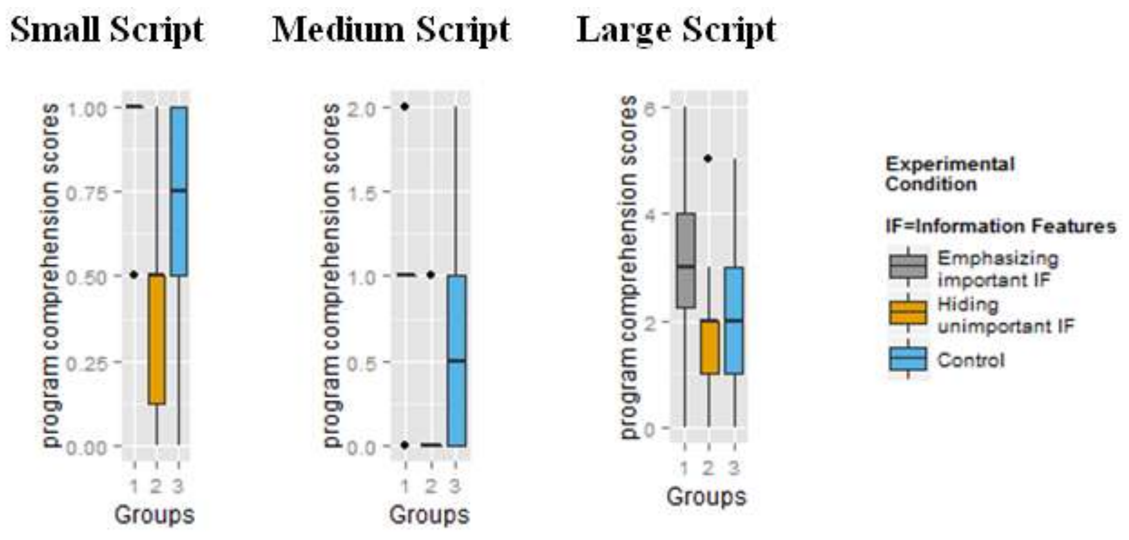


Figure 13. Comparing groups on rate of information gain for Script 1, 2 and 3 respectively (in units of "points per 10000 seconds").

RQ4: Does decreasing the relative weight of unimportant information features affect how much information an end-user programmer could find during a maintenance task?

There was weak/suggestive evidence (p-value = 0.08, F-value = 3.15) for a significant difference between Group 2 (hiding unimportant statements) and Group 3 (baseline) on program comprehension correctness. This indicates that decreasing the relative weight of unimportant information features has a *weak but inconclusive* influence on the amount of information an end-user programmer could find. Figure 12, above, compares the groups. There was no evidence (p-value = 0.9, F-value = 0.11) for any significant difference between the two groups across script size.

RQ5: Does decreasing the relative weight of unimportant information features affect how quickly an end-user programmer could find information during a maintenance task?

There was no evidence (p-value = 0.28, F-value = 1.14) for a significant difference between Group 2 (hiding unimportant statements) and Group 3 (baseline) on foraging time. This indicates that decreasing the relative weight of unimportant information features *did not influence* an end-user programmer's foraging time. There was also no evidence (p-value = 0.94, F-value = 0.06) for a significant difference between the two groups across scripts.

RQ 6: Does decreasing the relative weight of unimportant information features affect how efficiently an end-user programmer could find information during a maintenance task?

There was no evidence (p-value = 0.04, F-value = 4.33) for a significant difference between Group 2 (hiding unimportant statements) and Group 3 (baseline) on rate of information gain. This indicates that decreasing the relative weight of unimportant information features *did influence* the efficiency of an end-user programmer's information foraging. As Figure 13 shows, above, the mean rate of information gain for subjects from Group 1 (368.8/112.8/215.6 for small/medium/large scripts) was consistently lesser than the mean rate of information gain for Group 3 (859.4/842.9/444.78). There was no evidence (p-value = 0.99, F-value = 0.006) for a significant difference between the two groups across scripts.

Tables 4 and 5 summarize all of the results that were obtained. Note that each value in the third row of Table 4 differs from the ratio of the corresponding values in the first two rows because the third row shows the average of individual subjects' ratios rather than the ratio of the averages.

Comparing measure between groups	Highlighting important statements (N = 10)			Hiding unimportant statements (N = 10)			Baseline (N = 10)		
Script	1	2	3	1	2	3	1	2	3
Score (points)	0.9	1.1	3.1	0.45	0.2	1.9	0.65	0.7	2.1
Time (seconds)	117.7	192.1	769.8	128.1	196.1	862.6	129.7	178.2	802.3
Gain rate (points/seconds)	859.4	842.9	444.78	368.8	112.8	215.6	637.2	361.7	248

Table 4. Mean performance of participants on each measure for each script size

			P values	
Research Question	Measure	Treatment groups	Treatment	Script size
RQ1: Does **increasing** the relative weight of **important** information features affect **how much information** an end-user programmer could find during a maintenance task?	Comprehension score	1 vs 3	0.01	0.89
RQ2: Does **increasing** the relative weight of **important** information features affect **how quickly** an end-user programmer could find information during a maintenance task?	Time taken	1 vs 3	0.70	0.98
RQ3: Does **increasing** the relative weight of **important** information features affect **how efficiently** an end-user programmer could find information during a maintenance task?	Rate of information gain	1 vs 3	0.03	0.80
RQ4: Does **decreasing** the relative weight of **unimportant** information features affect **how much information** an end-user programmer could find for a maintenance task?	Comprehension score	2 vs 3	0.08	0.90
RQ5: Does **decreasing** the relative weight of **unimportant** information features affect **how quickly** an end-user programmer could find information during a maintenance task?	Time taken	2 vs 3	0.28	0.94
RQ6: Does **decreasing** the relative weight of **unimportant** information features affect **how efficiently** an end-user programmer could find information during a maintenance task?	Rate of information gain	2 vs 3	0.04	0.99

Table 5. Summary of results from the six statistical tests, with shading to indicate differences that were significant at $p < 0.05$

5.2 Discussion

As discussed in Section 2, researchers have already proposed tools based on highlighting and code summarization to help professional programmers. Highlighting had also been shown to be helpful to end-user programmers during debugging. Based on the success of these tools, it might be expected that highlighting important statements and removing unimportant statements could possibly help end-user programmers. On the other hand, highlighting could overwhelm users, and hiding code statements could reduce their ability to understand the program.

Results from this research indicate that highlighting important statements did help end-user programmers to reduce foraging cost in the study. It helped end-user programmers find *more* information, though *not in significantly less time*, leading to an overall *faster rate*. This may indicate that highlighting increases the benefit-to-cost ratio of processing an information patch not by reducing the cost of processing the patch, but by increasing the benefit of processing the patch (that is, by returning more value per unit effort). Future research is needed to investigate this interpretation of the results and to investigate *why* highlighting seemed to help subjects get more information out of visiting patches.

Unlike highlighting important statements, hiding unimportant statements didn't seem to help to reduce foraging cost. In fact, hiding unimportant statements *reduced* the rate of information gain, even though a review of the questions confirmed that users theoretically could find the correct answers mainly by looking at only the unhidden (i.e., important) statements. The mean comprehension score for the baseline group was marginally though not significantly better than the mean score for the group with hidden unimportant statements; the mean time taken was nearly identical for the two groups. So although there was not a statistically significant effect on either of the first two measures, the ratios of these did a slightly statistically significant difference (at p=0.04, as shown in Table 4). This result might be explained by previous research [56] on sense-making in end-user programmers' debugging strategies. This work suggests that male end-user programmers (87% of the subjects) exhibit a "selective information processing" style, where they tend to gather information depth-first rather than comprehensively reviewing all available information breadth-first before proceeding. Perhaps the subjects did not take the time to unhide unimportant statements even when doing so would have been slightly helpful for making sense of the important statements where the true answers to the questions were located. So perhaps, because less important portions within the patch were not immediately visible, they could not acquire all of the information from unimportant statements that might be needed to fully comprehend the program as a whole, resulting in poor performance. These results suggest that even unimportant statements might be needed to help end-user programmers understand the important statements. This hypothesis would need to be investigated by future studies.

5.3 Threats to validity

One strength of the study is that the specific scripts and tasks that were chosen were based on prior work investigating what kinds of scripts and maintenance tasks are common in the TouchDevelop environment. However, this experiment was a controlled lab study with artificial limitations and the subjects may not be representative of actual TouchDevelop users in the real world. It is also possible that the results might not generalize to other kinds of programming tools, particularly visual programming tools rather than scripting tools, since information foraging in visual programming environments is not nearly as well-investigated as foraging in textual programming environments. Further studies could investigate whether results apply to other users and other programming environments.

This study investigated the effect of manipulating the visual weights of information features on information foraging. The subject's information foraging was tested by asking program comprehension questions that reflect the amount of information the subject has foraged and understood. To help ensure construct validity, two different researchers studied and agreed upon the meaning of code changes in a previous study [19] to develop the program comprehension questions, which were also informed by previous research on questions that programmers ask during software evolution tasks [2]. However, the experiment did not actually ask users to complete the actual maintenance tasks, so although the results reveal the effects of highlighting and cue-removal on information foraging, it cannot be claimed that the results imply anything about the effects of highlighting and cue-removal on actual programming task completion. Finally, the program comprehension questions didn't test any tasks that involved large external data structures (such as databases). Future studies can investigate if the results apply to such tasks.

6 Conclusion and Future research opportunities

This work investigated how highlighting important information features and hiding unimportant information features would affect an end-user programmer's foraging during real-world maintenance tasks. Particularly, this research focused on end-user programmers because their information foraging has received minimal attention in prior work. Conclusions of this work are:

- highlighting important information features is beneficial to end-user programmers during foraging tasks.

- highlighting important information features affects the amount of information foraged and the rate of information gain.

- removing unimportant features is detrimental to an end-user programmer's rate of information gain.

Integrating the proposed technique into code editors could be beneficial to end-user programmers. Identifying and highlighting important information features as the end-user programmer types his program in the code editor may help them verify design specifications and prevent bugs at an early stage; automatically highlighting these important statements in the code editor may benefit during later maintenance tasks. Future works may investigate how well the proposed technique helps to prevent bugs and to reduce the time needed for maintenance.

As a second example of how these results could help guide tool improvements, highlighting important statements could also be beneficial during debugging tasks. During a debugging session, when an end-user programmer is stepping through an execution, the debugging tool could highlight important lines of code that were either just executed or that might be executed soon, in order to help the programmer focus attention on the lines of code that might matter most for recognizing and understanding a bug. Future work can investigate if highlighting important statements like this within an end-user program helps in debugging tasks.

Also, these results might be used to improve existing end-user programming tools for code reuse. For example, end-user programmers often depend on adapting existing code during reuse, called white-box reuse. Many end-user programming environments provide recommendations for reusable code based on a search query. After the environment has returned several recommendations, the end-user programmer still has to evaluate one or few returned end-user programs and decide if any of them is suitable for white-box reuse. Evaluating a few unfamiliar end-user programs and choosing one among many can be a time consuming process for the end-user programmer. While there has been some previous research on identifying reusable end-user programs [59,60], there are no prior research that investigate how these returned end-user programs can be effectively presented to the end-user for further evaluation. Results from this work suggests that future work should investigate if identifying and highlighting important statements within the returned end-user programs can help end-user programmers effectively and efficiently evaluate the returned results. Many subjects using prototype 1 (highlighting) indicated that the statements highlighted in blue - the void return *s_units* (function calls with side effects) were useful in answering the program comprehension questions after completing the experiment. Future work can further investigate the reason behind this and how it can be used to improve end-user programming tools.

Finally, it would also be interesting to investigate if the results can be generalized to visual programming environments for end-user programmers such as Scratch [22] where important information features would be symbols, code magnets rather textual statements. The research could try to discover what kinds of highlighting make most sense to end-user programmers in a visual language, and whether this highlighting helps with program comprehension.

Bibliography

1. Ko, A., Myers, B., Coblenz, M., & Aung, H. (2006, Dec.). An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. Software Engineering, IEEE Transactions, 32, 971 - 987.

2. Sillito, J., Murphy, G., & De Volder, K. (2006). Questions programmers ask during software evolution tasks. Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, 23 - 24.

3. Ko, A., Myers, B., & Aung, H. (2004). Six Learning Barriers in End-User Programming Systems. *Visual Languages and Human Centric Computing, 2004 IEEE Symposium* , 199 - 206.

4. Ko, A., Myers, B. (2004). Designing the whyline: a debugging interface for asking questions about program behavior. *Proceedings of the SIGCHI conference on Human factors in computing systems*, 151-158.

5. Cubranic, D., Murphy, G., Singer, J., & Booth, K. (2005, June). Hipikat: a project memory for software development. Software Engineering, IEEE Transactions on, 31 (6), 446 - 465.

6. Henninger, S. (1994). Using iterative refinement to find reusable software. Software, IEEE, 11 (5), 48 - 59.

7. Holmes, R., & Walker, R. (2007). Supporting the investigation and planning of pragmatic reuse tasks. Proceedings of the ACM/IEEE International Conference on Software Engineering, 447–457.

8. Simon, F., Steinbruckner, F., & Lewerentz, C. (2001). Metrics based refactoring. Software Maintenance and Reengineering, 2001. Fifth European Conference on, 30 - 38.

9. Pirolli, P., & Card, S. (1999). Information foraging models of browsers for very large document spaces. *In Proceedings of the Working Conference on Advanced Visual Interfaces*, 83–93

10. G., Sridhara; E., Hill; D., Muppaneni; L., Pollock; K., Vijay-Shanker (2010). Towards Automatically Generating Summary Comments for Java Methods. *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 43-52.

11. Ko, A., Myers, B., & Aung, H. (2004). Six Learning Barriers in End-User Programming Systems. *Visual Languages and Human Centric Computing, 2004 IEEE Symposium* , 199 - 206.

12. Scaffidi, C., Shaw, M., & Myers, B. (2005). Estimating the numbers of end users and end user programmers. *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium* , 207 - 214.

13. Petre, M., & Blackwell, A. (2007). Children as unwitting end-user programmers. *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing* , 239–242.

14. Wiedenbeck, S. (2005). Facilitators and inhibitors of end-user development by teachers in a school environment. *IEEE Symposium on Visual Languages and Human-Centric Computing* , 215-222.

15. Segal, J. (2005). When Software Engineers Met Research Scientists: A Case Study. *Empirical Software Engineering* , 517-536.

16. Wang, G. Y. (2009). Mashroom: end-user mashup programming using nested tables. *Proceedings of the 18th international conference on World wide web* , 861-870.

17. Wong, J. a. (2007). Making mashups with marmite: towards end-user programming for the web. *Proceedings of the SIGCHI conference on human factors in computing systems* , 1435-1444.

18. G. Little, T. L. (2007). Koala: Capture, share, automate, personalize. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* , 943-946.

19. Athreya, B., Bahmani, F., Diede, A., & Scaffidi, C. (2012). End-user programmers on the loose: A study of programming on the phone for the phone.

Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium , 75 - 82.

20. Magnuson, B. (2010). Building blocks for mobile games: a multiplayer framework for App inventor for Android. *master thesis, Massachusetts Institute of Technology* .

21. Cooper, S. D. (2000). Developing algorithmic thinking with Alice. *Information Systems Educators Conference* , 506-539.

22. Resnick, M. (2009). Scratch: Programming for all. *Communications of the ACM* , 60-67.

23. http://pipes.yahoo.com/pipes/pipes.popular

24. Nardi., B. (1993). *A Small Matter of Programming:Perspectives on End User Computing.* MIT Press.

25. Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies* , 543-554.

26. Shneiderman, B. a. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Parallel Programming* , 219-238.

27. Von Mayrhauser, A. a. (1995). Program comprehension during software maintenance and evolution. *Computer, IEEE* , 44-55.

28. Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology* , 295-341

29. Letovsky, S. (1996). Cognitive Processes in Program Comprehension. *Empirical Studies of Programmers: 1st Workshop* , 58.

30. Fleming, S. D., Scaffidi, C., Piorkowski, D., Burnett, M., Bellamy, R., Lawrance, J., et al. (2007). An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks. *ACM Transactions on Software Engineering and Methodology (TOSEM)* .

31. Tillmann, N. M. (2011). TouchDevelop: Programming cloud-connected mobile devices via touchscreen. *Symp on New Ideas, New Paradigms, Reflections on Programming and Software* , 49-60.

32. S. Li, T. X. (2013). A Comprehensive Field Study of End-User Programming on Mobile Devices. *VL/HCC*.

33. Pennington, N., & Grabowski, B. (1990). The tasks of programming. *Psychology of Programming* , 45-61.

34. Pirolli, P. (2007). *Information Foraging Theory: Adaptive Interaction with Information.* Oxford University Press.

35. Aula, A., Jhaveri, N., & Kaki, M. (2005). Information search and re-access strategies of experienced Web. *Proceedings of the International Conference on World Wide Web* , 583–592.

36. Jeng, J.-J., & Cheng, B. H. (1995). Specification matching for software reuse: a foundation. *SSR '95 Proceedings of the 1995 Symposium on Software reusability* , 97-105.

37. Podgurski, A., & Pierce, L. (1993). Retrieving reusable software by sampling behavior. *ACM Transactions on Software Engineering and Methodology (TOSEM)* , 286 - 303.

38. Reiss, S. (2009). Semantics-based code search. *In Proceedings of the 31st International Conference on Software Engineering* , 243 - 253.

39. Hill, E., Pollock, L., & Vijay-Shanker, K. (2009). Automatically capturing source code context of NL-queries for software maintenance and reuse. *ICSE '09 Proceedings of the 31st International Conference on Software Engineering* , 232-242.

40. Cottrell, R. W. (2008). Semi-automating small-scale source code reuse via structural correspondence. *In Proceedings of the ACM/IEEE International Symposium on Foundations of Software Engineering* , 214–225.

41. Piorkowski, D. F. (2013). Reactive information foraging: an empirical investigation of theory-based recommender systems for programmers. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* , 3063-3072.

42. Piorkowski, D., Fleming, S., Scaffidi, C., John, L., Bogart, C., John, B., et al. (2011). Modeling programmer navigation: A head-to-head empirical evaluation of predictive models. *Visual Languages and Human-Centric Computing (VL/HCC)* , 109-116.

43. Ducasse, S. R. (1999). A language independent approach for detecting duplicated code. *International Conference on Software Maintenance* , 109–118.

44. Prabhakararao, S., Cook, C., Ruthruff, J., Creswick, E., Main, M., Durham, M., et al. (2003). Strategies and behaviors of end-user programmers with interactive fault localization. *Human Centric Computing Languages and Environments, 2003. Proceedings. 2003 IEEE Symposium* , 15-22

45. Hill, E., Pollock, L., & Vijay-Shanker, K. (2009). Automatically capturing source code context of NL-queries for software maintenance and reuse. ICSE '09 Proceedings of the 31st International Conference on Software Engineering , 232-242.

46. https://www.touchdevelop.com/help/cloudservices

47. Haiduc S., A. J. (2010). Supporting Program Comprehension with Source Code Summarization. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering , 223-226.

48. S. Rastkar, G. M. (2011). Generating Natural Language Summaries for Crosscutting Source Code Concerns. IEEE International Conference on Software Maintenance (ICSM) , 103 - 112.

49. Kireyev, K. (2008). Using Latent Semantic Analysis for Extractive Summarization. In Proceedings of Text Analysis Conference .

50. Steinberger, J. a. (2009). Update Summarization Based on Latent Semantic Analysis. In Proceedings of 12th International Conference - Text, Speech and Dialogue .

51. Fisher II, M. R. (2005). The EUSES Spreadsheet Corpus: A Shared Resource for Supporting Experimentation with Spreadsheet Dependability Mechanisms. In 1st Workshop on End-User Software Engineering .

52. Jean, M., Hall, J. (1996). A Risk and Control-Oriented Study of the Practices of Spreadsheet Application Developers. Proc. 29th Hawaii Intl. Conf. System Sciences , 364-373.

53. http://docs.oracle.com/javase/tutorial/java/nutsandbolts/if.html

54. Scaffidi C., C. A. (2008). Using topes to validate and reformat data in end-user programming tools. Proceedings of the 4th international workshop on End-user software engineering , 11-15.

55. Biggerstaff T. J., R. C. (1989). Reusability framework, assessment, and directions. Software reusability: vol. 1, concepts and models , 1-17.

56. Grigoreanu, V., Burnett, M., Wiedenbeck, S., Cao, J., Rector, K., & Kwan, I. (2012). End-User Debugging Strategies: A Sensemaking Perspective. Transactions on Computer-Human Interaction (TOCHI) .

57. Olston, C., & Chi, E. (2003). ScentTrails: Integrating browsing and searching on the Web. ACM Transactions on Computer-Human Interaction (TOCHI) , 177-197.

58. Conover, W. J., & Iman, R. L. (1981). Rank Transformations as a Bridge between Parametric and Nonparametric Statistics. The American Statistician , 124-129.

59. Scaffidi, C., & Shaw, M. (2009). Inferring reusability of end-user programmers' code from low-ceremony evidence. End User Programming for the Web Workshop.

60. Scaffidi, C., Bogart, C., Burnett, M., Cypher, A., Myers, B., & Shaw, M. (2008). Characterizing reusability of end-user web macro scripts. International Workshop on Recommendation Systems for Software Engineering .

Appendix A - List of program comprehension questions

Small script

1. Suppose you decide to modify the program (source code displayed onscreen) such that the final score of a player is given by the formula: (1000 - time spent playing the game). Please write down the line numbers after which you would insert code to accurately save the time at which the game starts and the time at which the game ends.

Large Script

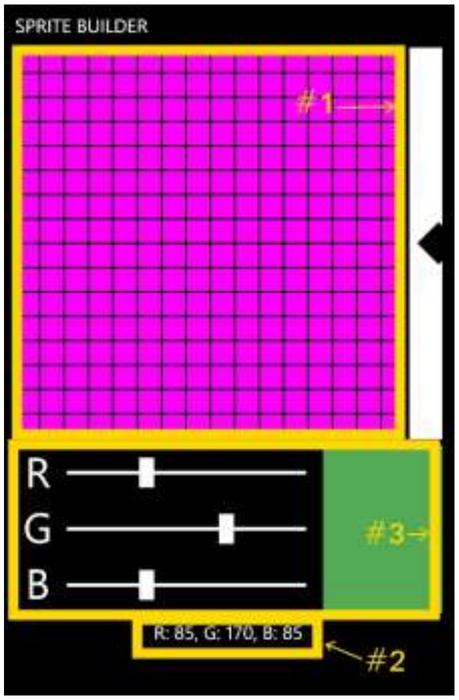


Figure 14. Image used in the study

2. What is the **line number** where a global variable is defined for the UI element marked as #1 (in yellow) in the screenshot (see Figure 14)?

3. What is the **line number** where a global variable is defined for the UI element marked as #2 (in yellow) in the screenshot (see Figure 14)?

4. What is the **line number** where a global variable is defined for the UI element marked as #3 (in yellow) in the screenshot (see Figure 14)?

5. Note that the ‘Options’ screen (on the phone) has buttons for picking a color and previewing. Write down the **line number** you would modify so that the distance between the top of the screen and each of the buttons is given by the following formula (height of the button * (index of the button + 25))

6. Note that the ‘Options’ screen (on the phone) has buttons for picking a color and previewing. Suppose you want to add a third button to this screen. Enter the **line number after** which you would insert code to add the third button to the end of the Options

7. Write down the **line number** after which you would insert code to handle the functionality of a third button.

Medium Script

8. Where (line number or value) does the sound stored in the global variable s-main come from? (program source code displayed onscreen)

9. Please write down the **line number** that is responsible for playing the sound when the timer finishes (program source code displayed onscreen)