View-based Editing of Variational Code

By
Miles Van de Wetering

A THESIS

submitted to

Oregon State University

University Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science
(Honors Scholar)

Presented May 22, 2017
Commencement June 2017

AN ABSTRACT OF THE THESIS OF

Miles Van de Wetering for the degree of Honors Baccalaureate of Science in Computer Science presented on May 22, 2017. Title: View-based Editing of Variational Code

Abstract approved:

Eric Walkingshaw

This paper discusses the merits of providing users variational views when editing variational code. I provide a plugin for the popular Atom Integrated Development Environment (IDE) which replaces #ifdef annotations commonly used by the C PreProcessor (CPP) with colored backgrounds, thus reducing code clutter and attempting to help programmers quickly distinguish code that belongs to different features. I also provide a number of helpful features designed to help the programmer create, remove, and refactor feature code. Finally, I present a user study conducted in order to determine how helpful each of the two main features (code folding and background color) are to programmers - it was determined that while there were no significant differences in efficiency or accuracy, the user experience was considerably enhanced.

Key Words: variation, ifdef, editing, hci, software product lines

Corresponding e-mail address: vandewmi@oregonstate.edu

©Copyright by Miles Van de Wetering
June 5, 2017
All Rights Reserved

View-based Editing of Variational Code

By
Miles Van de Wetering

A THESIS

submitted to

Oregon State University

University Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science
(Honors Scholar)

Presented May 22, 2017
Commencement June 2017

Honors Baccalaureate of Science in Computer Science project of Miles Van de Wetering presented on May 22, 2017

APPROVED:

Eric Walkingshaw, Mentor, representing Computer Science

Mike Bailey, Committee Member, representing Computer Science

Carlos Jensen, Committee Member, representing Computer Science

---

Toni Doolen, Dean, Oregon State University Honors College

I understand that my project will become part of the permanent collection of Oregon State University Honors College. My signature below authorizes release of my project to any reader upon request.

Miles Van de Wetering, Author

1 INTRODUCTION

Software Product Lines (SPLs) - programs consisting of a core set of functionality and one or many optional features - are increasingly important in today's world of complex, hierarchical software. They are also extremely difficult to reason about and debug. Often features are managed with pre-processor code like the #ifdef construct in C, which has for some time been noted as potentially harmful to code health [10]. Research into SPLs has suggested methods and architectures for better managing complexity and easier feature addition and management [1] [3] [9], but the fact remains that SPLs pose a challenge to even the best programmers. This research includes modularity and composition techniques which center around clear separation of concerns in the source code, but pre-processor code is still widely used in practice. Unfortunately, separation of concerns isn't always possible, convenient, or even useful.

To this end, I have designed a plugin for the Atom IDE which assists users in working with CPP annotations. Whether the programmer is working with legacy code (for example, the Linux kernel which contains a huge number of optional features all using CPP syntax) or designing a new software product line, my tool demonstrates a way of supporting programmers irrespective of methodology or architecture. Through the use of helpful color annotation designed to reduce clutter and noise generated by pre-processor annotations and a projectional viewing tool which allows the user to examine what code will be compiled under different scenarios, I aim to support programmers and make designing and debugging software product lines easier.

2 RELATED WORK

In previous research, a variety of methods have been proposed to create software product lines and ease the use of pre-processor code. Couto et al. set out to extract a software product line from ArgoUML, an

open source tool used to design systems in the Unified Modeling Language (UML), in order to evaluate different tools, techniques, and languages proposed to implement SPLs [2]. They noted a great deal of complexity in that project arose from ‘tangling’ and ‘nesting’ of features. While they were able to annotate the entire project using CPP code, they noted the tedium and complexity involved in doing so.

Another prominent project is the *Colored IDE* or CIDE [7]. Noting that fine-grained extensibility of code such as optional code statements or functions has the potential for reducing replicated code, potentially resulting in less obfuscation of code in SPLs, they set out to extend the Eclipse IDE to better support fine-grained variability. The essential idea of this software was to avoid pollution of the source code and allow programmers to more easily identify feature code. This was accomplished primarily through colorization - the programmer annotated their code, telling the editor that a line belonged to a specific feature. The line was then given an appropriate background color based on that feature. If code belonged to multiple nested features then it was marked with a background color that was a blend of the relevant feature colors. Lastly, a projectional view was implemented in CIDE which allowed the programmer to view what her code would look like if certain features were disabled or removed.

Marco Tulio Valente, Virgilio Borges, and Leonardo Passos designed a semi-automatic method of extracting software product lines from existing (unannotated) legacy code. Their tool accepted ‘seeds’ or small pieces of code that belonged to a certain feature, and attempted to propogate that feature annotation up syntactically if possible [12].

In our research’s group previous work, we tested a prototype of the tool presented in this paper which added colorful backgrounds to source code and asked users to perform a variety of tasks both with the colorization and without [8]. This prototype was based on the Choice Calculus [3] While the prototype was simple and non-interactive, users could switch between different ’projections’ of the code. This

allowed them to directly observe what code would be included in the product when different sets of features were enabled or disabled. We found that participants had a significantly easier time analyzing and understanding code which was annotated using our prototype tool than they did with the default CPP representation [8].

Walkingshaw and Ostermann also proposed a 'projectional editing' scheme for variational software [13]. In their approach, users would make edits to a partially configured software product line (similar to the approach that I take in my plugin) and those edits are reflected back automatically on the original source code according to defined rules. The core of this idea is the 'edit isolation' principle which mandates that any edits made in a particular configuration are isolated to the features which were configured when the edit was made [11]. Originally, my plug in was designed to employ this 'edit isolation' model, but I found very early on in the process that this principle can make editing somewhat tedious for users in practice. When edit isolation is enforced, users have no manual control over which features the code they are editing should belong to - any edits made are forced into the features that are currently configured. That approach is useful in several specific scenarios [13] but not necessarily useful in the general editing case. I instead opted for much more manual control to be given to the programmer - in this paper 'projection' will be used to indicate a partially configured view of the source code, but I pay no further attention to the edit isolation principle.

Worthy of special note is work that was done by Feigenspan et al. on background colors in IDEs. They implemented a program much like my own plugin which added background colors to features in an effort to help with program comprehension [4]. They did not implement any sort of code filtering or projectional viewing. Unfortunately, with their very limited sample size they were unable to draw statistically significant results demonstrating that their prototype tool was effective. They did, however,

note that subjects preferred color based annotations when given the choice, even though they were unable to conclude that programmers were more effective when provided with feature-based background coloring.

3 CONTRIBUTIONS

In our view, CIDE was an excellent step forward but had several limitations. The first and perhaps most important of these limitations was that CIDE only supported *optional features*. This precludes a case that is common in pre-processor syntax:

```
1    #ifdef MY_FEATURE
2    // do some stuff
3    #else
4    // do something else
5    #endif
```

While supporting only the basic `#ifdef ... #endif` is certainly easier, it also means that a great deal of existing legacy code is incompatible with their approach.

The second missing piece which we believe to be important is annotation of existing code. While CIDE provided convenient functionality for annotating existing code with features, it did not provide a way to import existing pre-processor annotated code into their tool. Third, color mixing seems intuitive as a means of representing and understanding nested features, but unfortunately results in background colors losing a great deal of meaning after just a few levels of nesting. Programmers are generally unable to juggle more than a few colors at a time, and the cross product of just 5 features might result in a

great many color combinations [5]. In CIDE they got around this problem by providing helpful tooltips clarifying which features a particular piece of code belonged to. Colors at this point simply served to point out that a piece of code was annotated with *some* features, but they were insufficient to discern which ones. We wanted a tool which programmers could tell, at first glance, what the relationships were between pieces of code and individual features. Lastly, CIDE limited feature annotation capability to valid syntax sub-trees. They made this decision on the grounds that annotations which are limited to syntax trees tend to be easier to support and understand. This may be true, but we are interested in supporting existing use cases and unfortunately situations like the following are quite common in industry code, but unsupported by syntax-tree limited software like CIDE:

```
1 #ifdef MY_FEATURE
2         if(condition) {
3 #else
4         if(different condition) {
5 #endif
```

Our group's previous research demonstrated that a prototype tool that allowed colorful background colors to differentiate features and a view selection tool to see what code would look like with certain features enabled or disabled assisted users in understanding and correctly interpreting code. Unfortunately, that study was unable to isolate the difference between the impacts that colorization may have had versus those that code folding and projectional viewing may have had. In thus study we seek to carefully control for these two different independent variables and determine which is more helpful to the user.

4 IMPLEMENTATION

Based on this prior research, I hypothesize that complicated preprocessor syntax like '#ifdef' can be made more understandable to the programmer through careful use of colorization combined with relatively familiar features like code folding. Ideally, these features would be integrated with an IDE in order to allow the programmer to leverage her usual workflow alongside the new tools at her disposal. Similar tools have been made in the past, but they have failed to implement several key features, and were written for IDEs that are used by only a subset of the industry. For the first several months of work on this project, my time was devoted to selecting and evaluating different IDEs that might be useful for our purposes. Unfortunately many of the IDEs that I experimented with suffered from the same problem - while they allowed adding new language highlighting rules, they generally did not allow the basic functionality of the editor to be modified. Two that I considered carefully were the JetBrains IDE and the MPS IDE (tree editing). JetBrains was eventually discarded because it did not allow me the versatility and control that I required in order to make extensive modifications. While the MPS IDE did allow for major changes to its functionality, we determined that it was too obscure to be of any great use for a user study. I wanted to design a plugin that would implement these key features in a modern, popular IDE. Furthermore, selecting a popular, simple IDE would help us evaluate the impacts of various features as accurately as possible. For this reason, I selected Atom.

Atom is a highly configurable development environment built on the Chrome web browser. This meant that all modifications would be written in JavaScript. I wrote a parsing program in Haskell using the Megaparsec library. Haskell was chosen as the language for the parser because other members of our lab are already very familiar with the language. This made it more maintainable for our group. The Haskell

parser reads in a C or C++ file and outputs an abstract syntax tree that defines all information required for the Atom plugin to replace concrete #ifdef syntax with colors. The AST nodes were defined as follows:

```
1   export interface ContentNode {
2           type: "text";
3           content: string;
4           span?: Span;
5           marker?: AtomCore.IDisplayBufferMarker;
6   }
7
8   type ChoiceKind = "positive" | "contrapositive";
9
10  export interface ChoiceNode {
11          type: "choice";
12          name: string;
13          thenbranch: RegionNode;
14          elsebranch: RegionNode;
15          kind: ChoiceKind;
16          span?: Span;
17          marker?: AtomCore.IDisplayBufferMarker;
18          delete?: boolean;
19  }
```

```
20
21        export interface RegionNode {
22                type: "region";
23                segments: SegmentNode[];
24                span?: Span;
25                hidden?: boolean;
26        }
```

The parser returns a Region node representing the top-level document to the JavaScript code whenever the plugin is toggled. Using this meta-syntax tree that the parser has constructed, the JavaScript hides the #ifdef syntax and replaces it with colors. The advantage of having a Haskell parser in addition to the main JavaScript plugin is that this allows us to import any valid CPP code and immediately use it with our plugin. It also means that the JavaScript code is syntax-agnostic: we can easily swap out any annotation construct in place of the usual '#ifdef' syntax. This was very helpful for us during development and it is a crucial feature for any plugin meant for use maintaining legacy code.

During the development process, complexity of the plugin grew significantly as features were added and edge cases discovered. Eventually this became almost unmanageable in JavaScript, so the code base was converted to Typescript (semantically equivalent to JavaScript, but allows for typed variables and a better object model. This made code maintenance and debugging much easier. The code seen above was in fact Typescript code taken from the final plugin source.

The plugin allows users to select which colors will represent each dimension of variation. Nested features are depicted by having the main, most specific feature color as the primary background color

of a piece of code, but all of the other, less specific features which the code is nested inside of are also represented as discreet bands of color along the left side of the editor, as shown in Figure 1.

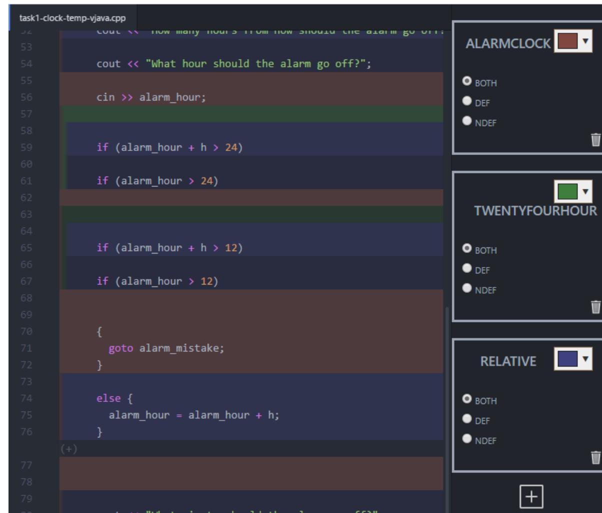


Figure 1: Nested Colorization in Task 1

It also allows them to 'filter' the code to see what will be compiled if different dimensions are defined or undefined. Code that is hidden with this feature may be viewed with a mouse-over tooltip. It allows the creation of new dimensions, and the insertion and deletion of arbitrarily nested choice nodes. Undo was never implemented in the plugin, unfortunately, because keeping track of a stack of tree states was not worth the usability gains for the purpose of our study. In the future, the ability to undo edits in the plugin would be extremely useful.

To complete the project, four variants of the plugin were made and published. The first is simply the default Atom IDE with undo disabled (undo was disabled in every version of the plugin). This was necessary to ensure that all treatment groups had access to the same basic features to avoid skewing results. The second included colorization and removal of '#ifdef' syntax, but did not allow variant folding. The third allowed variant folding but did not colorize the code, nor did it remove any of the '#ifdef' syntax. The fourth included all features.

5 RESEARCH QUESTIONS

The following research questions were central to the design of the view-based editing of variational code study. The ultimate goal is to identify those factors that assist users in understanding and modifying variational code – code containing *defined* and *undefined* variables that will be *included* or *excluded* from the program (respectively) . Towards this goal, we would like to better understand whether *filtering* or *colorization* of code provides the most benefit to users, or if the combination of the two is needed to assist users in understanding and modifying code of this type.

RQ1. Can users more *accurately* deduce the number of variants and describe the behavior of a particular variant represented in code presented in the plugin compared to code annotated with `CPP`?

RQ2. Can users more *quickly* deduce the number of variants and describe the behavior of a particular variant represented in code presented in the plugin compared to code annotated with `CPP`?

RQ3. Can users more *accurately* identify and correct errors within variants represented in code presented in the plugin compared to code annotated with `CPP`?

RQ4. Can users more *quickly* identify and correct errors within variants represented in code presented in the plugin compared to code annotated with `CPP`?

RQ5. Does filtering or colorization of variational code provide the contributing factor to performance differences when code is presented in the plugin?

6 METHODS

This section is divided into a profile of participants as described in section 6.1, the apparatus and materials used to conduct the study in section 6.2, and a detailed account of the procedure used in collection of data is found in section 6.3.

6.1 Participants

Study participants were recruited from undergraduate and graduate electrical engineering and computer science (EECS) students at Oregon State University. We employed the following eligibility criteria for selecting students:

1. Adults (18 years of age or older);
2. Graduate *or* undergraduate student in the EECS department;
3. Correctly answered four questions designed to test for a basic understanding of CPP code. These questions are shown in the Screening Test in Appendix A: Surveys.

We conducted our research using 29 subjects. Students were recruited from university mailing lists such as the *eecs-grads* and *eecs-undergrad* mailing lists for the EECS department. Twenty-five males and 4 females participated in the study. Two participants reported as 'Asian-American,' 5 as 'International,' 3 as 'Multi-Racial,' 9 as 'White/Caucasian,' 2 as 'Other,' and 12 declined to respond. Fourteen were between 18 and 24 years old, 7 were between 25 and 35 years old, and the rest did not report their age (all were at least over the age of 18, however). Of the undergraduate students that reported their grade level, there were 2 freshmen, 4 sophomores, 6 juniors, and 6 seniors. Eleven graduate students also participated.

The participants reported between a minimum of 1 year of C programming experience and a maximum of 15 years of experience, with an average of 3.5 years and a standard deviation of 3.5 years. Twenty two of the participants had taken and passed a data structures course, 7 had not. Eleven participants were familiar with Atom, 18 were not. None of the participants reported that they were color blind. Twenty-six of the participants had prior experience editing CPP code, while 3 did not.

As the goal of the study was to capture data for three treatments of the plugin (one with only filtering, one with only colorization, and one with both filtering and colorization) in addition to the control, each of

the participants was assigned to a particular treatment on a task-by-task basis. In this way we could vary the treatment across the tasks, and users were exposed to each of the four treatments in random order.

All subjects were given the same set of reading and understanding, and debugging tasks to complete using different treatments applied to the Atom IDE in the form of plugins. Participants were randomly assigned to treatments for each of the tasks.

6.2 Apparatus and Materials

Participants performed reading and understanding tasks as well as debugging tasks in a standard Microsoft Windows environment. The environment was installed on 7 Dell Optiplex desktop computers, and was implemented the same way on each to ensure consistency of the environment for all participants. In addition to the Windows environment, each participant had the following applications installed:

1. Firefox web browser;
2. Atom IDE (four plugins were installed in conjunction with this – see below);
    a. `variational-java` version 0.7.5 – this is the version of the plugin with both filtering and colorization;
    b. `VAtom-NoColor` version 1.0.6 – this is the version of the plugin with *only* filtering;
    c. `VAtomNoFeatures` version 0.1.0 – this is the *control*;
    d. `VAtomNoFold` version 0.1.0 – this is the version of the plugin with *only* colorization.

All applications were present and pre-configured to have the same representation for each participant. We assigned users a “subject ID” in order to maintain participant anonymity. These IDs were randomly generated combinations of alpha-numeric characters (each string 10 characters in length). Each participant took a 9-item “background” questionnaire (see Background Questionnaire in Appendix A: Surveys)

designed to characterize the participant's background and relative expertise in relation to using CPP. In addition, we asked users to self-identify color-blindness in order to further parse the affect of colorization on code understanding an modification. Participants were evaluated using a 6 to 8-item (this varied by task) Reading and Understanding questionnaire (see Appendix A: Surveys) to gauge the participant's ability to understand a particular variant of code based on which variables were included or excluded from execution.

Participants were then presented with a "bug report" from a user of the code involved with the task. They were asked to identify and correct the defect in the code, and then reflect on their experience by filling out a 5-item Post Task questionnaire (see Appendix A: Surveys). This post-task questionnaire was implemented utilizing the NASA-TLX questionnaire format [6]. The only modification we made to the formatting was to remove the "Physical Demand" input. Because of the nature of the tasks being performed, and the lack of physical exertion involved it was not relevant.

6.3 Procedure

Participants were assigned to specific seat locations where computers were already logged into Windows, with no other markers from previous study sessions remaining on the computer. The entirety of the procedure was replicated across all study sessions (which consisted of between three and five subjects each) in order to minimize any variance in the study session experience. In addition, a tutorial script was followed to ensure subjects all received the baseline information necessary for completing the study, as well as any information about the tools not covered in the minimum requirements for study participation.

Subjects were given a background questionnaire in order to establish some baseline information about experience using Atom as well as working with CPP in general. Tasks were provided to all subjects using

the same mechanism (study team members loaded the IDE environment and files prior to the start of each task), and all users had access to the Firefox browser (see Appendix B: Tasks) for a detailed view of the tasks all subjects received). Subjects were randomly assigned to treatments groups without prior knowledge of their background with Atom or CPP. All tasks were completed in their entirety before the next task was provided to them.

During the tasks, subjects were asked to answer questions designed to access their understanding of the code as well as its readability using the different versions of the plugin (as well as the default editor view). They were also asked to modify the code by performing a debugging task. Lastly during the tasks they were asked to reflect on the task load the encountered while completing the requirements of the task. Once all the tasks were completed, a post-study questionnaire was administered to evaluate the usefulness of the versions of the plugin.

6.3.1 Research Design. The choice of an exploratory study design should be valuable for identifying ways to improve our theories and models of how students approach and complete variational coding tasks. These theories and models will help us refine IDE plugins, and other tools for reading and editing variational code in order to reduce human effort required for variational coding tasks.

The study protocol is centered on giving subjects a particular set of tasks in a prescribed order, assigning the users a randomized treatment order for those tasks, and capturing reading and understanding information as well as the actions of the subjects as they complete a debugging task using the applications provided (Atom, Firefox, et al.). In a manner of speaking the *independent variable* is the task being completed, and the *dependent variables* are the different versions of the Atom plugin the users are using to complete those tasks. Therefore, we analyzed the level of understanding, time of completion, and successful completion of the tasks.

In addition to the independent and dependent variables described above, we identified our choice of platform (e.g. Microsoft Windows), choice of applications (e.g. Firefox and Atom IDE), the desktop environment consistency, and commonality of code feature numbers and expressions in the variational coding tasks as *control variables*. We also took into consideration potential *random variables* in the form of demographic information, and experience using the platform or applications beyond the minimum required for the study. The various dependent variables were mapped to a user comprehension metric, an observed task completion metric, and other relevant metrics.

7 DATA

We collected data in the form of questionnaires (section 7.1), instrumentation logs (section 7.2), as well as observational data sources (section 7.3).

7.1 Questionnaires

There were four questionnaires designed by the study team for use in this study. The first was a background questionnaire that was completed by the subjects before they had started any variational coding tasks. The aim of this first questionnaire was to explore the subject's background information about experience working with variational code and more general topics regarding demographic information. This was a 12-item questionnaire that highlighted expertise beyond the minimum requirement of the study.

The second questionnaire was a reading and understanding questionnaire that was completed by the subjects as part of the task completion process (e.g. after the code was given to them, but before the subjects began any debugging work). The aim of the second questionnaire was to ask some specific questions about the code the subjects would be working with in the debugging task that would follow.

The questionnaire had between 4 and 7 questions, depending upon the task, two of which were common questions across all of the tasks while the remainder were specific to the piece of code the subjects were looking at. This allowed us to gauge the abilities of subjects to understand different pieces of code using differing versions of the plugin.

For the third questionnaire we included a modified NASA-TLX [6] questionnaire designed to measure task load index. For the purposes of our study, we weren't interested in physical task load, so we used a subset of the original questionnaire as a basis for our post-task questionnaire. The aim of this questionnaire was to gauge responses to task load based on the task the subject just completed while the task was fresh in their mind.

The final questionnaire was a post-task questionnaire that was completed by the subjects as part of the exit process (e.g. after the subjects had completed all of the variational coding tasks). The aim of this questionnaire was to ask specific questions about the versions of the plugin the subjects just used to complete the tasks. We wanted to get an subjective measure of how users felt they did, and how well they felt the treatments affected their performance in each of the tasks.

7.2 Instrumentation Logs

Application-specific instrumentation was used to record log events. We collected Firefox window events (e.g. activate/deactivate, UI focus, tab activate/show/close, file downloads, print, and clipboard copy/cut/-paste).

For File I/O we used an off-the-shelf system logging tool to monitor file opens, closes, renames, etc, at the operating system level. We used a home-grown logging tool to record low-level keyboard, mouse and clipboard events that integrated with the application-specific logging tools that were already built.

All of the logs were merged using the shared ISO time field and some common fields like “app”, and “properties.” As different events had different relevant properties, some custom extraction code was needed for the properties field. The merged system logs gave a consistent time-ordered view of user behavior across all the applications at the disposal of the user.

In order to formulate the beginning and ending of tasks, we used a program designed for screen capture (TechSmith Camtasia Recorder 8). The frames within the screen capture gave us a straightforward way to observe the beginning and ending of the tasks, which allowed us to infer time to complete, as well as effort metrics from the events using the log event timestamps.

7.3 Observational Data

As it is not uncommon for instrumentation logs to miss key events, we also conducted screen recording screen recording of the subjects during the course of each session. As previously stated we used an off-the-shelf software package (TechSmith Camtasia Recorder 8) that ran in the background to avoid distracting users. It is worth noting that no recording of the actual subjects occurred.

In addition, we recorded observational data using journal entries for each session. These entries contained notes about the subjects on questions asked, struggles encountered, and barriers to completion of the tasks. These journal entries were coded for analysis.

8 RESULTS

None of the treatments demonstrated significant divergence from the control, except with respect to one metric which was ‘Frustration.’ Table 1 shows our results, along with p-values derived from an ANOVA analysis to determine if the treatment groups diverged from the control. Figures 2 through 8 show the

Figure 2: Reading & Understanding Times

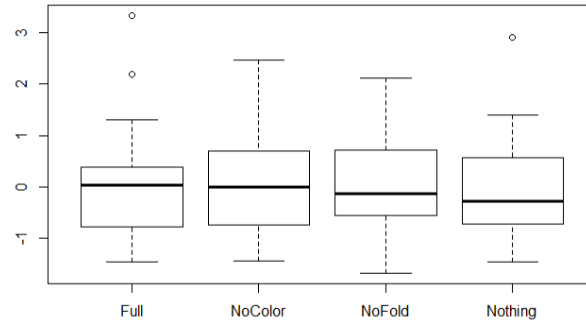
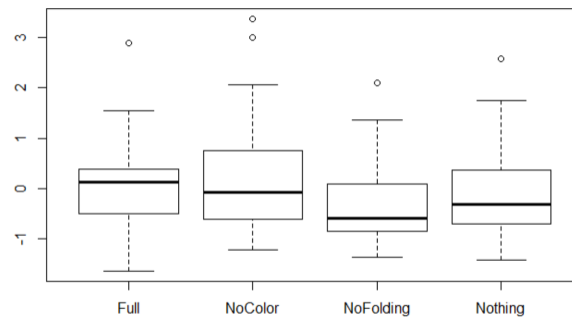


Figure 3: Debug Times



data as box-plots (the box-plots only compare the full treatment with the control for the NASA-TLX data). Fourteen participants reported that it was much easier to determine how many variants there were when using the plugin, 13 said that it was easier, and 2 said that it was the same difficulty either way. Fourteen said it was much easier to understand a particular variant using the full plugin, 13 said that it was easier, and 2 said that it was harder. Thirteen said that it was much easier to see how variants were related in the plugin, 1 said that it was easier, 1 said it was the same difficulty, 2 said that it was harder, and 1 said that it was much harder.

Table 1: Treatment Results

Measurement	Treatment	p-value
Reading and Understanding Time	NoFolding	.5854
	NoColor	.7119
	Full	.931
Debugging Time	NoFolding	.2971
	NoColor	.4327
	Full	.8284
Mental Difficulty	NoFolding	.5854
	NoColor	.7119
	Full	.1039
Temporal Difficulty	NoFolding	.4615
	NoColor	1
	Full	.857
Performance (Self-rated)	NoFolding	.3672
	NoColor	.2444
	Full	.3315
Effort	NoFolding	.2291
	NoColor	.4554
	Full	.2582
Frustration	NoFolding	.6777
	NoColor	.4652
	Full	.008 **

Figure 4: Mental Difficulty

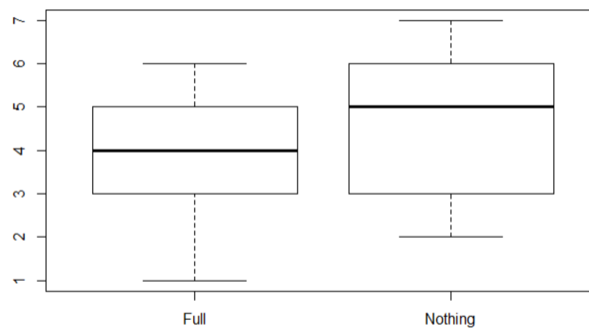


Figure 5: Temporal Difficulty

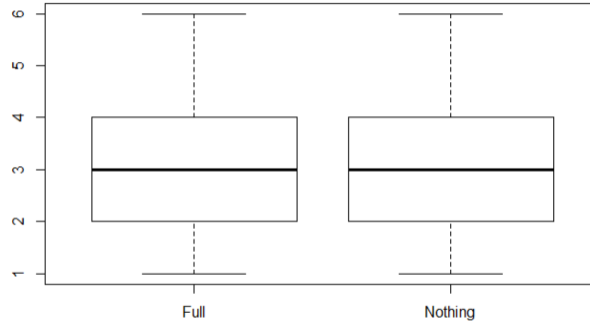


Figure 6: Performance

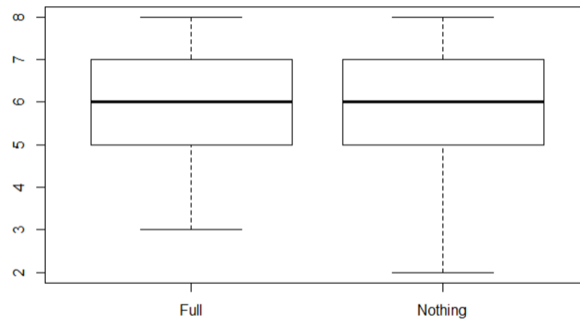


Figure 7: Effort

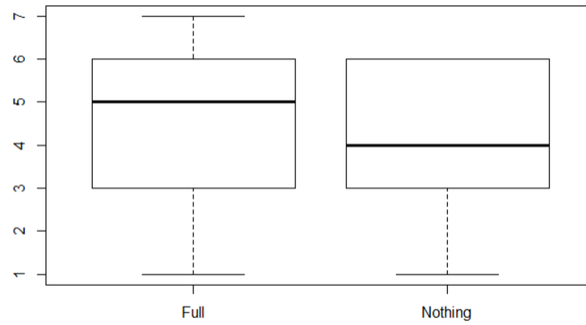
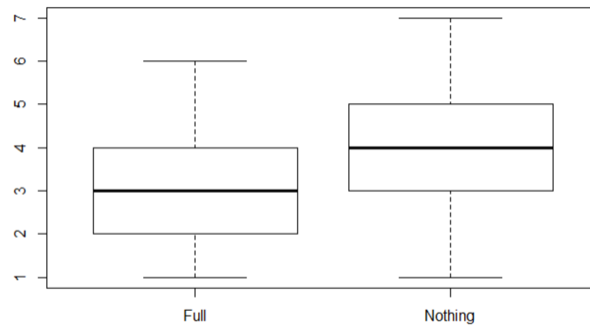


Figure 8: Frustration



9 DISCUSSION

In general, we were unable to glean any significant empirical data that colorization or folding are useful to programmers. There was little evidence that it either sped them up or improved their accuracy for the purposes of the tasks they were presented. This did not match our expectations, however, it does match the results of some other studies which have been conducted [4], with the exception of some of our group's previous work [13]. There are many possible reasons that these features might not be useful for our subjects. Firstly, since we attempted to study four unique treatments, subjects were only exposed to each new feature set for about twenty minutes. This may not be enough time to familiarize one's self with the tools. This theory is supported by some observational data. Users were sometimes seen to be performing tasks with the plugin without changing the colors (making it very difficult to discern which pieces of code belonged to which features) or making use of the projectional viewing functionality.

It is also possible the features we are providing users with are not significantly more useful than the basic representation because they do not actually add any new information. Rather, they reformat information that was already there to (hopefully) make things nicer for the user. This might explain why correctness values do not change with treatment type - correct responses from users are much more highly correlated with user experience than with treatment type.

User's responses to our subjective questions were extremely positive. Almost all the subjects reported that they found the tasks to be easier in a variety of ways when they were using the plugin, and almost all reported that they preferred the plugin to the default view. The user's perception of their experience did not match our empirical data - although it is important to note that they did no worse than with the default view, despite using new and very unfamiliar tooling.

With all this we are forced to conclude without significant evidence that users are more accurate or faster when working with our plugin. Rather, what we do have is a very solid consensus among our test subjects that their *experience* is better when given access to colorization and folding features. The user's experience is a valuable thing, and we believe that it is worth further consideration to see if these user-pleasing features can be enhanced to help user efficiency and accuracy in addition to their experience.

9.1 Threats to Validity

A concise listing of potential threats to validity for an empirical study is provided by Wohlin at al. [14], and we addressed those threats we were able to identify as pertinent to this study.

9.1.1 Internal Validity.

1. *History*: The study is exposed to this potential threat, since we have no idea whether or not subjects are performing tasks while "fresh" or after a long stressful day. We scheduled subjects into sessions at the same time of day (starting at 15:00) in order to minimize the effects of subjects having variation in their level of energy.
2. *Maturation*: Our study was scheduled to take as much as two hours to complete. Because subjects react differently as time passes, we feel there is some potential exposure to this threat. We mitigated the effect of this by maintaining a consistent stream of tasks so subjects weren't constantly waiting on the study.
3. *Selection*: Because humans naturally vary in performance, this is a potential threat. We minimized this threat through the use of our background questionnaire, as well as through the random assignment of treatments to tasks with a consistent presentation of task order. We also plan

to analyze data that compares how well each subject did compared with themselves, and then compare these relative numbers between subjects, in addition to a standard comparison of raw numbers. We believe that by adjusting this way for individual skill we will be able to strengthen the validity of our conclusions.

4. *Ambiguity*: It could be that some factor causes differences in performance between subjects, rather than the version of the plugin being used. The use of randomization of treatment assignment could potentially amplify this affect. We mitigated this by designing as unambiguous of tasks as we could, though we still believe there is some necessary bias introduced through the use of a single IDE as the delivery tool for tasks.

9.1.2 *External Validity.*

1. *Interaction of setting and treatment*: In this study, we selected participants from a specific population (students in an academic environment) in order to study the differences presented by the tools. Because we assigned people randomly into treatments on a per task basis, with enough participants we obtain a natural distribution of plugin versions per task.

9.1.3 *Construct Validity.*

1. *Interaction of different treatments*: Because users were randomly assigned to plugin versions on a per task basis, it becomes difficult to disentangle the effects of the plugin from the effects of user expertise. In order to minimize this we take into account a large number of users in order to obtain a natural distribution of plugin versions per task.

9.1.4 *Conclusion Validity.*

1. *Reliability of measures*: Reliance on instrumentation event logging to obtain a good portion of our results puts us at risk of exposure to this threat. To decrease our exposure we not only collected logging data, but also had an observer record any anomalies that may have occurred, and we also employed screen capture as a backup to review sessions that had missing or incorrect logging information.

In addition to the threats above, there are two other aspects that are worth considering as threats to validity in our study:

1. Human work behavior for “fake” tasks is not always the same as that for “real” or “important” tasks.
2. Completion of “fake” tasks is often less than “real” or “important” tasks.

REFERENCES

[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. Software Product Lines. In *Feature-Oriented Software Product Lines*. Springer Berlin Heidelberg, Berlin, Heidelberg, 3–15. DOI: https://doi.org/10.1007/978-3-642-37521-7_1

[2] M V Couto, M T Valente, and E Figueiredo. 2011. Extracting software product lines: A case study using conditional compilation. *2011 15th European Conference on Software Maintenance and Reengineering, CSMR 2011* (2011), 191–200. DOI: https://doi.org/10.1109/CSMR.2011.25

[3] Martin Erwig and Eric Walkingshaw. 2013. The Choice Calculus: A Representation for Software Variation. (2013). http://web.engr.oregonstate.edu/

[4] J. Feigenspan, M. Schulze, M. Papendieck, C. Kastner, R. Dachselt, V. Koppen, and M. Frisch. 2011. Using background colors to support program comprehension in software product lines. In *15th Annual Conference on Evaluation & Assessment in Software Engineering (EASE 2011)*. IET, 66–75. DOI: https://doi.org/10.1049/ic.2011.0008

[5] Mark Harrower and Cynthia A Brewer. ColorBrewer.org: An Online Tool for Selecting Colour Schemes for Maps. (????). DOI: https://doi.org/10.1179/000870403235002042

[6] Sandra G Hart and Lowell E Staveland. 1988. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. *Advances in psychology* 52 (1988), 139–183.

[7] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2008. Granularity in software product lines. *Proceedings of 30th International conference on Software engineering* (2008), 311–320. DOI: https://doi.org/10.1145/1368088.1368131

[8] Duc Le, Eric Walkingshaw, and Martin Erwig. 2011. #ifdef confirmed harmful: Promoting understandable software variation. *Proceedings - 2011 IEEE Symposium on Visual Languages and Human Centric Computing, VL/HCC 2011* (2011), 143–150. DOI: https://doi.org/10.1109/VLHCC.2011.6070391

[9] L. Neves, P. Borba, V. Alves, L. Turnes, L. Teixeira, D. Sena, and U. Kulesza. 2015. Safe evolution templates for software product lines. *Journal of Systems and Software* 106 (2015), 42–58. DOI: https://doi.org/10.1016/j.jss.2015.04.024

[10] Henry Spencer and Geoff Collyer. 1992. #ifdef Considered Harmful, or Portability Experience With C News. In *USENIX Conf.* https://www.usenix.org/legacy/publications/library/proceedings/sa92/spencer.pdf

[11] Stefan Stanciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wasowski. 2016. Concepts, Operations, and Feasibility of a Projection-Based Variation Control System. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 323–333. DOI: https://doi.org/10.1109/ICSME.2016.88

[12] M T Valente, V Borges, and L Passos. 2012. A semi-automatic approach for extracting software product lines. *IEEE Transactions on Software Engineering* 38, 4 (2012), 737–754. DOI: https://doi.org/10.1109/TSE.2011.57

[13] Eric Walkingshaw and Klaus Ostermann. 2014. Projectional editing of variational software. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 29–38.

[14] C Wohlin, P Runeson, M Host, MC Ohlsson, B Regnell, and A Wesslen. 2000. Experimentation in software engineering: an introduction. 2000. (2000).

A SURVEYS

Screening Test: View-based Editing of Variational Code

EXPLANATION OF RESEARCH

Project Title: View-based Editing of Variational Code
Principal Investigator: Eric Walkingshaw
Student Researcher(s): Keeley Abbott, Miles Van de Wetering, Rikki Gibson

This form contains information you will need in order to help you decide whether to take part in this screening test or not. Please read the form carefully and ask study team member(s) questions about anything that is not clear.

STUDY PURPOSE AND PROCEDURE

This is a research study, the results of which will be published at scientific conferences. The purpose of this research study is to understand the kinds of variation currently managed with conditional compilation by languages like the C Preprocessor language (CPP). We will be focusing primarily on comparing users' performance in reading and understanding, identifying and correcting errors in code (debugging), and adding new features to code in CPP and in a different environment, called "the plugin."

To be eligible for this screening, you need to be: (1) an adult (18+ years old) in a major in the Electrical Engineering and Computer Science department.

SCREENING TEST

To participate in the actual study, you are required to take part in a screening test, which is supposed to take less than 10 minutes. During this test, we will collect your name and email address as well as your test results. If you pass the test, we will contact you to schedule for the actual research study session, for which you will receive a compensation of $30 if you complete the session in full. In addition, your information and test results will be stored in a secure database that can only be accessed by members of our study group. If you do not pass the screening test, we will destroy your data.

RISKS

There are no foreseeable risks to you for participating in this screening test.

BENEFITS

This study is not designed to benefit you directly. Your participation will help us design tools that will produce better representations of variational code.

CONFIDENTIALITY

The information you provide during this screening test will be kept confidential to the extent permitted by law. To ensure confidentiality if you participate in the actual study you will be assigned a unique identification code and your name and email address will be deleted from our records (keeping only the identification codes). All research information will be stored securely. As the research data itself contains no personal information, even if there was a breach of confidentiality the risk would be minimal to you. Study data will be retained for a period of four years, after which all study data will be deleted.

VOLUNTARY PARTICIPATION

Taking part in this screening test is voluntary. You may choose to not take part at all. If you agree to take this test, you may stop at any time. If you decide not to take part, or if you stop taking at any time, your decision will not result in any penalty or loss of benefits to which you may otherwise be entitled. If you withdraw from the test, your data will be

destroyed. Declining to participate will not impact your grade, class standing or relationship with any instructor.

CONTACT INFORMATION

If you have any questions about this research project, please contact: Professor Eric Walkingshaw, eric.walkingshaw@oregonstate.edu (preferred), (541) 737-3342. If you have questions about your rights or welfare as a participant, please contact the Oregon State University Human Research Protection Program (HRPP) office, at (541) 737-8008 or by email at IRB@oregonstate.edu.

☐ I have read and understand the explanation of this study

Please indicate whether you are eligible to take this screening test or not:

Adult (18+ years old).

○ Yes

○ No

Please provide your name:

Please provide your email address:

In the C programming language, `#ifdef` annotations can be used to statically determine which code is included in a program before it is compiled into an executable. The C Preprocessor (CPP) is the tool that translates C code containing `#ifdef` annotations into plain C code that can be compiled.

To illustrate, consider the following C program containing `#ifdef` annotations. The macro `M` referred to by the `#ifdef` annotation can either be *defined* or *undefined*, leading to different plain C programs.

```
int main() {
#ifdef M
  int x = 2;
#else
  int x = 3;
#endif
  return x;
}
```

If we run CPP with macro `M` = *defined*, then we get the following plain C program.

```
int main() {
  int x = 2;
  return x;
}
```

If we run CPP with macro `M` = *undefined*, then we get the following plain C program.

```
int main() {
  int x = 3;
  return x;
}
```

In this screening test, consider the following C program with `#ifdef` annotations referring to two different macros `A` and `B`.

```
int main() {

#ifdef A
  int foo = 1;
#else
  int bar = 2;
#endif
  int qux = 3;

  if (qux == 4) {
#ifdef B
    return foo;
#else
    return bar;
#endif
  }
  return 0;
}
```

For each configuration of the `A` and `B` macros below, indicate whether or not the program produced by running CPP will compile successfully or not.

	compiles successfully	compilation error
A = defined; B = defined	○	○
A = defined; B = undefined	○	○
A = undefined; B = defined	○	○
A = undefined; B = undefined	○	○

Background Questionnaire: View-based Editing of Variational Code

EXPLANATION OF RESEARCH

Project Title: View-based Editing of Variational Code
Principle Investigator: Eric Walkingshaw
Student Researcher(s): Keeley Abbott, Miles Van de Wetering, Rikki Gibson

----

This form contains information you will need in order to help you decide whether to take part in this screening test or not. Please read the form carefully and ask study team member(s) questions about anything that is not clear.

STUDY PURPOSE AND PROCEDURE

You are being asked to take part in a research study, the results of which will be published at scientific conferences. The purpose of this research study is to understand the kinds of variation currently managed with conditional compilation by languages like the C Preprocessor language (CPP). We will be focusing primarily on comparing users' performance in reading and understanding, identifying and correcting (debugging), and adding new features to code in CPP and in a different environment, called "the plugin."

You are being invited to take part in this study because you have satisfied the following requirements:

(1) You are an adult (18+ years old).
(2) You passed the basic test for reading and understanding C programs containing CPP annotations.

ACTIVITIES

If you agree to participate, your involvement will consist of one session that will last up to two (2) hours. At the beginning of the session, you will be asked to fill out a background questionnaire. After filling out the questionnaire, you will be led through a tutorial to familiarize you with the basics of how to use the CPP code-viewing environment, and the plugin code-viewing environment. Once completed, the main experiment will begin.

The main experiment will consist of four tasks, one of the tasks will be completed in CPP, and the three remaining tasks will be completed using the plugin. Each task consists of reading and understanding variational code, identifying and correcting errors (debugging) variational code, and adding a new feature to variational code. Your answer and response time will be used to measure how well you were able to understand the variational code using the two tools.

Between each task, you will be asked a series of questions that will be used to gauge the index load of each task in the two tools as well as your confidence level in your completion of the task. After the four main tasks are finished, a final questionnaire will be used to gauge your reactions to the tasks and the environments.

The data collected (questionnaires and log files) will be made available only to members of the study team conducting the experiment.

RISKS

This study has no foreseeable risks to you as a participant other than the minimum possibility of a Breach of Confidentiality risk, which means that your participation in the study may be exposed. This risk is minimum, because all the study data is securely stored and is associated with an encoded participant ID rather than your actual identifying information.

BENEFITS

This study is not designed to benefit you directly. We intend to use the information gathered to improve the design of future software development environments, which may be an indirect benefit.

PAYMENT

You will receive compensation in the amount of $30 if you complete the study session in full. If you decide to withdraw from the study session during the session, you will receive no compensation.

CONFIDENTIALITY

Records of participants in this research project will be kept confidential to the extent permitted by law. However, federal government regulatory agencies and the Oregon State University Institutional Review Board (a committee that reviews and approves research studies involving human subjects) may inspect a copy of records pertaining to this research.

Your identity will never be used when results of the study are published, and your data will only be associated with a participant number. Your name, as well as any other information that could identify you, will not be used or disclosed to anyone.

VOLUNTARY PARTICIPATION

Participation in this study is voluntary. You may choose not to take part at all. If you agree to participate in this study, you may stop participating at any time. If you decide not to take part, or it you stop participating at any time, your decision will not result in any penalty or loss of benefits to which you may otherwise be entitled. If you withdraw from the study, your data pertaining to the study will be destroyed. Declining to participate will not impact your grade, class standing or relationship with any instructor.

STUDY CONTACTS

If you have any questions about this research project, please contact: Professor Eric Walkingshaw, eric.walkingshaw@oregonstate.edu (preferred), (541) 737-3342. If you have questions about your rights or welfare as a participant, please contact the Oregon State University Human Research Protection Program (HRPP) office, at (541) 737-8008 or by email at IRB@oregonstate.edu.

SIGNATURE

By checking the box below, you indicate that this study has been explained to you, that your questions have been answered to your satisfaction, and that you agree to participate in this study.

- [ ] I have read and understand the explanation of this study

Please indicate whether or not you are eligible to be part of this study:

(1) Adult (18+ years old).

- Yes
- No

Subject ID: ${e://Field/subjectID}

Gender:

- Male
- Female
- Non-binary/third gender
- Prefer to self-describe
- Prefer not to say

Racial/Ethnic Group:

- African American/Black
- Asian American
- Hispanic/Latino
- International
- Multi-Racial/Multi-Ethnic
- Native American/Alaskan
- Pacific Islander
- White/Caucasian
- Decline to respond
- Other

Age:

- 18 -- 24 years old
- 25 -- 34 years old
- 35 -- 44 years old
- 45 -- 54 years old
- 55 -- 64 years old
- 64 -- 74 years old
- 75 years or older

Current year in school:

- Freshman
- Sophomore
- Junior
- Senior
- Graduate Student
- Other

Years of C programming experience:

Have you taken and passed CS 261 (with a grade of "B" or higher)?

- Yes
- No

Have you used the Atom IDE to edit or view code before?

- Yes
- No

Are you color blind?

- Yes
- No

Do you have previous experience editing or viewing CPP code (#ifdef statements)?

- Yes
- No

Reading and Understanding Questionnaire: View-based Editing of Variational Code

Subject ID: ${e://Field/subjectID}

Task: ${e://Field/task}

How many dimensions of variation are there?

How many *unique* programs can be compiled from this code?

TASK1

The user entered '15', '10', '0', '1', '7', with

`ALARM=defined, RELATIVE=defined, TWENTYFOURHOUR=defined`

will the alarm sound? If so, how long will it be until the alarm goes off?

The user entered '15', '10', '0', '12', '10','0', '0', with

`ALARM=defined, RELATIVE=undefined, TWENTYFOURHOUR=undefined`

Will the alarm sound? If so, how long will it be until the alarm goes off?

TASK2

What are the values of *reg* and *ctr* after main has executed in the following scenarios?

`HASSKIP=defined, HASGOTO=undefined`

reg

ctr

HASSKIP=undefined, HASGOTO=undefined

reg

ctr

TASK3

What does the main function print in the following scenarios?

STRDATA=defined, POP=defined

STRDATA=defined, POP=undefined

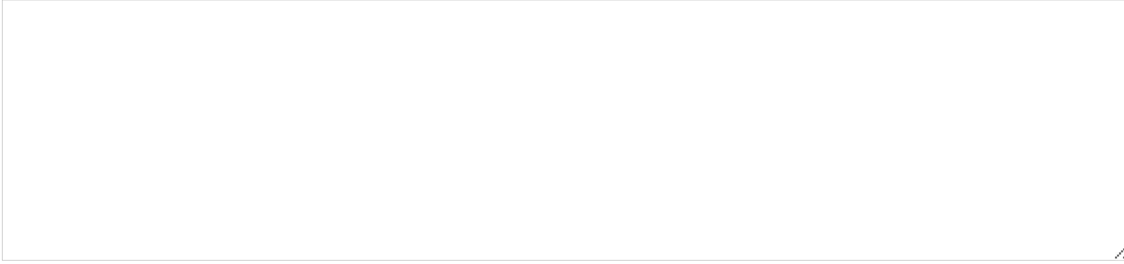
STRDATA=undefined, POP=undefined

TASK4

What should be printed on the first day (the first iteration of the while loop in main) for each of the following scenarios? What will crime be set to?

JOKER=defined, POLITE=defined, BUTLER=defined

JOKER=defined, POLITE=defined, BUTLER=undefined



Post-Task Questionnaire: View-based Editing of Variational Code

Subject ID: ${e://Field/subjectID}

Task: ${e://Field/task}

Mental Demand

	Very low							Very high
	0	1	2	3	4	5	6	7
How mentally demanding was the task?								

Temporal Demand

	Not hurried							Very hurried
	0	1	2	3	4	5	6	7
How hurried or rushed was the pace of this task?								

Performance

	Failure							Perfect
	0	1	2	3	4	5	6	7

How successful were you in accomplishing what you were asked to do?							

Effort

	Not hard 0	1	2	3	4	5	6 Very hard 7
How hard did you have to work to accomplish your level of performance?							

Frustration

	Not frustrated 0	1	2	3	4	5	6 Very frustrated 7
How insecure, discouraged, irritated, stressed, and annoyed were you?							

Post-Study Questionnaire: View-based Editing of Variational Code

Subject ID: ${e://Field/subjectID}

With which representation was it easier to determine how many variants there were?

Much easier with #ifdef	Easier with #ifdef	Same difficulty with both	Easier with plugin	Much easier with plugin
○	○	○	○	○

In which system was it easier to understand a particular variant?

Much easier with #ifdef	Easier with #ifdef	Same difficulty with both	Easier with plugin	Much easier with plugin
○	○	○	○	○

In which system was it easier to see how the different variants were related to each other?

Much easier with #ifdef	Easier with #ifdef	Same difficulty with both	Easier with plugin	Much easier with plugin
○	○	○	○	○

Overall, which tool do you think makes it easier to understand software that contains variation?

#ifdef	with color	with folding	with color and folding
○	○	○	○

Using the scale below, please evaluate the following statement:

"In general, the tasks I completed in this study were difficult."

Strongly agree	Agree	Do not agree or disagree	Disagree	Strongly disagree
○	○	○	○	○

Using the scale below, please evaluate the following statement:

"When looking at the `#ifdef` annotated code, I could tell which macros were meant to be mutually exclusive (only one can be selected at a time)."

Strongly agree	Agree	Do not agree or disagree	Disagree	Strongly disagree
○	○	○	○	○

Using the scale below, please evaluate the following statement:

"I think I could translate the `#ifdef` annotated code into an equivalent representation in the plugin."

Strongly agree	Agree	Do not agree or disagree	Disagree	Strongly disagree
○	○	○	○	○

Using the scale below, please evaluate the following statement:

"In general, I found the `#ifdef` annotated code easy to understand."

Strongly agree	Agree	Do not agree or disagree	Disagree	Strongly disagree
○	○	○	○	○

Using the scale below, please evaluate the following statement:

"I think I could translate the code in the plugin into an equivalent in `#ifdef` annotated code."

Strongly agree	Agree	Do not agree or disagree	Disagree	Strongly disagree
○	○	○	○	○

Using the scale below, please evaluate the following statement:

"In general, I found code in the plugin easy to understand."

Strongly agree	Agree	Do not agree or disagree	Disagree	Strongly disagree
○	○	○	○	○

B TASKS

B.1 Task 1: Clock

```
1  #include <iostream>
2  #include <windows.h>
3
4  using namespace std;
5
6  #ifdef ALARMCLOCK
7  bool alarm_set = false;
8  int alarm_hour = -1;
9  int alarm_minute = -1;
10 int alarm_second = -1;
11 #endif
12
13
14 int main()
15 {
16     int a = 0;
17     int h;
18     int m;
```

```
19     int s;
20 #ifdef ALARMCLOCK
21     int res;
22 #endif
23
24     mistake:
25
26     cout << "What␣is␣the␣current␣hour?\n";
27     cin >> h;
28
29     cout << "\nWhat␣is␣the␣current␣minute?\n";
30     cin >> m;
31
32     cout << "\nWhat␣is␣the␣current␣second?\n";
33     cin >> s;
34
35 #ifdef TWENTYFOURHOUR
36     if (h > 24 || m > 60 || s > 60)
37 #else
38     if (h > 12 || m > 60 || s > 60)
39 #endif
```

```
40     {
41           cout << "\nPlease␣try␣again!\n\n";
42           goto mistake;
43     }
44 #ifdef ALARMCLOCK
45     cout << "Would␣you␣like␣to␣set␣an␣alarm?␣(0-no/1-yes)";
46     cin >> res;
47
48     alarm_mistake:
49     if (res == 1) {
50
51 #ifdef RELATIVE
52       cout << "How␣many␣hours␣from␣now␣should␣the␣alarm␣go␣off?";
53 #else
54       cout << "What␣hour␣should␣the␣alarm␣go␣off?";
55 #endif
56       cin >> alarm_hour;
57 #ifdef TWENTYFOURHOUR
58 #ifdef RELATIVE
59       if (alarm_hour + h > 24)
60 #else
```

```
61       if (alarm_hour > 24)
62 #endif
63 #else
64 #ifdef RELATIVE
65       if (alarm_hour + h > 12)
66 #else
67       if (alarm_hour > 12)
68 #endif
69 #endif
70       {
71         goto alarm_mistake;
72       }
73 #ifdef RELATIVE
74       else {
75         alarm_hour = alarm_hour + h;
76       }
77 #endif
78
79 #ifndef RELATIVE
80       cout << "What␣minute␣should␣the␣alarm␣go␣off?";
81       cin >> alarm_minute;
```

```
82
83          if (alarm_minute > 60)
84          {
85            goto alarm_mistake;
86          }
87          cout << "What␣second␣should␣the␣alarm␣go␣off?";
88          cin >> alarm_second;
89          if (alarm_second > 60)
90          {
91            goto alarm_mistake;
92          }
93    #else
94          alarm_minute = m;
95          alarm_second = s;
96    #endif
97
98        }
99    #endif
100
101
102       while (a == 0)
```

```
103     {
104             Sleep(1000);
105
106 #ifdef ALARMCLOCK
107             if (h == alarm_hour && m == alarm_minute && s ==
                   alarm_second) {
108               cout << "BEEP!␣BEEP!␣BEEP!";
109             }
110 #endif
111
112             //increment time
113             s++;
114             if (s > 59)
115             {
116                     s = 0;
117                     m++;
118
119                     if (m > 59)
120                     {
121                             m = 0;
122                             h++;
```

```
123
124 #ifdef TWENTYFOURHOUR
125                       if (h > 24)
126                       {
127 #else
128                       if (h > 12)
129 #endif
130                       {
131                             h = 0;
132                       }
133                 }
134           }
135     }
136 }
```

B.2 Task 2: Goto

```
1 #include "stdio.h"
2
3 typedef enum {
4 #ifdef HASGOTO
5     GOTO,
```

```
 6 #endif
 7 #ifdef HASSKIP
 8     SKIP,
 9 #endif
10     ADD, END
11 } t_code;
12
13 typedef struct {
14     t_code code; int arg;
15 } t_instr;
16
17 int reg = 0;
18 int ctr = 0;
19
20 void exec(t_instr i) {
21     if (i.code == ADD) {
22         reg += i.arg;
23 #ifdef HASSKIP
24     } else if (i.code == SKIP) {
25         if (reg > i.arg) ctr++;
26 #endif
```

```
27 #ifdef HASGOTO
28     } else if (i.code == GOTO) {
29         ctr = i.arg;
30 #endif
31     }
32
33
34 #ifdef HASGOTO
35     if (i.code != GOTO)
36 #endif
37         ctr++;
38 }
39
40 void eval(t_instr *instrs) {
41     t_instr i;
42     while (1) {
43         i = instrs[ctr];
44         if (i.code == END) return;
45         exec(i);
46     }
47 }
```

```
48
49 int main() {
50     t_instr is[] = {
51         {ADD,1},
52 #ifdef HASSKIP
53         {SKIP,3},
54 #endif
55 #ifdef HASGOTO
56         {GOTO,0},
57 #endif
58         {ADD,1}, {END,0}};
59     printf("result=%d\n", eval(is));
60 }
```

B.3 Task 3: StackLang

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4
5 typedef struct Stack Stack;
6 struct Stack {
```

```
7  #ifdef STRDATA
8    void* value;
9  #else
10   int value;
11 #endif
12   Stack* next;
13
14 };
15
16 Stack* pushStack(Stack* s, int val) {
17   Stack* newStack = (Stack*) malloc(sizeof(Stack));
18   newStack->value = val;
19   newStack->next = s;
20   return newStack;
21 }
22 #ifdef POP
23 Stack* popStack(Stack* s) {
24   Stack* res = s->next;
25   free(s);
26   return res;
27 }
```

```
28 #else
29
30 #endif
31
32 void printStack(Stack* s) {
33   while (s != NULL) {
34 #ifndef STRDATA
35     printf("%d␣", s->value);
36 #else
37     printf("%s␣", (char*) s->value));
38 #endif
39     s = s->next;
40   }
41   printf("\n");
42 }
43
44 int main() {
45   Stack* s = NULL;
46 #ifdef STRDATA
47   s = pushStack(s, "I");
48   s = pushStack(s, "AM");
```

```
49   s = pushStack(s, "REALLY");
50 #ifdef POP
51   s = popStack(s);
52 #endif
53   s = pushStack(s, "ENJOYING");
54   printStack(s);
55 #ifdef POP
56   s = popStack(s);
57 #endif
58   s = pushStack(s, "A");
59   s = pushStack(s, "BEAVER");
60   s = pushStack(s, "BELIEVER");
61 #ifdef POP
62   s = popStack(s);
63   s = popStack(s);
64   s = popStack(s);
65 #endif
66 #else
67   s = pushStack(s, 2);
68   s = pushStack(s, 3);
69 #ifdef POP
```

```
70   s = popStack(s);
71 #endif
72   s = pushStack(s, 5);
73   printStack(s);
74 #ifdef POP
75   s = popStack(s);
76 #endif
77   s = pushStack(s, 6);
78 #ifdef POP
79   s = popStack(s);
80   s = popStack(s);
81   s = popStack(s);
82 #endif
83 #endif
84   printStack(s);
85 }
```

B.4 Task 4: Jeeves

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
```

```
4
5  int health = 10;
6
7  #ifdef JOKER
8  int crime = 0;
9  #endif
10
11 void say(char* message) {
12 #ifdef POLITE
13   printf("%s", "Sir,␣\n");
14 #endif
15   printf("%s", message)
16 }
17
18 void morning_greeting() {
19   say("Good␣morning!\n");
20 }
21
22 void lunch_time() {
23   say("It␣is␣time␣for␣lunch\n");
24 }
```

```
25
26 void bed_time() {
27   printf("%s", "It's␣getting␣late...\n");
28   if(crime > 0) {
29     say("To␣the␣Bat␣Cave!\n");
30 #ifdef JOKER
31       crime--;
32       health--;
33 #else
34       crime--;
35 #endif
36   } else {
37     say("Go␣to␣bed");
38     health++;
39   }
40 }
41
42 int main() {
43
44   while(health > 0) {
45 #ifdef BUTLER
```

```
46      morning_greeting();
47      lunch_time();
48      bed_time();
49  #else
50      say("Batman␣forgets␣to␣take␣care␣of␣himself.");
51      health--;
52  #endif
53
54  #ifdef JOKER
55      crime += 3;
56  #else
57      crime += 1;
58  #endif
59    }
60  }
```