

EUKLAS

Supporting Copy-and-Paste Strategies for Integrating Example Code

Christian Dörner

Senacor Technologies AG

Erika-Mann-Str. 55

80636 München

christian.doerner@senacor.com

Andrew R. Faulring

HCI Institute

Carnegie Mellon University

5000 Forbes Avenue, Pittsburgh, PA

faulring@cs.cmu.edu

Brad A. Myers

HCI Institute

Carnegie Mellon University

5000 Forbes Avenue, Pittsburgh, PA

bam@cs.cmu.edu

Abstract

Researchers have paid increasing attention in recent years to the fact that much development occurs through example modification. Helping programmers with some of the pitfalls and vagaries of working with example code is the goal of our tool, called Euklas. It helps developers to integrate JavaScript example code into their own projects by using familiar IDE interaction techniques of the Eclipse IDE. The Euklas plugin uses static, heuristic source code checks to highlight potential errors and to recommend potential fixes, when incomplete sections of code are copied from a working JavaScript example and pasted into the program being edited. The most unique feature of the tool is the ability to automatically import missing variable and function definitions from an example file into a new project file. Our preliminary user study of Euklas suggests that it supports users in fixing errors more easily.

Categories and Subject Descriptors D.2.6 [Programming Environments]: Programmer workbench.

General Terms Algorithms, Design, Human Factors

Keywords Code Reuse, Copy-and-Paste, JavaScript, Eclipse, Natural Programming, Examples

1. Introduction

Leveraging examples is an established technique in design [1], and has recently received increasing attention from

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org. PLATEAU'14, October 21 2014, Portland, OR, USA Copyright 2014 ACM 978-1-4503-2277-5/14/10...\$15.00 http://dx.doi.org/10.1145/2688204.2688208

researchers focusing on programming tools [2-5]. With the rise of search engines and web repositories of code along with discussion threads, blogs, and code example websites, people often create new systems by copying and pasting code snippets from such sources [6]. Surveys and research show that looking for examples is often people's preferred way to learn how to perform a task or to learn how to use application programming interfaces (APIs) [7].

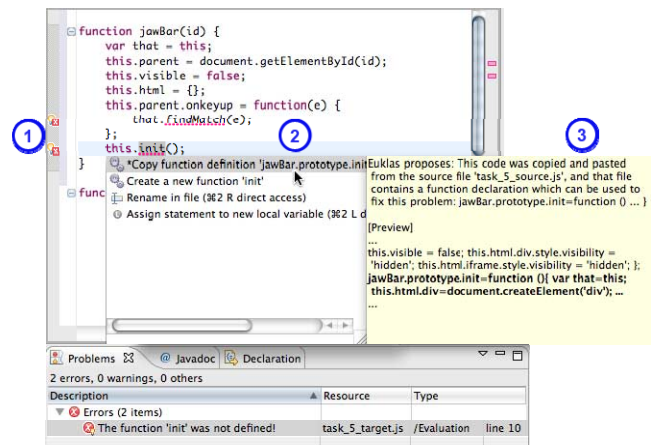



Figure 1. Euklas supports peoples' integration attempts as it enhances Eclipse's JavaScript editor by (#1) highlighting errors in the source code, (#2) providing quick fixes including the use of the context in the origin of any copied code, and (#3) including explanations for copy-and-paste errors based on the code of the example snippet that was used.

Most of the research on reusing examples has focused on building improved search and data mining tools to help with finding the examples (e.g., [1, 3, 5, 7, 8]). However, there has not been much research on assisting users in reusing and integrating the examples after they have been

found, with the exception of [4, 9, 10]. Copy-and-paste has been identified as a common usage pattern for reusing code (e.g., copying several lines, blocks, or even whole methods) [11-13]. Our aim is to support people’s copy-and-paste attempts by providing familiar, easy-to-use, and effective guidance to help them make their copied code work.

This paper introduces our tool, called EUKLAS (Eclipse Users’ Keystrokes Lessened by Attaching from Samples)¹, which helps users to more successfully employ copy-and-paste strategies for reuse (see **Figure 1**). Euklas provides specific guidance for assisting users in fixing some copy-and-paste errors that are typically caused by copying only a part of the required code pattern [11]. The goal is helping users with reusing working code from others and to fix resulting errors.

JavaScript is one of the most popular scripting languages for web programming [15] and is used by a broad variety of users, from end-user developers [16] such as interaction designers [6], to professional programmers. In spite of this widespread use, JavaScript development tools often provide less programming support for users (e.g., no static code checking, limited auto-completions), compared with other, non-scripting languages such as Java. One reason may be that analyzing JavaScript is difficult since it has weak, dynamic typing that makes it challenging to perform static analyses at edit-time. We overcome this limitation as well as provide more sophisticated guidance for correcting errors resulting from copy-and-paste.

The following example presents a typical use case for the kind of copy-and-paste reuse of JavaScript code that is supported by Euklas. Jamie is a JavaScript developer and has found a code snippet on the web that creates various types of enhanced combo-boxes from which she wants to use one in her website. Jamie has to explore the example code in more detail to separate relevant pieces of code from irrelevant pieces. That is, she has to identify which functions, variables, and imports of JavaScript and CSS files are necessary to make her code work the same way as the example. This is often a time-consuming and cumbersome task especially since Jamie’s JavaScript editor does not provide any help. In contrast, Euklas can help Jamie with the integration process as it provides her with editing guidance. First, Euklas highlights errors, such as undefined variables, missing function definitions, and missing imports of CSS and JavaScript files, in her target code by inserting error markers  and squiggly underlines (see #1 in **Figure 1**). Second, Euklas also computes quick fixes for the identified errors to help Jamie easily correct them (see #2 in **Figure 1**). For example, it suggests copying the missing variable and function definitions from the source file and

offers to insert the missing import statements for the CSS and JavaScript files.

Euklas makes the following major contribution: It helps users to integrate JavaScript example code into their own projects by identifying some of the potential errors in the pasted code and recommending potential fixes, based on the consideration of the original code from which sections were copied. The editing guidance uses familiar interaction features of the Eclipse IDE and extends it with new detection and repair algorithms. These algorithms are based on heuristic edit-time source code checks for JavaScript, such as checking for uninitialized variables and undefined functions, and the analysis of the example file (from where code was copied) to provide useful quick fixes for the identified errors.

We evaluated Euklas in a preliminary user study with 12 people, comparing it to Eclipse’s JavaScript editor. The results suggest that the features have been implemented in a usable and effective manner, since participants were not confused that Euklas can sometimes be wrong or not helpful. The study also suggests that participants using Euklas were able to fix about two times as many errors as people in the control group, when integrating example code into their target systems.

2. Related Work

The reuse of example code consists of two main phases: 1) locating the example code and 2) integrating it into the target system. There are several systems that assist users in finding relevant example code (e.g., in repositories or on the web) [2, 3, 17, 18]. However, more relevant to our work are tools that support users in integrating code into their systems.

JDA (JavaScript Dataflow Architecture) aims to enable users that have no programming skills to build applications from JavaScript code that was found on the web [10]. Users write simple HTML commands to connect the different pieces of JavaScript code in a way similar to “pipes” in UNIX systems. However, JDA treats the pieces of JavaScript code as “black-boxes”, which means that users cannot access or change the internal code structure. In contrast, Euklas treats all JavaScript code as “glass boxes”, allowing users to access and copy even incomplete internal pieces of their code structure.

D.MIX targets web designers that are familiar with HTML and scripting languages, such as JavaScript. It enables them to build and share mashups created from pre-existing web sites. D.MIX inspired the design of Euklas as it follows a copy-and-paste approach that works on a richer representation of the selected data [19]. In that way, elements’ parameters can be changed after pasting them to D.MIX’s editing environment. We take the idea of D.MIX one step further, since Euklas provides specific editing

¹ Euklas is German for Euclase, which is a gemstone. Euklas is pronounced oy-class.

operations, in the form of quick-fixes, which help users to integrate their copied pieces of code into their projects.

The Looking Glass IDE helps middle school students to reuse functionality they find in other programs by helping them to understand how the code works [4]. It guides students to do this by enabling them to 1) record the execution of the program they are interested in, 2) identify the start and end of the functionality they are interested in, and 3) integrate this functionality in their new program. Euklas uses the idea of guiding users to support their copy-and-paste-based reuse strategies, but integrates this guidance with familiar interaction techniques in Eclipse.

JIGSAW is a plug-in for the Eclipse IDE that uses a copy-and-paste interaction technique for reusing Java code [9]. It inspired the design of Euklas, since Jigsaw assists developers with the integration of the reused source code into the developer’s own source code. Jigsaw compares the example code and the target code to suggest which pieces of the example code would fit best in the target code. However, Jigsaw can only work with pieces of example and target code that have similar AST (Abstract Syntax Tree) structures. Jigsaw was tested in a small study with two developers, which showed that the resolution of conflicts is cumbersome and not yet well supported by Jigsaw.

JSLINT is a popular tool for detecting errors in JavaScript code. JSLint works on a subset of JavaScript and its goal is to help programmers to write better code. This makes it different from Euklas’s error detection, since we do not restrict which features of the language can be used. In addition, Euklas supports users in fixing the errors.

Many JavaScript editors do not provide much programming support other than syntax highlighting. At the next level, IDEs like NetBeans provide simple code inspections, such as checking whether a variable has been used or not. One of the most advanced commercial products for programming JavaScript is WebStorm, which provides more advanced code inspections by leveraging JSLint. For example, it offers analyses to check whether variables and functions have been defined. In contrast to JSLint, WebStorm also offers quick fixes for the detected errors, allowing users to easily declare and define a variable, or to create a new empty function definition. However, the provided quick fixes do not analyze the files from where code was copied, making it impossible for WebStorm to offer the kind of quick fixes that Euklas provides.

Euklas’s copy-and-paste design is strongly inspired by Rosson’s and Carroll’s observations about “the reuse of uses in Smalltalk programming” [13]. They observed a reuse strategy where programmers copied and pasted a piece of code that they considered to be promising for the functionality that they intended to reuse. After pasting it into the target context, they let the environment provide editing directions about what would be necessary to make the code work. This was often accomplished in several

cycles of fixing and waiting for new editing suggestions, until all of the copied code was fixed.

Kim et al. report that related code snippets (e.g. referenced fields/constants and caller/callee methods) are usually copied together because they belong to the same functionality [12]. In addition, Ko et al. provided empirical evidence that the identification of related code snippets often fails in the first attempt, requiring programmers to spend time on identifying all relevant pieces of code [11].

Euklas supports users when they try to reuse code by copying-and-pasting it into their target code since Euklas will not only mark errors, but it also provides quick fixes for some of these errors (which might involve copying and pasting additional pieces of the example code). These steps are performed iteratively until all errors in the target code have been fixed.

3. EUKLAS

In this section we first discuss Euklas’s user interface design. Afterwards we will present how the system was implemented and integrated into the Eclipse IDE.

3.1 User Interface

Returning to the example discussed in the introduction, we explain how Euklas helps Jamie to integrate the combo-box code she found on the web. Jamie identifies that the con-

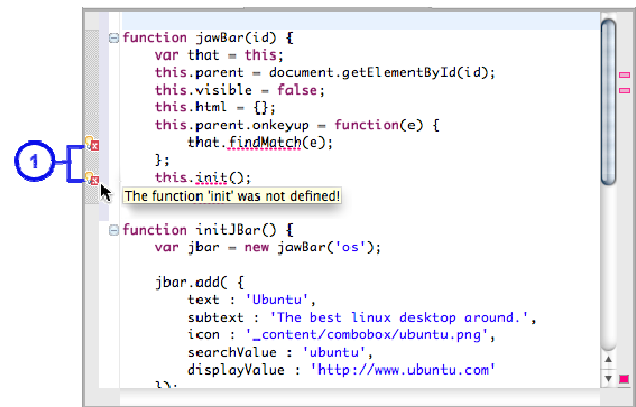


Figure 2. Euklas marks errors in the function `jawBar()` after Jamie pastes it into the target file.

structor (`jawBar(id)`, see Figure 2) is a promising piece of code for making the combo-box work. She copies and pastes the constructor into her target file. Euklas identifies two errors in the constructor: the two undefined function calls `findMatch(e)` and `init()` (see #1 in Figure 2). These are undefined because their function definitions are not available in the target file, since they have not yet been copied from the example file. Euklas uses Eclipse’s familiar marker system including the squiggle underlines and the error markers in the margin to indicate these two errors in the code.

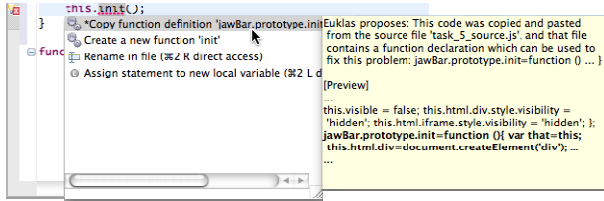


Figure 3. Euklas proposes the top two quick fixes for the undefined function `init()`. More information about the first (selected) proposed fix is shown in the beige pop-up at the right. The pop-up shows a short proposal explaining what Euklas intends to do, followed by a preview of how the code will look after the selected fix has been applied. Euklas augments Eclipse’s quick fix feature which is well-known by many programmers.

In addition to marking the errors, Euklas also suggests solutions for fixing them. By clicking on the error marker that refers to `init()`, Jamie gets four options to fix this error, as shown in **Figure 3**. Again, Euklas uses Eclipse’s functionality for showing the quick fix options. The quick fixes proposed by Euklas (the first two in the list in **Figure 3**) use Euklas’s icon to distinguish them from the quick fixes proposed by Eclipse itself (the last two in the list).

The first quick fix that Euklas proposes would copy the function `jawBar.prototype.init()` that is defined in the example file (the file from which Jaime copied the function `jawBar()`) to the target file. The second quick fix that Euklas proposes is a “default fix” that creates a new function, called `init()` that would have an empty function body. This is sufficient to remove the syntax error, but it would not serve Jamie’s goal of making the enhanced combo-box work in her website. We added this default option because it resembles the kinds of quick fixes that are offered by Eclipse’s Java editor and by WebStorm with which programmers may be familiar. The default option is also a fallback option for cases in which Euklas is not able to provide a better quick fix based on the example code.

Jamie reads the explanation in the beige window (see **Figure 3**) that describes how Euklas proposes to fix the error by using the example code. She decides that the first option is probably the correct fix for her situation and selects it. Euklas pastes the function `jawBar.prototype.init()` from the example file into the target file. This successfully fixes the missing definition of the function `init()`. However, the function `findMatch(e)` is still undefined and Euklas identified an additional error in the newly code. Jamie continues to fix the remaining errors until the code is fixed and the combo-box is working.

3.2 Implementation

Euklas was implemented on top of Eclipse’s JavaScript editor as a plug-in to the IDE. Eclipse’s JavaScript and HTML editors do not provide as much programming sup-

port. In particular, they only provide error highlighting for syntax errors and some basic quick fixes, such as “rename in file” and “assign statement to new local variable.”

JavaScript is a dynamic language, which makes it hard to reliably detect errors at edit time. Jensen et al. identify various situations that can cause runtime errors: invoking a non-function value (e.g. undefined) as a function, reading an undefined variable, and accessing a property that is ‘null’ or ‘undefined’ [20]. Starting with this list, we looked for additional errors that may arise from copy and paste operations. We identified three additional types of errors that are specific to JavaScript, and arise from its close relationship with HTML and CSS code. Our analysis resulted in the following list of potential copy and paste errors that can be detected by Euklas: 1) missing parameter definitions in a function’s parameter list, 2) missing local and global variable definitions, 3) missing function definitions, 4) missing CSS style sheet imports (e.g. for general layout definitions), 5) missing JavaScript file imports (e.g. scripts, which might be located on remote servers), and 6) missing HTML elements being accessed by the global JavaScript function `getElementById(HTML_Element_ID)`.

Euklas employs heuristic, static analyses on the Abstract Syntax Tree (AST) to find potential errors in JavaScript code. Euklas’s analysis of the AST is invoked if a user pastes a piece of code from an example file to the target file. Euklas provides error highlighting for all six error types mentioned above. However, Euklas currently only provides quick fixes for numbers 1-3, which we consider to be the more important types of errors, since they may cause runtime errors in the JavaScript code. Numbers 4-6 cause errors in the HTML part of the source code, which may cause problems with displaying the webpage correctly but do not necessarily lead to JavaScript execution-time errors.

Euklas’s code analyses are different from those that are used in other static code analysis tools, such as FindBugs [21]. Since JavaScript does not provide static typing of variables, it is impossible to run certain dataflow analyses. Euklas instead employs heuristic analyses, which has proven to be a successful alternative approach for JavaScript [22]. The implementation of Euklas assumes that the code of the used examples is syntactically and semantically correct, which means that the code is executable and delivers a useful result (a limitation of the current Euklas prototype). Euklas uses three different algorithms to identify the errors listed above. The first detection algorithm checks for undefined variables (error numbers 1 and 2), the second detection algorithm checks for undefined functions (error number 3), and the third detection algorithm checks for the errors in the HTML code (error numbers 4-6).

The heuristics used for identifying errors numbered 4-6 are more fragile than the heuristics used for detecting errors numbered 1-3. The reason is that content can be loaded dynamically, e.g. loading CSS code within JavaScript code.

To visualize our confidence level of the more fragile heuristics, error numbers 4, 5, and 6 are marked with a warning marker ⚠ instead of an error marker ❌.

There are some cases in which Euklas produces false positives or false negatives, due to the used heuristics. This means that the analyses can either mark correct code as an error, or miss marking some existing errors in the code. Consider the following example that caused one of the errors in the evaluation. Euklas cannot detect errors that are caused by “hiding” variables: `var that = this; that.findMatch(e);`. In this case Euklas is not able to detect that the function call `that.findMatch(e);` could be related to the object `this` instead of to the object `that`. It would produce a false positive because it does not find the function definition of `findMatch(e)` that belongs to the object `that`.

Euklas not only finds potential errors in the code, but it also can propose fixes for these errors. Euklas remembers the links between the position in the example file and the specific region in the target file when users copy and paste code. Each target file can have multiple regions and each region has exactly one link to exactly one corresponding region in an example file from which the code was copied. Euklas maintains the meta-data about these connections in memory. It updates the regions when the target files are edited to keep the metadata correct. Euklas loads and saves this metadata as part of the project when Eclipse starts and shuts down.

Currently, Euklas needs to have access to each of the example files, i.e., they must be present in Eclipse’s workspace. We decided that this would be a reasonable limitation for the Euklas prototype, to enable investigation of the usability and usefulness of its ideas, but a real system should handle example code from external files or individual snippets of code (e.g., code copied from the web, or small pieces of code in blog posts).

The ASTs of the example files are analyzed for potential solutions for some classes of errors. For example, if the error in the target file refers to an undefined variable, Euklas analyzes the example file(s) for a definition of this variable (under the assumption that the code in the example file(s) is syntactically and semantically correct). If there is a suitable definition in an example file, Euklas creates a quick fix proposal to copy that definition from the example file into the target file, and adds this quick fix to the error marker in the target file. If the user selects a proposed quick fix, Euklas parses the AST of the target file to insert the copied AST piece from the example file.

4. Evaluation

The evaluation had the goal of answering two questions: Can users understand and successfully use Euklas’s features? Is the integration of example code faster and more

correct with Euklas than with Eclipse’s standard JavaScript editor or compared to a more sophisticated JavaScript editor, which already offers some error detection features?

4.1 Participants

We feel that Euklas is most appropriate for people who have some experience with using JavaScript, and we did not want to have to train people on how to use Eclipse. Therefore, we recruited participants who had about one year of experience using Eclipse (for developing in any language), and who had done at least one programming project in JavaScript (using any development environment). We recruited 12 participants (10 male, 2 female) from our local university community. Each participant was compensated \$15 for participating. Their ages ranged from 19 years to 37 years (median: 25, *s.d.* 5). The participants had diverse backgrounds, such as business administration and software engineering.

4.2 Apparatus and Materials

The study was conducted in our lab on the university grounds. We used an iMac running a standard Eclipse installation including the WTP (Web Tools Project) plug-in. The control group used Eclipse’s JavaScript editor, which does not highlight errors or provide quick fixes, and each experimental group used one of two versions of Euklas: “Euklas lite” and “Euklas full”. The main difference between the two versions was that “Euklas lite” only analyzed the target file to identify errors and to compute quick fixes, while “Euklas full” also analyzed the example file(s). We chose to have these two different versions of Euklas to be able to separately study the following two aspects. First, we wanted to explore the effects of providing error highlighting and quick fixes with a user interface similar to Eclipse’s Java editor. Second, we were interested in the effects of having the more sophisticated analyses and quick fixes that took the example file(s) into account.

We set up Eclipse’s workspace with the files that were used for the study. Each of the examples could be executed in Firefox to allow participants to explore the examples and test whether their target code was working.

4.3 Procedure

The study used a between-subjects design using the tool version as independent variable with three conditions: Eclipse’s JavaScript editor (Control condition), “Euklas lite” and “Euklas full”. The dependent variable was whether the tasks were completed successfully. The between-subjects design was chosen because it would have been impossible for the participants to redo the tasks, since they would have known the answer to each task after the first run. Participants were randomly assigned to each group, since all subjects reported approximately equal JavaScript

programming experience. Participants in all groups received a spoken introduction to the study and signed the consent form. All participants were briefly introduced to Eclipse’s JavaScript editor. Members of the experimental groups received an additional introduction to Euklas’s extensions to Eclipse’s JavaScript editor, e.g. information about its error and warning markers and the quick fixes. All task descriptions explained which code should be copied and what the desired results would be after the code was pasted and all errors were fixed.

All participants performed the tasks in the same order. The tasks were designed to cover all cases in which Euklas provides support, as well as cases where it provides misleading help or does not help at all. Participants were allowed to work on each task for a fixed amount of time, which varied from 7 to 20 minutes, based on the task’s difficulty. The researcher who conducted the study measured the time it took participants to work on each of the tasks and stopped them if they ran over the maximum time allowed for each task.

Task	No. of Errors	Types of Errors*	Source LOC	Copied LOC	Max. Time (min.)
1	1	2	112	10	7
2	3	2, 3	112	13	7
3	2	1, 2	122	29	7
4	3	2, 5, 6	103	5	10
5	5	3, syntax error	218	125	12
6	10	2, 3, 4	1507	996	20
Sum	24		2174	1178	63

Table 1. Summary of the tasks (*types of errors according to the list in section 3.2)

Table 1 shows the number of errors per task (errors that occurred after the first paste operation), the types of errors included in the tasks (based on the types of errors presented in section 3.2), the lines of code (LOC) of the source example files (HTML and JavaScript files together), the total number of lines that had to be copied to solve the tasks, and the maximum time participants were given to complete each of the tasks. We judged tasks to be finished successfully if the code could be executed without causing any errors in Firefox and if it performed as required by the specification.

In task #1, participants had to choose and integrate the correct part of a larger function for setting a cookie, while they had to choose and integrate a different part of that function for getting a cookie in task #2. For task #3, participants had to integrate an enhanced pop-up menu into the target file. In task #4, participants had to add a pushpin to a ‘Bing’ map and integrate it into the target file. To complete task #5, participants had to integrate an enhanced combo box (the one that we described above as Jamie’s example) into their target file. Finally, in task #6, participants had to integrate an inner window into their target file.

At the end of the study, participants filled out a questionnaire. The questions primarily used five-level Likert scales, but some were open answer.

4.4 Results

The analysis of the data shows that participants using “Euklas full” completed more tasks (Control: 7/24, “Euklas lite”: 13/24, “Euklas full”: 18/24.) and fixed about twice as many errors (average = 22.25) as the control group did (average = 11.75) when integrating the example code into the target system.

For the analysis we combined the tasks shown in **Table 1** into an “easy” group (tasks 1-3) and a “difficult” group (tasks 4-6). There are several reasons for combining the tasks into the two groups. The median average success rate for all six tasks across all three conditions was 0.5. We defined “easy” tasks as those with an average success rate at or above the median, and “difficult” tasks as those below the median. Another reason was that the tasks had been designed with an increasing difficulty. We allotted more time for completing the more difficult tasks. Finally, without combining the tasks we would have not been able to perform any statistical analysis, due to the highly differentiated completion values. The same reasoning justifies the statistical tests on the number of errors fixed.

“Easy” Tasks	“Difficult” Tasks
1: 0.75	4: 0.33
2: 0.50	5: 0.17
3: 1.00	6: 0.42

Table 2. Average success rate per task (std. error: 0.13)

We ran a logistic regression analysis of the tasks, which showed a statistically significant difference for the success rate with respect to the tasks’ difficulty, i.e. between the “easy” and “difficult” tasks: likelihood ratio $\chi^2[1] = 14.79$, $p < 0.0001$. We also analyzed how the success rate of participants was affected by the tool used (Control, Euklas lite, Euklas full) and the task difficulty (easy tasks, difficult tasks). A nominal logistic regression analysis showed that the effect of the tool on the success rate was significant: likelihood ratio $\chi^2[2] = 14.97$, $p = 0.0006$. The effect of the task difficulty on the success rate was also significant: likelihood ratio $\chi^2[1] = 19.34$, $p < 0.0001$. The interaction was not significant. In other words, participants were more successful based on the tool used across all tasks. We performed pairwise nominal logistic regression tests to determine which levels of tool use had a significant effect on the success rate: Control vs. “Euklas lite”: likelihood ratio $\chi^2[1] = 3.21$, $p = 0.074$; Control vs. “Euklas full”: likelihood ratio $\chi^2[1] = 14.90$, $p < 0.0001$; “Euklas lite” vs. “Euklas full”: likelihood ratio $\chi^2[1] = 4.44$, $p < 0.035$. In summary, “Euklas full” was better than both Control and “Euklas lite”. “Euklas lite” was slightly better than Control.

Looking at the number of corrected errors, i.e. the number of errors that were fixed by the participants, the data

also shows that “Euklas full” participants fixed almost twice as many errors as participants in the control group. We ran an ANOVA to test for the effect of the tool used and the task difficulty on the number of corrected errors, and found that the number of corrected errors differed significantly across the three tools, $F[2,9] = 13.82$, $p < 0.002$. We ran contrasts to compare the tools with each other. The Control group made fewer error corrections than the “Euklas lite” group ($F[1,9] = 13.37$, $p = 0.005$) and also corrected less errors than the “Euklas full” group ($F[1,9] = 26$, $p < 0.001$).

Unfortunately, we were not able to analyze differences for the timing data, i.e. how long participants took to complete each of the tasks, since there were so many tasks that participants failed to complete in the maximum allotted time. The analysis of the final questionnaires provides more details on the differences between the two Euklas versions. Participants who used “Euklas full” agreed that it usually provided helpful quick fixes (average 4 out of 5). One “Euklas full” user nicely expressed why he liked it: “Intelligent error messages and debugging makes it infinitely more useful, especially when it checks against the source of your copy.” “Euklas lite” got lower ratings in terms of helpfulness (average 3.25 out of 5) of its quick fixes from the participants who used it, which was not surprising. One of the “Euklas lite” participants suggested the following improvement, which reflects exactly the improvements in “Euklas full”: “Provide [a] pop-up menu which can suggest to copy blocks of code to resolve errors.” The questionnaire also asked whether Euklas speeds up the integration of JavaScript code compared to other editors. Participants using “Euklas full” strongly agreed with this statement (4.75 out of 5) while participants using “Euklas lite” did not share this view (3.75 out of 5).

5. Discussion

The tasks that were combined in the “easy tasks” group contained a maximum of three errors and participants were allotted the same maximum time of seven minutes for each of the tasks. Even though the number of errors was low and the scripts were short, participants in the Euklas conditions were more likely to complete the three tasks and fixed more errors than participants in the control group. However, due to the rather low complexity of tasks in this group, we did not expect that “Euklas full” participants would have big advantages compared to “Euklas lite” participants, which was reflected by the very similar results in the two Euklas conditions.

For the three tasks in the “difficult tasks” group, however, the situation is different. “Euklas full” users had a larger advantage than participants in the other two groups. The code they used was longer and more complex (1828 LOC instead of 346 LOC), contained more errors (18 instead of

6) and the errors were more difficult to find (see **Table 1**). Therefore, we increased the maximum time that participants were allowed to work on each task. The task completion rates were higher for participants using “Euklas full” than participants using “Euklas lite” and participants in the control condition. Also, the results show that participants using “Euklas full” were able to fix more errors than participants in any of the other two conditions.

Overall, the results show that the “Euklas lite” version, which provided error detection features and standard quick fixes, already brought many improvements in comparison with Eclipse’s JavaScript editor, which did not offer such features. This is not surprising, since we know that highlighting errors helps users. “Euklas full”, which offered additional analyses and additional quick fixes, improved performance even more. “Euklas full” especially showed advantages when the pieces of copied code were longer and/or more complex (as in Tasks 4-6). Euklas’s approach of considering a broader context for the computation of potential quick fixes has implications for many other programming languages, such as Java.

Using Eclipse’s marker feature for highlighting and fixing errors seems to be an appropriate UI choice as participants had a very positive attitude towards this approach and considered it to be easy to learn. The most important aspect about using this feature is that participants knew what to expect from the provided quick fixes. They knew that these were usually right in “Euklas full”, but that they could sometimes also be wrong and might not be helpful. Such a situation was simulated in task 5, where participants had to fix a syntax error. The task also included a missing function definition that was not detected by either of the two Euklas versions. Participants were generally able to distinguish between these cases where they got standard “quick fixes” from Eclipse and where they got more sophisticated quick fixes from Euklas, since they did some manual checks to see if a proposed quick fix was appropriate or not before they used it.

There were a total of 24 errors that occurred after participants pasted the code from the source files into the target files. “Euklas full” provided 20 helpful suggestions for fixing these errors, plus 33 generic suggestions (e.g. declare a function with an empty body). Eclipse generally added in two additional suggestions per error that were not helpful at all. Participants sometimes picked one of Euklas’s generic fixes (e.g., for one of the missing functions in task 5). They knew that this was not sufficient for making the code work and therefore looked at the example code to manually find the correct piece of code. In cases where Euklas did not show an error at all (e.g., for one of the missing functions in task 5), participants did not perform worse than the participants in the control group, so there appears to be no disadvantage to using Euklas. However, the evaluation has some limitations. First, the evaluation only had the rather

small number of 4 participants in each condition. Second, participants used the tools for only one hour. Third, the tasks may not have been representative of realistic tasks. Although the pieces of code were real-world examples that were taken from the web, we had to reduce the complexity of dealing with these examples to limit the amount of time spent on completing each of the tasks.

6. Conclusion and Future Work

In this paper, we presented our new Eclipse plug-in, Euklas, which supports JavaScript programmers in some of the tasks when using copy-and-paste-strategies for reusing example code. This kind of reuse does not create clones in a codebase [14], but instead helps users to reuse working code from others to introduce new functionality to their codebases and understand how this functionality can be implemented. Euklas supports these strategies by analyzing the target code for errors and by suggesting fixes for these errors. An important innovation is augmenting the fixes through an analysis of the code from where the example was copied. Our evaluation shows that Euklas's users were able to fix a much higher number of copy-and-paste related errors than participants who used Eclipse's JavaScript editor, which does not provide any debugging support.

Euklas's main contribution is analyzing the file from where code was copied to provide more detailed error descriptions and much better quick fixes for these errors. We think that applying Euklas's ideas to editors used for other languages (e.g., Python, Java, and C++) could increase programmers' performance in these situations in the same way it did in our case for JavaScript.

In addition to reducing the limitations discussed above, future work could include implementing some additional features for Euklas. One idea for providing improved error detections would be to design a heuristic for analyzing the context of a variable to try to determine its runtime type. This would allow the implementation of better quick fixes, since the system could distinguish between variables with the same name, but of a different type.

Euklas points to a future where programming support tools better help developers by taking into account all of the available contextual information, and the provenance of resources used. The success of Euklas shows that this approach is feasible and can be successful, and developers can make effective use of recommendations, even when they are heuristic. The incorporation of extended copy and paste support into different kinds of editors would be a first step into this future.

Acknowledgements

This work was conducted in 2010/2011, when the first author was a postdoc at Carnegie Mellon University. The

authors would like to thank Sara Kiesler for her invaluable assistance in the data analysis. The first author thanks the Alexander von Humboldt-Foundation for his Feodor Lynen Research Fellowship for Postdoctoral Researchers. This research has also been supported by SAP, Adobe and the National Science Foundation, under grant CCF-0811610. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the sponsors.

References

- [1] Lee, B. et al., Designing with interactive example galleries. in CHI '10, 2257-2266.
- [2] Bajracharya, S. et al., C., Sourcerer: An internet-scale software repository. in ICSE SUITE Workshop '09, IEEE, 1-4.
- [3] Brandt, J. et al., S.R., Example-centric programming: integrating web search into the development environment. in CHI '10, 513-522.
- [4] Gross, P.A. et al., A code reuse interface for non-programmer middle school students. in UI '10, 219-228.
- [5] Hartmann, B. et al., What would other programmers do: suggesting solutions to error messages. in CHI '10, 1019-1028.
- [6] Myers, B. et al., How Designers Design and Program Interactive Behaviors. in VL/HCC'08, 177-184.
- [7] Brandt, J. et al., Two studies of opportunistic programming. in CHI '09, 1589-1598.
- [8] Stylos, J. and Myers, B.A., Mica: A Programming Web-Search Aid. in VL/HCC '06, 195-202.
- [9] Cottrell, R. et al., Semi-automating small-scale source code reuse via structural correspondence. in FSE-16, 214-225.
- [10] Lim, S.C.S. and Lucas, P., JDA: a step towards large-scale reuse on the web. in OOPSLA '06, 586-601.
- [11] Ko, A. J. et al., Eliciting Design Requirements for Maintenance-Oriented IDEs. In ICSE '05, 126-135.
- [12] Kim, M. et al., An Ethnographic Study of Copy and Paste Programming Practices in OOP. in ISESE '04, 83-92.
- [13] Rosson, M.B. and Carroll, J.M. The reuse of uses in Smalltalk programming. ACM TOCHI, 3 (3). 219-253.
- [14] Rahman, F., Bird, C. and Devanbu, P., Clones: What is that smell?. In MSR '10, 72-81.
- [15] Crockford, D., JavaScript: The Good Parts, Sebastopol, CA: O'Reilly & Associates, 2008.
- [16] Lieberman, H., Paternò, F. and Wulf, V. End User Development. Springer, Dordrecht, 2006.
- [17] Sahavechaphan, N. and Claypool, K., XSnippet: Mining For sample code. in OOPSLA '06, 413-430.
- [18] Holmes, R., Walker, R.J. and Murphy, G.C. Approximate Structural Context Matching: An Approach to Recommend Relevant Examples. IEEE TSE, 32 (12). 952-970.
- [19] Hartmann, B. et al., Programming by a sample: rapidly creating web applications with d.mix. in UIST '07, 241-250.
- [20] Jensen, S.H., Møller, A. and Thiemann, P., Type Analysis for JavaScript. in 16th International Symposium SAS, Springer, 238-255.
- [21] Ayewah, N. et al., Using Static Analysis to Find Bugs. IEEE Software, 25 (5). 22-29.
- [22] Ko, A. and Wobbrock, J., Cleanroom: Edit-Time Error Detection with the Uniqueness Heuristic. in VL/HCC '10, 7-14.