Cleanroom: Edit-Time Error Detection with the Uniqueness Heuristic

Andrew J. Ko and Jacob O. Wobbrock
The Information School | DUB Group
University of Washington
{ajko, wobbrock}@uw.edu

Abstract

Many dynamic programming language features, such as implicit declaration, reflection, and code generation, make it difficult to verify the existence of identifiers through standard program analysis. We present an alternative verification, which, rather than analyzing the semantics of code, highlights any name or pair of names that appear only once across a program's source files. This uniqueness heuristic is implemented for HTML, CSS, and JavaScript, in an interactive editor called Cleanroom, which highlights lone identifiers after each keystroke. Through an online experiment, we show that Cleanroom detects real errors, that it helps developers find these errors more quickly than developers can find them on their own, and that this helps developers avoid costly debugging effort by reducing how many times a program is executed with potential errors. The simplicity and power of Cleanroom's heuristic may generalize well to other dynamic languages with little support for edit-time name verification.

1. Introduction

Dynamic languages such as JavaScript, Perl, PHP, Python, and Ruby have quickly become the foundation of the interactive web. And with good reason: their support for implicit declaration, reflection, code generation, and other dynamic features frees developers to quickly express and iterate on code without worrying about variable declarations or types. For example, because JavaScript's objects are essentially hash tables, developers can customize objects with unique properties and other metadata (as in `object.checked = true`, where no such property `checked` has been declared), construct property references at runtime (e.g., `calendar['week'+week]`), and dynamically modify and generate functions on the fly. It also allows developers to use *reflection* in more facile ways, inspecting whether objects have certain properties (e.g., `object.hasOwnProperty('checked')`).

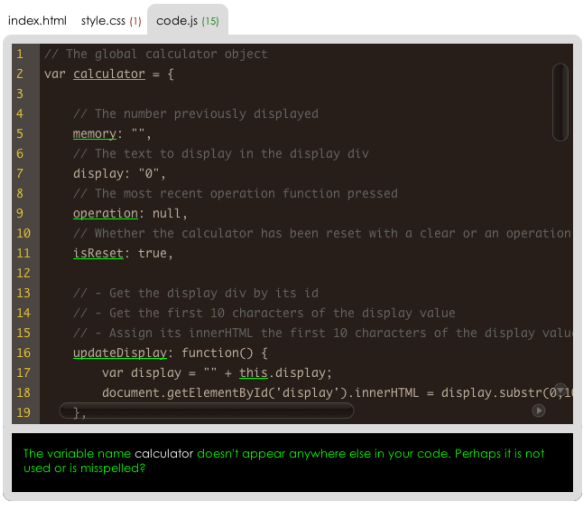

Figure 1. Cleanroom: a web-based, bug-finding JavaScript/HTML/CSS editor.

Of course, this benefit also imposes a great cost: because program entities need not be declared, there are few opportunities for compilers or interpreters to warn developers about identifiers that might not exist, deferring the detection of many errors to runtime. For example, if the developer who wrote the code in the previous paragraph had typed `cheked` instead of `checked`, there would not have been a warning that the property was undeclared until the code executed; worse yet, many identifier names appear in string literals, making them impossible for traditional type checkers to verify.

While the *semantic* name resolution in dynamic languages is not possible, there has been little research on alternative forms of verification. In this paper, we present one such verification, which we call the *uniqueness heuristic*: any name or pairs of names that occurs only once in a program is likely unintended. We explore the merits of this heuristic with *Cleanroom* (Figure 1), a new web-based editor that implements the uniqueness heuristic for HTML/CSS ids and class names, Javascript variables, object properties, function names, and string literals.

Although the uniqueness heuristic is simple, we have found it to be powerful: through an online experiment that asked users to complete a JavaScript-based calculator program, we show that Cleanroom identifies a variety of legitimate errors, that Cleanroom users identify these errors faster and that they find these errors *before* executing their programs. These results demonstrate that the uniqueness heuristic, and Cleanroom's implementation of it for popular client-side web scripting languages, is an effective means of detecting problems with identifiers, without requiring developers to declare names.

In the rest of this paper, we discuss other forms of bug detection and prevention for dynamic languages and then describe Cleanroom's design and implementation. We then describe the details and results of our experiment and discuss the implications of our work on the future of editors and dynamic languages.

2. Related Work

Although the existence of a name can be difficult to verify in dynamic language code, there are a variety of approaches that can detect and prevent some forms of errors. First and foremost, many errors in HTML and CSS code occur not because of *semantic* errors, but because of *syntax* errors. HTML and CSS validators can find syntax violations and check for duplicate HTML identifiers and reserved words. These validators do not check semantics; for example, they do not verify whether an HTML class name referenced in a CSS rule appears anywhere among the HTML files of a web site. Some systems attempt to statically validate dynamically generated HTML, but require the use of specification languages [2].

Beyond syntax errors, there are many tools that cat detect potential errors by identifying error-prone use of language constructs. One notable tool is *JSLint* (http://jslint.com), which checks for a missing semicolons, inappropriate line ending punctuators, risky expressions, unreachable code, and also any variable that is not declared with the `var` keyword. JSLint does *not* check property names of objects, because properties can be dynamically assigned and deleted, but it does generate a list of all identifiers string literals, which developers can look through for misspellings. Google's *Closure* compiler (http://code.google.com/closure/) also catches common JavaScript errors, including redeclared variables, function names that mask variables, redefined namespaces, references to variables before declarations, and potentially unused object properties. While both tools detect many important errors, they also detect many false positives, because the heuristics used are so strict. Such high false positive rate can limit the utility of warning messages [5].

Modern web development tools also provide error prevention features through auto-completion. For example, Dreamweaver's *code hinting* keeps track of JavaScript frameworks declared with object literal notation, to support some auto-completion (e.g., typing `YAHOO.util.` will show all known properties and functions of the utility object in the Yahoo UI toolkit). Visual Studio 2008 has similar features, also supporting some type checking. For example, it is aware that the standard `document.getElementById()` function returns an `HTMLElement` type, and propagates this knowledge through a function, so that auto-completion can display an object's functions and arguments. Both auto-completion tools prevent errors by having developers *select* identifiers rather than *type* them. Unfortunately, neither system propagates knowledge through aliasing, where one variable is assigned to another. Moreover, these features only work when code is syntactically correct, forcing developers to correct syntax errors elsewhere in their code before getting feedback about problems with the code they are writing.

In recent years, researchers have begun to focus on more sophisticated ways of detecting errors in web applications. For example, researchers have used user-session data to generate tests [3,8]; these, however, require an application to be deployed. Another approach crawls the state space of AJAX applications [4], detecting many real errors in the process, but requires developers to configure the system for each program, choosing application-specific invariants (which the authors admit, can be quite difficult). Artzi et al. describe a similar approach for PHP programs, using symbolic execution and model checking to capture logical constraints on inputs, which are then used to check for crashes automatically [1]. These approaches are heavyweight and require significant developer effort to use and configure. In contrast, Cleanroom requires no configuration, and developers can even ignore it until it highlights potential problems.

3. Design & Implementation

In contrast to prior work, Cleanroom contributes a simple, easy to implement form of error detection feedback, which catches many of the same errors in prior work, and many new kinds of errors, with less developer effort. In this section, we describe the rationale behind Cleanroom's features and interaction design and then describe its implementation in detail.

3.1. Interaction Design

Cleanroom's design embodies two major ideas. First is the uniqueness heuristic: *names or pairs of names that appear only once across a project are likely to be wrong*. For example, the name `console` in Figure 2 and the pair `animal.species` in Figure 3 appear only once in their respective programs, leading to a warning.

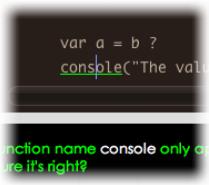

Figure 2. Errors can still be detected in the presence of syntax errors.

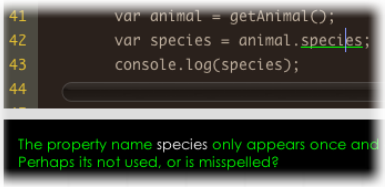

Figure 3. Cleanroom detects sequences of names that only appear once in the code, identifying potentially undeclared properties.

What makes this heuristic valuable is its simplicity and effectiveness: it is both easy to implement, efficient to evaluate, and easy for developers to understand, while also catching a variety of undeclared, unused, or accidental names, across a variety of program constructs. For example, Table 1 lists the types of errors that can be detected in HTML/CSS/JavaScript code through the sole application of the uniqueness heuristic. Moreover, because the heuristic only requires tokenized identifiers and string literals to work, it can catch errors in the presence of syntax errors. This enables feedback to appear consistently throughout a developers' editing, uninterrupted by missing delimiters (as in Figure 2).

Cleanroom's second major idea is in its *immediate feedback* about potential errors, which appear after every keystroke as simple underlines (reminiscent of the squiggly red underlines for misspelled words in Microsoft Word). Again, the power of this idea is in its simplicity: by having the editor always provide feedback about the presence of names, Cleanroom's feedback plays a *validating* role in editing tasks, as described in Figure 4. For example, while a developer is typing the name `lastElement`, Cleanroom displays a green underline. This confirms what the developer knows, that the identifier is not yet complete. When the developer is done typing, one of two things will happen. If `lastElement` appears elsewhere, the green underline will go away; this confirms the developer's expectations, which is useful feedback. If `lastElement`

Undeclared HTML identifiers and class names
Undeclared JavaScript variables
Unused identifiers and functions
Accidental typos, such as `identfier`
Casing typos, such as `endoffile` vs. `endOfFile`
Invalid HTML tags and attribute names
Invalid CSS properties and values
Undeclared object properties, such as `YAHOO.Dom.getStyle`
Undeclared object functions
Missing source file includes
Potentially invalid string literals containing names
Identical names used in incompatible contexts

Table 1. Potential HTML/CSS/JavaScript issues that Cleanroom detects with its uniqueness heuristic.

does *not* appear elsewhere, the green underline persists; this violates the developer's expectations, drawing their attention to a potential problem. Cleanroom provides the same type of feedback when warnings appear or disappear in a *different* file after a keystroke. For example, Figure 5 shows a scenario in which the clear() function is called, causing the file warning count to drop from 2 to 1 in `index.html` and 14 to 13 in `code.js`.

While this feedback simple, it is also quite powerful: the experience of *expecting* the green underline to disappear, but still seeing it, creates a substantial surprise, which draws the developers attention to the potential error. This is akin to the *surprise-explain-reward* design strategy [11], in which some knowledge gap draws a user's attention to an explanation, which enables them to take some action to gain some reward. In the case of Cleanroom, the surprise is the discrepancy between a developer's expectations about whether the Cleanroom warning will disappear and whether it actually does. The explanation is the warning message and the reward is the fixed bug.

In addition to playing the role of confirming and conflicting feedback, Cleanroom's warnings also play a *reminding* role, when the warning is on a declaration. For example, Figure 6 shows Cleanroom warnings on object property and function names that do not appear elsewhere. When these names are later referenced, the developer receives confirmation that the function or variable exists, as well as that the name they referenced was the one they intended.

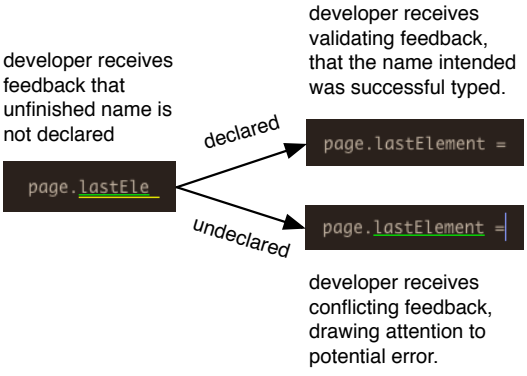

Figure 4. Cleanroom's timely, immediate feedback about the existence of names, either confirming or violating a developers' expectations.

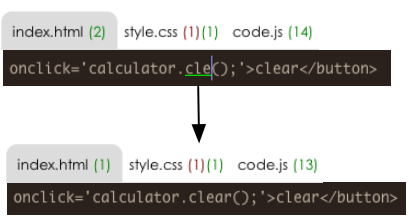

Figure 5. File-level feedback also confirms or violates developer expectations.

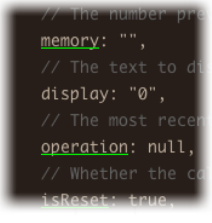

Figure 6. Declaration warnings highlight unused code.

Figure 7. Cleanroom suggests other names a developer might have meant.

In addition to simply warning about errors, Cleanroom also provides error messages with recommendations of similar names, as in Figure 7. When these recommendations come from a syntax or API, as in the case of CSS property names, HTML tags, or standard JavaScript browser globals such as `document`, `window`, and `console`, they can play an educational role, teaching the developer valid names.

3.2. Implementation

Cleanroom consists of several ANTLR tokenizers (http://www.antlr.org/), the Bespin code editor (https://bespin.mozillalabs.com/), JSLint (http://jslint.com), and the Cleanroom code. The Bespin editor was customized to invokve the Cleanroom analyses after each keystroke and to draw underlines beneath tokens with warnings.

The Cleanroom algorithm involves a short sequence of analyses, executed each time a file is loaded or modified. First, each file that has not been processed or was recently modified is incrementally tokenized, adding and removing tokens for characters in response to modifications. The tokenizer is also responsible for assigning a Cleanroom *token type* to tokens that may represent names (primarily identifiers and string literals) and reporting these name tokens back to Cleanroom.

After tokenization, Cleanroom updates a table of name tokens that appear across all files. The table is divided first by the token types assigned by the tokenizer. The types in our HTML/CSS/JavaScript implementation include `HTMLTag`, `HTMLAttributeName`, `HTMLClass`, `HTMLID`, `CSSPropertyName`, `CSSValue`, `JSFunction`, `JSProperty`, `JSVariable`, and `JSLiteral`. Within each of these types is a hash table of name token lists, separated by name. It is important to note that token types span languages: for example both the HTML and CSS tokenizer generated `HTMLTag` names, because both languages refer and use `HTMLTag` names. Moreover, HTML code can contain JavaScript and CSS code, so the names of a particular token type can appear across files written in different languages.

String literals are processed in a special way. If the entire literal matches the JavaScript identifier format (based on a simple regular expression) it is promoted to a name token. If the string literal successfully parses as a JavaScript statement, it is tokenized as JavaScript code, and used to identify additional name tokens. All other string literals are ignored.

Once the table of name tokens is constructed, warnings are computed for all identifiers in the table, by token type. For each type, names whose name token list is often length 1 are selected for warnings. However, each token type is also allowed to compare these unique names against any number of other dictionaries. For example, HTML tag names and CSS property names are compared against a dictionary of valid names, preventing warnings about `<body>` tags, for example, which usually appear only once. JavaScript function name tokens are also compared against standard global JavaScript names, such as `document`, `console`, `window`, and others.

Each Cleanroom token type can also declare other token types to which names should be compared. For example, `JSFunction` tokens are not only compared with each other, but also with `JSPropertys`, `JSVariables`, and `JSLiterals`. Similarly, `JSLiterals` are compared against all other JavaScript names, since they may refer to functions, variables, or properties.

For each name token that appears only once, and does not appear in any of the additional dictionaries, Cleanroom generates a warning. Each token type declares its own message format. Then, to each custom warning, we append a list of similar names (as in Figure 7). We compute these similar names by computing the Levenshtein string distance [6] between the warned name and all names in the token type's dictionaries (specifically, we use the Wagner-Fisher algorithm [10], which treats transposition as an atomic operation). Names whose distance is 1 are included.

To generate warnings about *pairs* of names, Cleanroom's tokenizers also maintain *previous* and *next* links between name tokens, when names are separated by one of an accepted list of delimiters. For JavaScript, these delimiters were `.` (dot) and `[` (left square bracket) both of which are valid ways of identifying properties and functions of objects. These links are used to generate a global hash table of concatenated name pairs, which is used to identifier pairs that occur only once. This analysis also utilizes any custom name pair dictionaries to avoid false positives. For example, we converted the standard browser APIs into a name pair dictionary, which contains standard pairs such as `document.getElementById()` and `window.location`.

In general, generating Cleanroom warnings for a language involves customizing a tokenizer, defining a set of token types, supplying dictionaries of valid names for each, as well as indicating which token types should be compared to one another. The rest of the Cleanroom implementation is language-independent.

4. Evaluation

Our goals for Cleanroom were to help developers notice legitimate errors before execution, so that they may fix them more quickly than they would through debugging. To do this, we designed an online experiment, comparing a version of Cleanroom that showed warnings (the *Cleanroom* condition), to a version of Cleanroom that tracked warnings, but did not show them (the *control* condition). This allowed us to observe how developers' behavior changed as a result of Cleanroom's highlighting. Both versions also showed JSLint warnings, to give the baseline version some novelty for recruiting purposes. JSLint also identifies some of the same errors that Cleanroom can (namely undeclared variables through implied global detection).

4.1. The Calculator Task

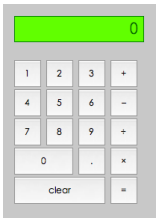

Figure 8. The calculator that developers implemented.

Developers were asked to complete the graphical calculator in Figure 8. In the reference implementation, the UI was implemented in HTML and CSS with HTML class names and ids. Event handlers were attached to each button's `onclick` attribute, to call JavaScript functions that operated the calculator. A `code.js` file contained a `calculator` object literal, with the properties `memory`, `display`, `operation`, and `isReset`, and the functions `pressDigit`, `pressOperation`, `clear`, `add`, `subtract`, `multiply`, `divide`, and `updateDisplay`. The calculator worked by appending digits to `display` with the `pressDigit()` function, saving this string in `memory`. The `pressOperation` function assigned the name of the operation function to later call on the calculator to the `operation` property. When the operation was the *equals* button, the name of the function stored in `operation` was retrieved using reflection and called. Each of the operation functions used the string stored in `memory` and `display`, parsed each of these numbers, assigned the result to `display`, and then updated the HTML display tag. Finally, `isReset` kept track of whether the calculator had just finished a *clear* or *equals* operation, indicating that the next digit pressed would overwrite the value currently displayed.

The experiment version of the calculator omitted the code summarized in Table 2, including all event handlers and functions, except for `updateDisplay`. Overall, developers were responsible for writing 43 lines of JavaScript code. To ensure developers wrote similar code, we converted each line of the reference implementation into natural language (without using exact identifier names), providing developers with line-by-line specifications in comments above each function. This allowed developers to focus on implementing the specifications, rather than conceiving of a solution.

	reference code	experiment differences
index.html	60 lines of HTML, with 18 inline event handler calls.	missing 18 inline event handler calls, attached to buttons' `onclick` attributes.
code.js	101 lines of JavaScript code in an object called `calculator`.	76 lines of JavaScript, missing function implementations.
style.css	44 lines of CSS, 4 rules.	same

Table 2. The differences between the reference and experiment versions of the calculator implementation.

To test whether developers had completed the tasks, the editor injected several automated tests upon each preview, checking whether pressing the calculator buttons would provide correct answers for 9+5, 9–5, 9x5, 9/5, as well as properly display 0 after pressing the clear button. The results were shown alongside the Cleanroom editor, as in Figure 9.

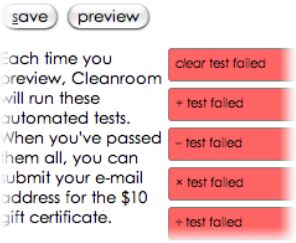

Figure 9. The automated test feedback, updated after each preview.

4.2. Developer Recruiting

Developers were recruited from university mailing lists known to have web developers. An email was sent with the subject "try Cleanroom, a new bug-finding JavaScript/HTML/CSS editor," describing Cleanroom and the study, with a link to the application. Developers were offered a $10 at Amazon for completing the task. According to Google analytics, 94 potential developers visited the site from the direct link while the experiment was open. Forty logged in, revealing the editor and task description. Of these, 22 typed for more than 3 minutes; of these, 8 *Cleanroom* and 8 *control* developers made significant progress. Although success on task was not an explicit factor in our study, it is worth noting that 3 of 8 Cleanroom developers succeeded (and 3 more were missing only one function) and 5 of 8 *control* developers succeeded; the rest of the developers made significant progress on the task. Our final data consisted of these 16 developers and their warnings.

Our goal was to involve developers who knew JavaScript, HTML, and CSS syntax, and regularly used these languages to develop web sites. Therefore, upon arriving at the site and choosing a log in name, developers were asked to complete the statement, "In the past month, I've written JavaScript code ..." and select from *never*, *once*, *weekly*, *daily*, or *hourly*. Translating these responses to a 1-5 scale, with 5 being *hourly*, *Cleanroom* developers averaged 2.0 and *control* developers averaged 2.5. Analysis using ordinal logistic regression showed no significant difference between the two groups ($\chi^2(1, N=16) = 0.73$, n.s.).

measurement	operationalization
whether the warning was **active**	Warnings were considered active if there was no keystroke that caused them to disappear.
the time **duration** of the warning	The time between warning creation and either (1) the warning elimination or (2) the last recorded keystroke, less all periods of time inactivity greater than 1 minute.
The **kind** of token on which the warning appeared	One of `HTMLTag`, `HTMLAttributeName`, `HTMLClass`, `HTMLID`, `CSSPropertyName`, `CSSValue`, `JSFunction`, `JSProperty`, `JSVariable`, and `JSLiteral`.
Whether the warning was a **declaration**.	Whether the token appeared after the `function` or `var` keyword.
explicit **creation**	Whether the warning was appeared because of an operation on the token itself.
explicit **elimination**	Whether the warning disappeared because of an operation on the token itself.
executions	How many times the program was executed while the warning was active.

Table 3. Data extracted from logs about warnings.

4.3. Data Collection, Extraction, and Cleaning

As developers worked, Cleanroom tracked every keystroke applied to each file, every time the application was previewed, and every time the Cleanroom window focus was lost or gained, with each even time stamped. Cleanroom also recorded each warning it identified, the keystroke that created it, and the keystroke that caused it to disappear. With each warning, we recorded information about the token it regarded, including the token's text, its kind (one of the types listed in section 3.2) and whether the token was a declaration. Using the data recorded above, we extracted the warning measurements summarized in Table 3. We ignored warnings that were shown as identifiers were typed, by focusing on those that still appeared after 2 seconds of typing inactivity. Moreover, out of the 845 warnings obtained in our experiment, 136 of them were discarded because they were generated as a result of file loading delays. This left 709 data points to be used in our analyses, 332 from Cleanroom developers and 377 from the control group.

4.4. Results

Let us begin by discussing our analysis approach. Our unit of analysis was a single warning, resulting in unbalanced repeated measures on each developer based on their number of warnings. Logistic regression was used for dichotomous, categorical, and ordinal results. For continuous measures, a mixed-effects model analysis of variance was used with subject properly nested within cleanroom and modeled as a random effect to handle within-subject correlation.

Were warned names fixed? One of our primary questions was whether the warnings that Cleanroom identified (regardless of whether they were shown), were actually addressed by developers in each condition. If both conditions were actively working on the task, we would expect both groups to successfully remedy legitimate warnings at similar rates. This was indeed the case. *Cleanroom* developers fixed 258/332 (78%) of warnings; *control* developers fixed 308/377 (82%). The difference was non-significant (χ^2(1, N=709) = 0.12, n.s.), confirming that Cleanroom identified real errors, because warnings were addressed even when they are not highlighted.

Were warned names explicitly modified? In addition to checking how often warnings were addressed, we also checked whether warnings were *directly* addressed through the explicit modification of the identifier they highlighted (as opposed to changes to other identifiers or large blocks of text deletion). Of all addressed warnings, *Cleanroom* developers explicitly modified 98/164 (60%), whereas *control* developers explicitly modified only 69/203 (34%). This difference was significant (χ^2(1,N=367)=14.16,p<.001), showing that *Cleanroom* developers were more likely to focus their edits specifically to problematic names, and not on segments of code indirectly related to the warning.

How long did warnings persist? Having confirmed that the Cleanroom warnings identify legitimate errors, and focus users' attention on them, to what extent did *showing* Cleanroom warnings help developers save time? The median *Cleanroom* warning lasted 141 seconds (from 1 to 3,085), whereas the median *control* warning lasted 223 seconds (from 1 to 15,558). This difference was significant (F(1,16.8)=9.18, p<.01), suggesting that Cleanroom significantly reduced how long errors remained in code. (Note that because of duration's high skew, values were log-transformed; this is a common practice for data with a power law distribution). Of course, because the task had no time limit, the duration of warnings that were *not* addressed depended on how long developers worked. Excluding periods of inactivity lasting longer than 1 minute, Cleanroom developers worked an average of 29.3 minutes (sd=21.1), while control developers worked 51.5 (sd=26.8). Despite the lower time spent by Cleanroom users, these differences were not significant (F(1,14) = 3.40, n.s.). This suggests that part of the differences in durations may have been due to the control group's extra time on task.

How often was the program executed while warnings were active? Since Cleanroom's warnings were potential errors, it generally behooved programmers to address warnings *before* executing their programs, to avoid debugging effort. Cleanroom developers' warnings, on average, persisted through about 1.7 executions (sd=4.6), whereas control developers' warnings, on average, persisted through about 6.4 executions (sd=17.0). This difference in (log-transformed) executions showed a trend in Cleanroom's favor, although it was not quite significant (F(1,14.3) =4.49, p=.052). This finding suggests that Cleanroom does not only reduce the duration of potential errors, but also the debugging effort required to detect errors.

What kinds of errors did Cleanroom find? Having demonstrated Cleanroom's benefits quantitatively, we now turn to a qualitative analysis of the types of errors Cleanroom identified. By inspecting the names highlighted in the Cleanroom condition, we saw that the warnings covered the full range of error types described in Table 1, including undeclared names, unused names, and a variety of typos (including `parseFLoat`, `getElementByID`, `onlcick`, `alert_box`, etc.).

However, Cleanroom identified more than just typos. One user used the word `dim` to declare a variable, apparently borrowing from Visual Basic syntax, but it was warned by Cleanroom. There were several cases where the developers called calculator functions as if they were global, but upon receiving Cleanroom feedback, added `calculator` before the call. Another developer mistakenly used the word `sum` to refer to the calculator's `add` function, and fixed the mistake less than a minute later. Another developer attempted to give a variable the type `int`, but removed it after seeing the highlight. All of these errors go beyond simple typos, helping developers identify misunderstandings about the JavaScript language and other semantic slips.

What non-errors did Cleanroom highlight? Surprisingly, the *only* false positives were on the `add`, `subtract`, `multiply`, and `divide` function declarations. The implementation did not call these functions explicitly, but instead were referred to in string literals; it turned out that there was a bug in the tokenization of string literals inside of string literals in HTML, which prevented strings like `"calculator.pressOperation ('add')"` from including the `add` in analyses. In other words, for at least the calculator task, *none* of Cleanroom's warnings were false positives.

4.5. Study Limitations

A number of limitations may influence the validity and generalizability of our study results. For example, we recruited student developers, whose expertise may have been limited compared to experienced web developers, who may have less need for Cleanroom's error detection. Moreover, we only studied 16 developers; expertise varies enough that this number may have been too small to capture the different ways that developers might respond to Cleanroom's feedback.

Another limitation was our study's online deployment. We do not know exactly how often developers left the task and returned, or to what extent they analyzed Cleanroom's code instead of working on the task. In fact, one of the developers wrote the first author in the middle of the task pointing out a SQL injection vulnerability in Cleanroom's implementation.

There were also some issues with our measurements. For example, the duration of a warning can be affected by many things other than *when* a developer noticed the problem: a developer might declare a variable, but then not reference it for an arbitrary amount of time. Moreover, each execution was likely unique in which warned names it executed; it is difficult to tell from our data how often a warned name was *actually* executed.

Finally, our experiment did not include a condition without warnings, since both conditions included JSLint warnings. A condition without warnings may have revealed different strategies than those observed in our experiment. For example, developers who were reminded that their code might have errors might have been more vigilant in avoiding them.

5. Discussion

Our experiment results suggest that Cleanroom's uniqueness heuristic, and its simple feedback about uniqueness warnings, is effective at helping developers find and fix errors before executing. However, the heuristic does have several limitations and opportunities for improvement. We now discuss these in detail.

5.1. Design Limitations

One of the heuristic's major limitations is that it can catch typos that occur once, but not typos that occur multiple times. For example, in the development of Cleanroom itself, the first author repeatedly typed `identifer`, omitting the final `i` in the name. Cleanroom would not have detected these errors, because the misspelling was not unique. Such misspellings might be detected by incorporating word processor style spell checking to words in identifier parts, although care would have to be taken to avoid false positives on abbreviations and non-words.

One form of identifier error not accounted for is in the *construction* of names. For example, imagine a web page with several elements representing weeks, each with an HTML id with the prefix `week`, followed by a number (e.g., `week1`, `week2`). A simple way to operate on these weeks as a set is to construct these ids in a loop (e.g., `"week" + number`). Future versions of Cleanroom could reason more intelligently about such dynamically generated prefixed and postfixed names.

While there were no false positives in our experiment, the uniqueness heuristic may not always be right: some names that appears only once may be correct; some literals that look like names may not be. We found, however, that the simplicity of the heuristic made false positives more tolerable and sometimes helpful. For example, in our earlier `"week" + number` example, Cleanroom would have warned about `week`; this would have been a false positive, but it would have also reminded the developer that the line has the potential for error (because the number may not exist, or the prefixes might change). This is consistent with prior studies, which show that users' willingness to tolerate false positives has mostly to do with users' ability to understand a system's reasoning [9].

5.2. Design Alternatives

In designing cleanroom, we considered several alternative designs, drawn primarily from text entry research. One was that of *digraph likelihood*, in which an editor would flag unlikely pairs of characters. This would catch a variety of typos. For example, consider the code `<script type="text/javscript">`: the `vs` character pair is not common in English writing, suggesting an error. We considered computing digraph frequencies from general English, a broad corpus of JavaScript programs, or individual programs. However, we found that digraph frequencies are highly idiosyncratic: low-frequency digraphs in the NYTime's front page appear as high-frequency digraphs in the Seattle Times front page. Moreover, the uniqueness heuristic appears to subsume most errors that unlikely digraphs would detect.

We also investigated the *causes* of typos, to look for ideas about how to prevent or detect typos. For example, typos tend *not* to occur in the first and last character of a word. This comes from a finding that if the first and last characters of a word are correct, people can still read sentences [7]. Another observation is that spelling errors come in many kinds, including character duplication, mistaken spelling of vowel or consonant sounds, and other mistakes. These facts could have been used to decide whether or not to show a uniqueness warning. However, after experimenting with some examples, we found that the simplicity of the uniqueness heuristic would have been diminished by selectively omitting warnings. In other words, the *predictability* of the heuristic was its strength.

5.3. Design Generalizability

How well would Cleanroom work for other dynamic languages, such as Perl, PHP, Python, and Ruby? Like Javascript, none of these languages require variables to be declared, all of them are dynamically typed, and most of them support reflection and anonymous functions. Where they differ is in their variable scoping. For example, in Perl, variables can be referenced as `$scalars`, `@arrays`, `%hashes` and `&functions`, and so the name space can be quite broad. In contrast, Ruby variable's scope is determined by lexical characteristics: locals are lowercased, constants start with an uppercase letter, globals are prefixed by $, and so on. This would allow Cleanroom to be more confident in its detection of unique names within these separate name spaces. PHP is different in that by default, functions have no access to the global namespace, except through the `global` keyword; such declarations are their own source of error. Therefore, Cleanroom's uniqueness heuristic applies well to all of these languages, but how names are scoped might enhance or restrict the heuristic's applicability and false positive rate.

6. Conclusions

In this paper, we have presented Cleanroom, a web-based HTML/CSS/JavaScript editor that warns developers about names and sequences of names that appear only once. We have shown that Cleanroom detects real errors, that it helps developers find these errors more quickly than developers can find them on their own, saving costly debugging effort. Cleanroom is just one point in a design space of error detection tools that exploit human causes of software errors. In future work, we hope to explore other ways of detecting errors by exploiting similar patterns in developer behavior.

7. Acknowledgements

This material is based in part upon work supported by the National Science Foundation under Grant Number CCF-0952733.

8. References

1. Artzi S., Kiezun A., Dolby J., Tip F., Dig D., Paradkar A., Ernst M.D. 2010. Finding bugs in web applications using dynamic test generation and explicit state model checking. *IEEE Transactions on Soft. Engineering*, to appear.
2. Braband, C., Møller, A., and Schwartzbach, M. 2001. Static validation of dynamically generated HTML. *Workshop on Program Analysis for Soft. Tools and Engineering*, 38-45.
3. Elbaum, S., Karre, S., and Rothermel, G. 2003. Improving web application testing with user session data. *Int'l Conf. on Soft. Engineering*, 49-59.
4. Mesbah, A. and van Deursen, A. 2009. Invariant-based automatic testing of AJAX user interfaces. *Int'l Conf. on Soft. Engineering*, 210-220.
5. Kim, S. and Ernst, M. D. 2007. Which warnings should I fix first? *ACM Foundations of Soft. Engineering*, 45-54.
6. Levenshtein, V. I. 1965. Binary codes capable of correcting deletions, insertions, and reversals. *Doklady Akademii Nauk SSSR* 163(4), 845-848.
7. Rayner, K. (1998). Eye movements in reading and information processing: 20 years of research. *Psychological Review*, 124 (3), 372-422.
8. Sprenkle, S., Gibson, E., Sampath, S., and Pollock, L. 2005. Automated replay and failure detection for web applications. *Automated Soft. Engineering*, 253-262.
9. Stumpf S., Rajaram V., Li L., Wong W-K., Burnett M., Dietterich T., Sullivan E., and Herlocker J. 2009. Interacting meaningfully with machine learning systems: Three experiments. *Int'l J. of Human-Computer Studies* 67, 639-66.
10. Wagner R.A. and Fisher M. 1974. The string-to-string correction problem. *J. of the ACM*.
11. Wilson A., Burnett M.M., Beckwith L., Granatir O., Casburn L., Cook C., Durham M., and Rothermel G. 2003. Harnessing curiosity to increase correctness in end-user programming. *ACM Conf. on Human Factors in Computing Systems*, 305-312.