Managing Duplicated Code with Linked Editing

Michael Toomim, Andrew Begel, Susan L. Graham
Department of Computer Science
University of California at Berkeley
{toomim, abegel, graham}@cs.berkeley.edu

Abstract

We present Linked Editing, a novel, lightweight editor-based technique for managing duplicated source code. Linked Editing is implemented in a prototype editor called Codelink. We argue that the use of programming abstractions like functions and macros—the traditional solution to duplicated code—has inherent cognitive costs, leading programmers to chronically copy and paste code instead. Our user study compares functional abstraction with Linked Editing and shows that Linked Editing can give the benefits of abstraction with orders of magnitude decrease in programming time.

1. Introduction

The task of managing duplicated or "cloned" code[1] has occupied the minds of programmers for the past 50 years. During this time, researchers and practitioners have developed a variety of techniques for removing or avoiding it by employing functions, macros and other programming abstractions. Functional abstraction was designed into early programming languages, such as Fortran and Lisp. Object-oriented programming, originating with Simula-67, has provided further mechanisms for parameterized reuse to avoid duplication. Aspect-oriented programming has allowed cross-cutting duplication to be abstracted. Engineering practices like Refactoring [18] and Extreme Programming [6] have promoted specific methodologies of abstracting duplicated code. In the last decade, a multitude of tools have been developed (both in research and in industry) that help programmers semi-automatically find and refactor existing duplication into functions, macros and methods [3, 4, 5, 8, 10, 12, 14, 15, 17, 19, 23, 24, 25, 27, 28, 29, 30, 31, 33, 35, 36, 37, 40]. Given this long-term commitment to programming abstractions as a solution to duplicated code, it stands to reason that there should be little duplication left in practice.

However, duplicated code remains chronically entrenched in our programs. Recent studies estimate that the Linux kernel (as of 2002) is 15–25% duplicated [1], the GNU Compiler Collection (1999) is roughly 9% duplicated [14], the X Window System's source code (1995) is roughly 19% duplicated [3], and the Sun Java JDK [38] (2002) is 21–29% duplicated [25]. These are all large, highly developed, well-known open software systems. Some privately developed systems are much worse off, for example Ducasse et. al. [14] analyzed an internal Cobol payroll server (1999) with 60% duplicated code, and Baker [2] analyzed a closed AT&T software system (1992) with 38% duplication. In a 2004 ethnographic software engineering study by Kim et. al., expert programmers produced an average of four code clones per hour [26]. These data show that duplicated code is pervasive—despite the years of research and development in employing function, macro, and other programming abstractions, duplicated code has not gone away.

We believe that the problem is due to an over-reliance on language-level programming abstractions: abstractions can be difficult to create and use, and may be impossible to implement. As a result, programmers *will* write duplicated code—our approach is to mitigate its disadvantages. We introduce *Linked Editing*, an editor-based interaction technique for managing duplicated code. With this technique, the programming environment keeps track of code clones and provides enhanced visualization and editing facilities to the programmer, allowing him or her to understand and modify many code clones *as one*—but without incurring the cognitive costs of a heavyweight language-level abstraction.

We implemented Linked Editing in a prototype editor called Codelink, and conducted a user study to evaluate it. The user study showed that Codelink can provide benefits comparable to functional abstraction, but with much less programming work. The results suggest that programmers would use Linked Editing in situations in which they would otherwise duplicate code.

[1] We use "duplicated code" and "cloned code" synonymously to mean two or more multi-line code fragments that are either identical or similar, particularly in their structure.

The contributions of this research are the following: a cognitive account of the ways duplicated code impedes the task of programming, and the abstraction costs that cause programmers to duplicate code; a novel editor-assisted technique embodied in the Codelink tool for solving the problem; and an empirical user study comparing the benefits and drawbacks of Codelink with functional abstractions when applied to some types of duplicated code.

We first describe the problems of duplicated code in Section 2. We then propose a theory of why programmers duplicate code in Section 3. We illustrate Linked Editing, Codelink, and how they solve the duplicated code problems in Section 4, and our user study and results in Section 5. The remaining sections, 6, 7, and 8, describe Codelink's implementation, future work, and related work.

2. The duplicated code problem

We wish to eliminate the *problems* of duplicated code, rather than the code itself. Thus, we must understand how duplicated code impedes the task of programming. We interviewed three programmers (computer science graduate students at U.C. Berkeley), and combined the results with a survey of the literature and our own analysis to identify the following four problems with duplicated code:

Verbosity. Redundant cloned code creates clutter and obscures meaningful information, making code difficult to understand.

Tedious, repetitive editing. Edits made to one clone must often be made to its copies. Thus, a single modification often requires many edits, making sustained modification unwieldly.

Lost clones. Although edits to one clone must often be made to other clones as well, there is no way of getting to those other clones from the first, or of even realizing that the others exist. Missed edits lead to inconsistent code.

Unobservable inconsistency. Even if programmers find all clones and edit each one, it is impossible to verify that the edits have been made *consistently*—that the common regions are identical, and the differences are retained—without manually comparing each clone's body, word-by-word, and hoping that no important details were missed.

These problems, and the prevalence of duplicated code, define the *duplicated code problem*.

3. Why programmers duplicate code

Programmers employ functions, macros, classes, aspects, templates, and other programming abstractions to reduce duplication. The identical sections of the clones become the body of the abstraction's definition, and the differences become parameters. However, abstractions can be costly, and it is often in a programmer's best interest to leave code duplicated instead. Specifically, we have identified the following general *costs of abstraction* that lead programmers to duplicate code (supported by a literature survey, programmer interviews, and our own analysis). These costs apply to any abstraction mechanism based on named, parameterized definitions and uses, regardless of the language.

Too much work to create. In order to create a new programming abstraction from duplicated code, the programmer has to analyze the clones' similarities and differences, research their uses in the context of the program, and design a name and sequence of named parameters that account for present and future instantiations and represent a meaningful "design concept" in the system. This research and reasoning is thought-intensive and time-consuming.

Too much overhead after creation. Each new programming abstraction adds textual and cognitive overhead: the abstraction's interface must be declared, maintained, and kept consistent, and the program logic (now decoupled) must be traced through additional interfaces and locations to be understood and managed. In a case study, Balazinska et. al reported that the removal of clones from the JDK source code actually *increased* its overall size [4].

Too hard to change. It is hard to modify the *structure* of highly-abstracted code. Doing so requires changing abstraction definitions *and* all of their uses, and often necessitates re-ordering inheritance hierarchies and other restructuring, requiring a new round of testing to ensure correctness. Programmers may duplicate code instead of restructuring existing abstractions, or in order to reduce the risk of restructuring in the future.

Too hard to understand. Some instances of duplicated code are particularly difficult to abstract cleanly, *e.g.* because they have a complex set of differences to parameterize or do not represent a clear design concept in the system. Furthermore, abstractions themselves are cognitively difficult. To quote Green & Blackwell: "Thinking in abstract terms is difficult: it comes late in children, it comes late to adults as they learn a new domain of knowledge, and it comes late within any given discipline." [20]

Impossible to express. A language might not support direct abstraction of some types of clones: for instance those differing only by types (float vs. double) or keywords (if vs. while) in Java. Or, organizational issues may prevent refactoring: the code may be fragile, "frozen", private, performance-critical, affect a standardized interface, or introduce illegal binary couplings between modules [41].

Programmers are stuck between a rock and hard place. Traditional abstractions can be too costly, causing rational programmers to duplicate code instead—but such code is viscous and prone to inconsistencies. Programmers need a flexible, lightweight tool to complement their other options.

4. Our approach: Linked Editing

We propose Linked Editing: a novel technique for visualizing and editing duplicated code based on the programming environment rather than the programming language. With this technique, two or more code clones are identified as being similar and are persistently linked together. The differences and similarities are then analyzed, visualized, and recorded, allowing users to work with all linked elements simultaneously, or particular elements individually. This "linked" editing model eliminates the problems of duplicated code (verbosity, tedious editing, lost clones, and unobservable consistency) without requiring extra work from the programmer. We implemented a prototype of Linked Editing in an extension to XEmacs we call Codelink.

We illustrate Linked Editing and its benefits with a scenario. We begin with the duplicated code shown in Figure 1.1: two methods `wake()` and `wakeAll()` from the threads implementation of an educational Java operating system used at U.C. Berkeley. Notice that the only difference between the two methods is that one has an `if` statement where the other has a `while` loop. This type of difference, as mentioned in Section 3, is impossible to abstract directly in Java. We will show, instead, how a programmer would use Linked Editing (in Codelink) to manage this code.

4.1. Linking code

First, the programmer "links" the two clones in the editor. To do so, she selects the first clone with her mouse, and then selects the second while holding Control (creating a second simultaneous selection). Once both clones are selected (Figure 1.2), she clicks the "Link Selections" button. The system now runs a differencing algorithm (discussed in Section 6) on the two code fragments, and displays the results by highlighting all common regions in blue, and differences in yellow (Figure 1.3). This visualization provides an immediate benefit to the programmer—she can now easily scan the duplicated code and understand precisely and completely how the two pieces of code are alike and how they differ—alleviating the *unobservable inconsistencies* problem. Linked Editing can be used in this way as a pure visualization aid, for instance by a third-party programmer trying to understand existing duplicated code.

Figure 1. (1) Before linking two similar Java methods in Codelink. (2) After selecting the methods. (3) After linking the methods.

4.2. Editing linked code

The core capability of Linked Editing, however, is that the programmer can edit all linked clones simultaneously. As she moves her editor's text cursor in a clone, a blue *ghost cursor* appears in the corresponding position of each related clone. Now, if she decides to add a debugging statement to each instance, she simply types the statement in one of the clones, and the other clones are modified simultaneously with each keystroke (Figure 2.1). She can also navigate among linked clones with keyboard hot-keys. Thus, Linked Editing allows the programmer to edit all instances of a particular clone at once, as if they were a single block of code. This is our solution to the *tedious, repetitive edits* problem.

Unlike previous work in simultaneous editing [32], however, Linked Editing allows the programmer to just as easily make a change to a *single* clone individually. Clicking the "Linked Editing" button causes the ghost cursors to disappear and the main cursor to revert to a thin black bar. The programmer can now remove the debugging statement from one of the clones by simply deleting it from that clone (Figure 2.3). Codelink re-analyzes the code after each keystroke and makes the new differences yellow. This simple operation would have been much more difficult using macros, since the programmer would have had to modify the macro definition, making it support a new parameter, in addition to each of its instantiations—in order to support an isolated change to a single instance.

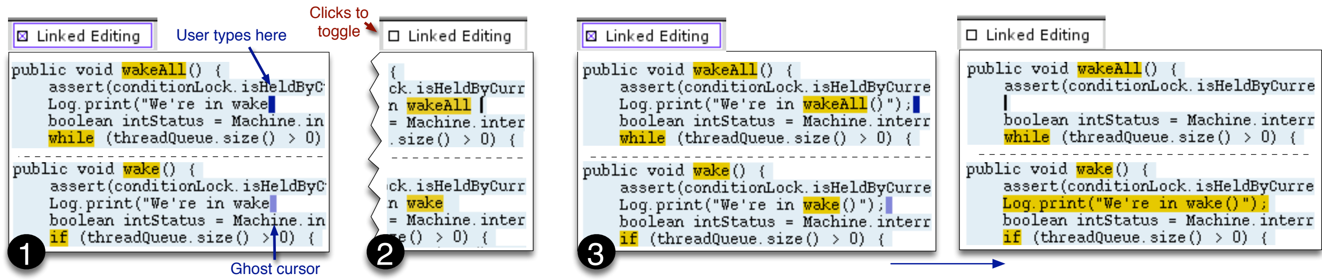

Figure 2. (1) Adding a line to two clones. (2) Modifying one instance. (3) Deleting line in one instance.

4.3. Elision, persistence and refactoring

To further enhance code understanding, Linked Editing provides *selective elision* of clones—the programmer can hide the redundant common regions of linked code with ellipses, leaving the differences visible, as shown in Figure 3. Now, the clone looks remarkably similar to a traditional function definition and use. In this way, Linked Editing alleviates the *verbosity* of duplicated code. If the programmer clicks the triangle, or moves her cursor into the elided procedures, it toggles back to its expanded form.

When the programmer saves her file, the "links" between clones are saved as meta-data, so that any programmer with a Linked Editing-enabled editor that opens the file will see and have access to the links. With this feature we expect the system to alleviate the need for clone-finding tools. Since the cost of linking code is so small, it would be to her benefit to create links between most clones that she runs across in practice, improving her ability to read and edit them. In this way, we imagine that programmers would systematically add clone links to their code-base—thus "documenting" all existing clones as a side-effect of their normal work. By making cloned code an explicit property of a program, the *lost clones* problem of duplicated code is alleviated.

Finally, Linked Editing has been specifically designed to augment, not replace existing tools and techniques. A programmer may decide that a set of linked clones are really part of a higher-level concept, and want to refactor the code into a method or macro now that it has been written and tested. The Linked Editing tool will be able to automate this task, using its existing knowledge of the code's similarities and differences, and present the programmer with a dialog box in which to name the method or macro and its parameters. Programmers without Codelink could then take advantage of the link, since they could work with the functionally abstracted form rather than the Linked Editing form. In addition, code could be transformed in the other direction: from abstract functions and macros to concrete inlined code. This would make some abstracted code, like complicated macros, easier to understand. In fact, a design goal is to allow programmers to fluently move back and forth between program representations, to suit their changing tasks and style of work. (This feature is not yet implemented in Codelink.)

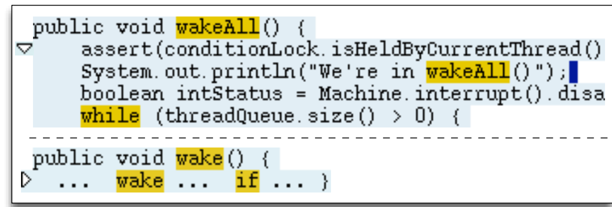

Figure 3. An elided clone looks similar to a function definition and use

4.4. Summary of benefits

Our approach addresses the difficulties described in section 3. With Linked Editing, it is easy to *create* new abstractions—the user just selects a set of clones with the mouse or keyboard and clicks "link selections". It is easy to *modify* all clones since changes to any one clone are automatically and simultaneously made to all linked clones, and it is easy to modify a *single* clone instead by changing editing modes. It is easy to *understand* the correspondences between clones because Linked Editing provides an as-you-edit incrementally-updated visualization of clone similarities and differences in addition to clone-aware navigation and ghost cursors. It is easy to *predict* edits under simultaneous editing because all blue regions are guaranteed to remain identical after arbitrary edits. Linked Editing is also very *general* and immune to language faults: it is language-independent, can link arbitrary syntactic structures, and can even link clones between files of different languages.

5. User study

We conducted a user study to compare the use of Codelink with programming abstractions. Our hypotheses were that programmers would be able to link clones with Codelink in much less time than it would take to abstract the clones, and that Codelink would provide programmers with comparable benefits after linking the code.

We paid 13 students from U.C. Berkeley to participate in the study. Subjects had a diverse range of programming skill, ranging from graduate students in Computer Science to introductory-level undergraduates. Subjects performed their programming tasks in the Scheme programming language since functional abstractions in Scheme are expressively powerful and well-understood by students at Berkeley, thus mitigating biases from language-specific abstraction costs. We expect the results to transfer to functions, macros and methods in other languages as well.

We used a within-subjects experimental design. With both functional abstraction and Codelink, subjects were asked to perform a set of programming tasks: (1) to abstract or link two short pieces of cloned code, (2) to perform a modification task requiring new code to be added to both clones or instances (the same code to each), and (3) to perform a modification task requiring new differences between each clone or instance. We believe these programming tasks illustrate the tradeoffs in editing duplicated/abstracted code. Although both sets of programming tasks followed the general sequence given above, the specific tasks and code were very different for each technique to eliminate learning effects. The pairing between techniques and task-sets and the ordering of techniques used were fully counterbalanced to eliminate ordering, learning and task biases.

Before performing each set of programming tasks, subjects completed a short (5-10 minute) tutorial to teach them about the technique (functional abstraction or codelink, depending on the condition) and what was expected of them on the tasks. The tutorial walked them through the three types of programming tasks, with very simple code and modifications. Subjects filled out a questionnaire after each experimental task-set to assess the particular technique paired with that task-set. The entire study lasted between 30 and 90 minutes. The programming tasks were recorded with a screen-capture program, and audio was captured and merged into the video to facilitate data analysis. Subjects did not test their code; they were rather instructed to stop when they thought their code would work.

5.1. Evaluation metrics

We recorded two dependent variables: the **time** it took subjects to link or functionally abstract the code (Step 1 in each set of programming tasks), and the **ratings** they gave each technique on the post-task questionnaires. Abstraction/link time was measured from the subject's first keypress after reading the task instructions to the last keypress before flipping to the next task's instructions.

Do you think it is easier or harder to **understand** the code with the functional abstraction than without it?

1	2	3	4	5	6	7
Much Easier	Moderately Easier	Slightly Easier	Same	Slightly Harder	Moderately Harder	Much Harder

Figure 4. Sample questionnaire question

On the post-programming questionnaires, subjects rated each technique along the following five metrics: **maintainability**, **understandability**, **changeability**, **editing speed**, and **editing effort**; reported on a 7-point semantic differential scale. Each question asked subjects how the technique they used (functional abstraction or Codelink) helped or hindered them on the programming tasks, as compared to editing the duplicated code directly. An example question is shown in Figure 4. With these questions, we judged how each technique helped or hindered programmers *after* the initial abstraction or linking of code.

Finally, the experimenter asked subjects the following question verbally: "If you had the Codelink tool in your editor or programming environment, and the other programmers on your programming project had it too, how likely is it that you would use it in your own programming work?" The responses were classified into three groups: *probably or definitely wouldn't use Codelink*, *not sure*, and *probably or definitely would use Codelink*. Because this question was only incorporated into the study after the first three subjects had been run, we only received 10 responses instead of 13.

5.2. Results

The timing data, shown in Figure 5, verified our first hypothesis: whereas functional abstractions took 13:06 minutes to create on average, the mean time to link code with Codelink was 22 seconds. This is a dramatic difference—the link time was 2% of the abstraction time. This result is statistically significant with $p < 0.01$, obtained with a two-tailed t-test. The standard deviations for the functional abstraction and link times were 9:23 minutes and 11.6 seconds, respectively.

The questionnaire results are shown in Figure 6. Surprisingly, on average subjects rated Codelink higher than functional abstractions with respect to *all* metrics. This difference was statistically significant for **understandability** and **changeability** (paired two-tailed t-test, $p < 0.05$). Thus, the questionnaire data more than verified our hypothesis that Codelink provides benefits comparable with those of functional abstractions. We find this very encouraging.

Since subjects did not test their code, they sometimes

Technique	Mean Abstraction Time	Std. Dev.
Functions	13min 06sec	9min 23sec
Codelink	22sec	12sec

Figure 5. Functional abstraction refactoring time vs. Codelink link time

made programming errors. The most serious errors occurred when making large edits during step 1 of functional abstraction.

Finally, we wanted to know if subjects would use Codelink in their real-life work. 9 of the 10 subjects asked reported that they "would" or "probably would" use Codelink in their work. The other one reported "not sure."

In summary, subjects learned how to use Codelink quickly, and in very little time achieved results competitive with functional abstraction. Almost all subjects said they would use Codelink in their regular work. These results indicate that Linked Editing holds much promise as a solution to the duplicated code problem. Furthermore, the fact that functional abstraction was rated lower than Codelink provides evidence for the costs of abstraction discussed in Section 3.

6. Implementation

Codelink is implemented within Harmonia-Mode [39], an XEmacs extension that provides interactive program analyses via the Harmonia interactive language-analysis framework [7, 21]. It can be applied to many programming languages, thanks in part to the language-independence of the Harmonia framework. A thorough user-centered, iterative design process was used to design and verify the usability of the user-interface prior to the study described in Section 5.

Our difference analysis uses the dynamic programming version of the longest-common-subsequence (LCS) algorithm [11], operating on subsections of lexical units in the program obtained by splitting each token at "word" boundaries—dashes, underscores, whitespace characters, etc. The sequence of tokens returned by this LCS algorithm became the blue "common" regions, and all other tokens became the yellow regions.

When an edit is made to a single clone, an incremental version of the difference algorithm is run. It isolates the token subsections in which the edit occurred, and then re-runs the original difference algorithm in the smallest yellow region that fully encompasses this edited region. This method of re-differencing changed regions is not optimal in all situations, but worked without problems in the user study.

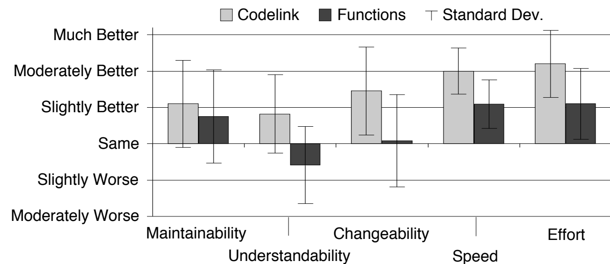

Figure 6. Questionnaire ratings for Codelink and functional abstraction

7. Future work

Although the study results are promising, there are a number of obstacles to be overcome before Codelink is a viable option in real-world projects. The LCS algorithm used in the prototype, although adequate for the user study, has two shortcomings: it takes $O(n^k)$ time (for k clones of size n), and does not always report the most intuitive set of differences between any two code fragments. (Some of the issues are described by Heckel [22]). We are developing a better differencing algorithm that uses interactive syntactic information (provided by the Harmonia framework) to derive differences that more closely correspond to the way humans view duplicated code, with a much faster running time. We are also revising the incremental re-differencing algorithm, and developing a mechanism to allow users to give feedback and fine-tune the types of differences reported by the algorithm.

Apart from differencing, we can also experiment with *when* to invoke differencing engine and link the clones. While the current implementation requires the user to select all clones and click "Link Selections," even this step could be performed automatically, either as a result of the user copying and pasting code or via the output of a third-party clone-finding tool, such as the ones mentioned in Section 1.

If programmers are able to link many clones simultaneously across the breadth of a project, an "overview" window or pane that visualizes all linked clones as the programmer edits one of them would be necessary. We would also like to make our link meta-data resilient to file modifications made by third-party tools. Lastly, we notice that there are often higher-level patterns to clones (like consistent variable renaming) for which Linked Editing may be able to infer and provide automated editing support.

We are working on many of these issues and plan to release a more robust version of Codelink to the public as an open-source software project in the future. This will allow us to get real-world usage data to verify that Linked Editing can scale to real programs, and that programmers would use

it in their real work.

Finally, we are working to extend the general technique of Linked Editing to support documents in non-programming domains, such as spreadsheets, web sites, form letters, graphic charts, and music scores. Although these documents frequently contain duplicated content, their authoring environments provide impoverished or nonexistent abstraction facilities, and are frequently used by non-programmers. We feel that Linked Editing could provide a substantial benefit to these domains.

8. Related work

Some researchers have posited that the problem with duplicated code is that programmers cannot find it in their programs. To this end, a wide array of tools have been developed to detect duplicated code, by both the research community and the industrial and open-source communities (see references in Section 1). However, clone finders do not alleviate any of the costs of abstraction. If abstracting a clone is too costly when a programmer originally creates it, it may be too costly when a clone finder detects it as well. On the other hand, we would like to integrate a clone finder with the Codelink tool in order to automatically or semi-automatically link existing duplicated code in a code-base.

Other research has attacked the duplicated code problem by automating some of the typing necessary to rewrite duplicated code into programming abstractions. For example, XRefactory [42], Eclipse [16], CloRT [4], and Clone Dr. [9] support semi-automated refactorings for some types of duplicated code. However, these tools do not solve any of the costs of *using* abstractions (discussed in Section 3) that can make the resulting abstractions difficult to understand and modify. Furthermore, they do not help programmers with any of the *cognitive* work of creating abstractions, such as architecting a new design concept and designing the abstraction interface, which we believe to be more substantial than the typing they reduce. Indeed, no study has been conducted to show that programmers having such tools would produce fewer clones.

Linked Editing provides a novel variant of *simultaneous editing*, which was first introduced by Lapis [32]. Linked Editing's variant differs from simultaneous editing in Lapis by making links persistent; allowing blocks of code to be edited individually in addition to simultaneously; visualizing the consistencies and inconsistencies between clones; and providing elision of redundant text. Furthermore, Linked Editing's *model* of similarity and edit-propagation is new. Lapis generalizes each edit individually with an artificial intelligence algorithm that is not fully predictable by the user and can make mis-generalization errors. Linked Editing, on the other hand, establishes the correspondences between clones up-front when linking code, displays them to the user as blue regions of text, and guarantees that the blue regions will remain identical after arbitrary simultaneous edits. This predictability allows Linked Editing users to make complicated, sustained modifications and still be able to verify that changes have been generalized to other instances appropriately.

Linked Editing can be considered an example of programming by analogy. Other projects in this area are Graphical Rewrite Rule Analogies [34] and VisualAgentTalk [13], in which programmers can generalize logic to multiple instances *analogically* (e.g. "Cars move on roads like trains move on tracks"), rather than abstractly. Like Linked Editing, this paradigm does not require the specification of an abstract concept to represent similarity, and thus avoids the costs of abstraction of Section 3.

9. Conclusion

We described Linked Editing, a technique that augments a text editor to provide programmers with a lightweight mechanism to read, write, and edit patterns of duplicated code in an abstract way. We implemented a prototype of Linked Editing named Codelink, and compared it to functional abstraction in a user study. The study found that Linked Editing can provide the same benefits as functional abstractions with drastically less work. Most subjects said they would use a tool like Codelink in their real-life work. These results indicate that Linked Editing would be likely to be used in practice by developers, and would be powerful enough to alleviate the issues of duplicated code in many situations.

Software developers continuously deal with reading, writing, and maintaining programs that are infused with duplicated code because functions, macros and other programming abstractions don't adequately support their needs. An improvement to this situation would greatly benefit the state of software development at large.

10. Acknowledgments

We give special thanks to Marat Boshernitsan for guidance with the theory and paper; and Jonathon Jamison and Laura Germine for providing crucial feedback on the design of the Codelink user interface and user study. This research was supported in part by NSF Grant CCR-0098314 and by a research award from Intel Corporation.

References

[1] G. Antoniol, M. D. Penta, E. Merlo, and U. Villano. Analyzing cloning evolution in the linux kernel. *Journal of Information and Software Technology*, 44(13):755–765, 2002.

[2] B. S. Baker. A Program for Identifying Duplicated Code. *Computing Science and Statistics*, 24:49–57, 1992.
[3] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Second Working Conference on Reverse Engineering*, pages 86–95. IEEE, 1995.
[4] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Partial redesign of Java software systems based on clone analysis. In *Proceedings: Sixth Working Conference on Reverse Engineering*, pages 326–336. IEEE, 1999.
[5] I. D. Baxter, A. Yahin, L. M. D. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, pages 368–377. IEEE, 1998.
[6] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.
[7] M. Boshernitsan. Harmonia: A flexible framework for constructing interactive language-based programming tools. Master's thesis, University of California, Berkeley, June 2001. Appears as Computer Science Technical Report CSD-01-1149.
[8] K. W. Church and J. I. Helfman. Dotplot: A program for exploring self-similarity in millions of lines for text and code. *American Statistical Association, Institue for Mathematical Statistics and Interface Foundations of North America*, 2(2):153–174, 1993.
[9] Clone Dr. http://www.semdesigns.com/Products/Clone/index.html.
[10] Clonefinder. http://www.studio501.com/.
[11] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT Press/McGraw Hill, 1990.
[12] Copy-Paste Detector. http://pmd.sourceforge.net/cpd.html.
[13] B. Craig. Behavior combination through analogy. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 270–273. IEEE, 1997.
[14] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In H. Yang and L. White, editors, *Proceedings of the International Conference on Software Maintenance*, pages 109–118. IEEE, 1999.
[15] Duptective. http://c2.com/cgi/wiki?DupTective.
[16] Eclipse. http://www.eclipse.org/.
[17] F. Fioravanti, G. Migliarese, and P. Nesi. Reengineering analysis of object-oriented systems via duplication analysis. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 577–586. IEEE, May 2001.
[18] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
[19] S. Giesecke. Duplication Management Framework. http://sourceforge.net/projects/dupman.
[20] T. Green and A. Blackwell. Cognitive dimensions of information artefacts: a tutorial. *BCS HCI Conference*, http://www.ndirect.co.uk/~thomas.green/workStuff/Papers/, 1998.
[21] Harmonia Project Web Site. `http://harmonia.cs.berkeley.edu`.
[22] P. Heckel. A technique for isolating differences between files. *Communications of the ACM*, 21(4):264–268, 1978.
[23] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. *ACM SIGPLAN Notices*, 25(6):234–245, June 1990.
[24] J. H. Johnson. Substring matching for clone detection and change tracking. In *International Conference on Software Maintenance*, pages 120–126. IEEE, September 1994.
[25] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(6):654–670, 2002.
[26] M. Kim, L. Bergman, T. Lau, and D. Notkin. Ethnographic study of copy and paste programming practices in OOPL. In *International Symposium on Empirical Software Engineering*, 2004.
[27] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1–2):77–108, 1996.
[28] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eigth Working Conference on Reverse Engineering*, pages 301–309. IEEE Computer Society, Oct. 2001.
[29] B. Laguë, D. Proulx, E. M. Merlo, J. Mayrand, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proceedings of the International Conference on Software Maintenance*, pages 314–321. IEEE, 1997.
[30] A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the 16th Conference on Automated Software Engineering*, pages 107–114. IEEE, 2001.
[31] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance*, pages 244–, 1996.
[32] R. C. Miller and B. A. Myers. Interactive simultaneous editing of multiple text regions. In *Proceedings of the USENIX Annual Technical Conference*, pages 161–174, 2001.
[33] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Proceedings of the 8th Symposium on Software Metrics*, pages 87–94. IEEE, 2002.
[34] C. Perrone and A. Repenning. Graphical rewrite rule analogies: Avoiding the inherit or copy & paste reuse dilemma. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 40–47. IEEE, 1998.
[35] Sametool. http://sourceforge.net/projects/sametool.
[36] Simian. http://www.redhillconsulting.com.au/products/simian/.
[37] Simscan. http://www.blue-edge.bg/download.html.
[38] Sun Microsystems. Java Development Kit (JDK). http://java.sun.com/j2se/.
[39] M. Toomim. *Harmonia-Mode User's Guide*, 2002. http://www.cs.berkeley.edu/~harmonia/projects/harmonia-mode/introduction.html.
[40] Unpaste. http://sourceforge.net/projects/unpaste.
[41] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia. Problems creating task-relevant clone detection reference data. In *Proceedings of the 10th Working Conference on Reverse Engineering*, pages 285–295, November 2003.
[42] XRefactory. http://www.xref-tech.com/speller/.