

Abstractions for End-Users

Michael Toomim
Department of Computer Science
University of Washington
toomim@cs.washington.edu

ABSTRACT

Software Engineers use abstractions to make software scalable and avoid inconsistency errors. End-users, however, are abstraction-averse, preventing them from managing large, complex documents. We have been developing an environment-based technique, called Linked Editing, as a lightweight form of abstraction for end-users.

INTRODUCTION

Abstractions are fundamental tools in Software Engineering. They allow software to scale in size by encapsulating re-used concepts, and reduce errors by ensuring consistency across instantiations.

End-users, however, tend to work without abstraction. They use graphical direct manipulation environments—WYSIWYG word processors and web page editors, paint and illustration environments, spreadsheets, 3D CAD environments, music score editors—where they manipulate concrete objects of their interest with incremental actions and immediate visual feedback. Such environments have succeeded with end-users by providing *concrete* interaction models. However, concreteness forfeits the benefits of abstractions: when Direct Manipulation documents contain re-used or duplicated content (e.g. repeated styles) they can be difficult to scale and prone to inconsistencies.

As a result, interface designers have developed a variety of special-purpose abstraction facilities for these authoring environments. Powerpoint provides the concept of an abstract “master slide” that all concrete slides inherit from. Many music sequencers allow the user to specify bars that repeat for multiple measures. Microsoft Word and the W3C’s HTML introduce elaborate style sheet systems. Dreamweaver provides a “template” abstraction to maintain consistent headers, footers, and navigation bars across a website.

Unfortunately, the use of these abstraction features is often problematic. As put by 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.” [5] Abstraction features are difficult to learn, and each authoring environment has unique special-purpose abstraction mechanisms, in-

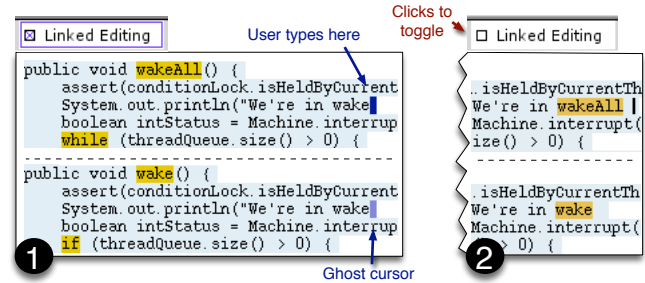


Figure 1: Prototype of Linked Editing for programmers in a text editor

hibiting knowledge transfer. Abstractions are difficult to design and implement: doing so requires much mental effort and planning, and unanticipated changes can require a re-architecture, giving users reason to put off abstraction for fear of premature commitment. Sometimes end-users avoid abstract or indirect interfaces because they find concrete operations easier to predict and safer to trust [2]. On the other hand, designers often constrain the power and applicability of abstractions in an effort to make them more concrete. The Powerpoint master slide, for instance: cannot be parameterized, is global and singular (users cannot create multiple slide styles per presentation), and has a fixed granularity (users cannot abstract content within a slide, nor abstract sets of multiple slides). Thus, the master slide’s applicability as an abstraction mechanism is limited. Content abstractions are far from panacea: they are difficult to learn, constrained in applicability, and place layers of indirection between the user and his or her objects of interest—defeating the original purposes and advantages of Direct Manipulation and concrete WYSIWYG interaction.

As a result, users often work without abstractions. Even expert programmers do so—studies show that the Linux kernel, Java JDK, FreeBSD, MySQL, PostgreSQL, and X Window System are all 20–30% duplicated, and some software is as much as 60% duplicated [6, 8, 9]. But end-users are dramatically more abstraction-averse. For instance, Blackwell found that, in a group of Microsoft Word users, end-users were less than a tenth as likely to create “text style” abstractions in their documents as programmers and scientists, even if they knew how to use that feature of Word [2, 3]. End-users were similarly less likely to create a range of other abstractions, such as nested directory structures, bookmark categories, and telephone quick-dial codes. In another domain, Bellotti reports that designers principally work in terms of concrete repre-

sentational artifacts, rather than abstract concepts, even if abstractions are available [1]. In software, novice programmers are known to create fewer abstractions than experts [4]. Thus, while programmers create far fewer abstractions than would be ideal, end-users and novices create very few at all.

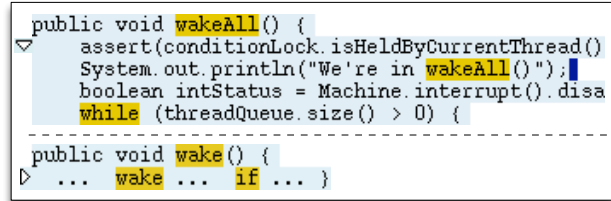
This is unfortunate. If end-users will not abstract away patterns of duplication, they will be unable to author, understand, and modify digital documents beyond a certain size and complexity. An abstractionless user could certainly not extend an abstractionless CNN.com, for example. Nor could such a user easily work in other domains; e.g. authoring a computer-graphic landscape with hundreds of trees, or editing an electronic music score with hundreds of voices. This would be a regretful scenario, since the ideas on CNN.com’s website are not difficult for an end-user to comprehend, edit or express—but rather the structural characteristics of their transcription in a website.

Linked Editing: Abstraction in concrete interfaces

We believe that an intelligent authoring environments can remedy this problem. Linked Editing [9] is a novel technique we are developing for visualizing and editing duplication without explicit abstraction or additional layers of indirection. Our hypotheses are that end-users will prefer its concrete interaction style over abstraction, and that it will let them edit larger, more complex documents than they would otherwise be able to, with fewer errors.

Figure 1 displays the current implementation of Linked Editing, which was developed for programmers, rather than end-users, to help them manage duplicated code. The system automatically finds duplicated code, and highlights common regions in blue and differences in yellow. Then the user can edit all instances at once by editing any single instance (a variant of Simultaneous Editing [7]). If the user wants to edit just a single instance, she toggles the “Linked Editing” mode checkbox on the toolbar before typing, and the system incrementally finds and highlights the new similarities and differences. Linked Editing allows duplication to be edited scalably with Simultaneous Editing. By highlighting similarities and differences, simultaneous edits are predictable (blue regions are guaranteed to be identical after arbitrary edits) and unintended inconsistencies are highlighted in yellow and thus can be avoided.

Linked Editing for web authoring We are now extending Linked Editing to a variety of end-user authoring environments. Here we will illustrate how we envision it assisting an end-user to modify CNN.com in a WYSIWYG web authoring environment. First, the system automatically analyzes the website and finds all patterns of duplication (using a custom algorithm we are developing). Then, as the user moves her cursor over a duplicated block of content, such as a navigation bar, the system provides a visualization of the block’s corresponding copies—miniature depictions of the navigation bars on other CNN web pages. The user can now change all navigation bars simultaneously by simply editing any one. The user can also make changes to any single instance or subset of instances. For example, the user may want to modify the navigation bars on all “science” pages to have additional entry for the “computer science” news category. First, she



```
public void wakeAll() {
    assert(conditionLock.isHeldByCurrentThread())
    System.out.println("We're in wakeAll()");
    boolean intStatus = Machine.interrupt().disa
    while (threadQueue.size() > 0) {
        ...
    }
}

public void wake() {
    ... wake ... if ...
}
```

Figure 2: An elided block of duplication looks similar to a function definition and use

would select one or two science pages from the miniaturized visualization. The system then infers by example that the user is selecting all pages with “science” in their header. It briefly highlights, in orange, the word “science” in each document’s header to indicate the pattern it inferred. The user now simultaneously edits the desired entry into all science navigation bars.

Note that introducing a new type of difference amongst duplicated instances, as was done here, would be much more difficult using a traditional template or function abstraction: the user would have had to add a new template definition, or parameter in the function definition, to represent the new type of difference (science or not science page) as well as modify each use of the function or template to provide the appropriate parameter or select the appropriate template. In general, abstraction systems become more complicated as additional differences are required. The system described here, however, adapts automatically to the concrete content created and infers an implied inheritance hierarchy behind the scenes.

Transitioning to traditional abstractions Figure 2 shows the result of clicking a button to elide the identical portions of block from view, leaving only the differences visible. This is similar to how a function call hides the function’s body and shows only parameters. In the future we also envision allowing the user to specify an optional name for a repeated block of content, and optionally transform the content into a traditional abstraction. By making names, elision, and simultaneous editing optional and independent, the system provides the user with a continuum of incremental abstraction, letting users work concretely or abstractly, at their discretion.

Other examples of duplication There are a variety of duplication situations in which Linked Editing could be useful beyond those already given. For instance, spreadsheet users often copy and paste complex formula between cells, to perform similar calculations. Accountants sometimes create duplicated versions of entire sheets, with minor changes, to analyze “what-if” scenarios. Secretaries periodically compose form letters and want them personalized for some recipients, which is difficult to accomplish with a “database merge” abstraction. Music composers repeat melodies and drum beats across an entire score, but modify them for some measures. Presenters copy graphical diagrams to multiple slides, modify them on certain slides to represent change, and then need to update an aspect of all diagrams at once. Researchers create multiple versions of a user study script for each condition of the experiment, and must be extremely careful to maintain differences (the independent variables) as they copy and

paste and evolve the scripts in parallel.

Preliminary results

We conducted a user study comparing Linked Editing with functional abstraction. Linked Editing took dramatically less time to implement and use, and resulted in code that programmers reported as being easier to understand and change [9]. These results are very encouraging, and we suspect they will be similar for end-users, in non-programming situations.

RELATED WORK

Lapis [7] introduced Simultaneous Editing, but supports one-off interactive edits rather than persistent abstractions, and differs from Linked Editing in other ways as described in [9]. Other projects have implemented demonstrational inference for specific subtypes of duplication (e.g. Tourmaline [10] infers styles in word processing documents) but are not as general as Linked Editing.

CONCLUSION

With or without abstractions, authoring and maintaining large documents is a major challenge in end-user software engineering. By providing abstraction-like scalability benefits without requiring layers of abstract indirection, Linked Editing may be a solution that end-users can benefit from.

REFERENCES

1. Victoria Bellotti, Simon Buckingham Shum, Allan MacLean, and Nick Hammond. Multidisciplinary modelling in hci design...in theory and in practice. In *Proceedings of the conference on Human Factors in Computing Systems*, pages 146–153, 1995.
2. Alan F. Blackwell. See what you need: Helping end-users to build abstractions. *Journal of Visual Languages and Computing*, 12:475–499, 2001.
3. Alan F. Blackwell. Personal Communication, 2004.
4. Francios Detienne. *Software Design – Cognitive Aspects*. Springer, 2002.
5. Thomas Green and Alan Blackwell. Cognitive dimensions of information artefacts: a tutorial. *BCS HCI Conference*, <http://www.ndirect.co.uk/~thomas.green/workStuff/Papers/>, 1998.
6. Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th International Conference on Operating Systems Design and Implementation*, December 2004.
7. Robert C. Miller and Brad A. Myers. Interactive simultaneous editing of multiple text regions. In *Proceedings of the USENIX Annual Technical Conference*, pages 161–174, 2001.
8. Damith C. Rajapakse and Stan Jarzabek. An investigation of cloning in web portals. In *International Conference on Web Engineering, ICWE'05*, pages 252–262. Springer, July 2005.
9. Michael Toomim, Andrew Begel, and Susan L. Graham. Managing duplicated code with linked editing. In *Proceedings of the IEEE Symposium on Human-Centered Computing and Visual Languages (to appear)*. IEEE, 2004.
10. Andrew J. Werth and Brad A. Myers. Tourmaline: Macrostyles by example. In *Proceedings INTER-CHI'93: Human Factors in Computing Systems*, page 532, April 1993.