Exploring Modal Locking in Window Manipulation
Why Programmers Should STASH, DUPLICATE, SPLIT, and LINK Composite Views

Marcel Taeumel
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
marcel.taeumel@hpi.uni-potsdam.de

Robert Hirschfeld
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
robert.hirschfeld@hpi.uni-potsdam.de

ABSTRACT
Window manipulation plays a vital role in multi-tool user interaction, especially for programmers exploring software artifacts, gathering information for better understanding. However, today’s window managers offer only limited means to organize screen contents, which increases cognitive efforts for both tool builders and users. Builders must account for live integration of composite views; users might have to work around disruptive mode errors when actual tasks conflict with a tool’s design. We follow a pattern-finding approach and present four new verbs for direct window manipulation, which we consolidated from existing tools and systems. If window managers would offer to STASH, DUPLICATE, SPLIT, and LINK views, we believe that programmers could better maintain flow during exploration activities.

KEYWORDS
Program comprehension, window management, direct manipulation, tool building, exploration

ACM Reference Format:

1 INTRODUCTION
Think about what you see. Write down a thought. Click on that label to browse. Branch, merge, and backtrack. — While comprehending programs, programmers constantly estimate

Unpublished working draft. Not for distribution.
We assume that programmers will manually interact with tool windows (or any view container) when projecting their thoughts (and level of understanding) into the environment, onto the screen. Consequently, we are not looking for automatic reduction of friction. Instead, we are questioning the current way of interacting with windows, which is unnecessarily indirect and increases cognitive load. We are looking for new gestures that support programmers in directly manipulating windows to immediately mirror updates from their mental model [18, pp. 12–17][11].

In this paper, we propose four new verbs that should support programmers managing screen contents through visual containers: STASH, DUPLICATE, SPLIT, LINK. The environment should log each such interaction and also offer a flexible UNDO and REDO. Following a pattern-finding approach [7], we justify our proposal by presenting known uses in existing systems, each offering some of these verbs. Our approach is obviously inspired by the historical efforts that enabled “modeless” text editing as cut/copy/paste gestures through keyboard shortcuts [10, 23, 28]. We aim at a comparable experience for (manual) window management, which can also guide tool builders to design modular application models that can anticipate, maybe avoid, modal locking.

Note that we present a first draft of what might become a pattern language around window manipulation. Given our focus on composite tools and modal locking, our presentation thus deviates from the usual pattern form [1, 6, 7]. Following this paper’s theme, we combine the descriptions of each verb’s intent, known uses, and consequences side-by-side instead of separated.

In section 2, we clarify our vocabulary and explain how mode issues are different between domain models and application models. Our main contribution lies in section 3, where we describe the four new verbs and which existing systems may serve either model or both, which affects modular decomposition [15, pp. 39–64]. The command history for a domain model (e.g., text undo) is typically closely coupled

2 STATEFUL MODELS IN TOOLS

In this section, we explain our perspective on programming tools being implemented as domain models and applications models, which glue interactive graphics to invaluable data-under-exploration. Figure 2 illustrates an example for modal lock in a code browser.
2.1 “Modeless” Domain Models

Many tools represent their primary domain-artifacts in a way that supports direct manipulation [9]. Think about paragraphs on a text page, pixels in a photograph, lines in a drawing, shapes on a presentation slide, or cells in a spreadsheet. In all these examples, users can click on (and maybe select multiple) such representations to then invoke (reversible) operations (or verbs [21, pp. 59–62]), which usually includes cut/copy and paste to re-structure contents. This level of directness often embraces experimentation through a complementary undo/redo log. We can find comparable safety nets in tools for the programming domain, too. Even if not file-based but still textual, code artifacts are typically under version control. In live-programming systems such as Squeak/Smalltalk, fine-granular logging [24] can facilitate inadvertent recovery, while transactional changes [14] can render the feedback loop more robust. — In any case, there is no modal dependency between such direct edits when domain models shine through the tool’s user interface [20]. The interaction feels “modeless” and directly supports the users train of thought.

By definition, it is not modal locking when the current state of a domain artifact prohibits certain operations. Such circumstances are part of the domain rules (i.e., model) and hence do not impede (or block) but drive the user’s workflow. For example, if there are no red pixels on a drawing canvas, the user has no intention in performing a flood-fill selection on red pixels. Consequently, when users commit changes to the domain model, the tool’s (graphical) interface does not enter a specific mode. In self-sustaining programming systems, tool builders have to bear the risk of breaking the tools they are currently using. Yet, locking yourself out of the tools by accident will trigger a need for recovery, not a mode switch.

2.2 Modal Application Models

Given that domain models represent the current state of domain artifacts, application models manage the rest of a tool’s graphical interface. Think about the bounds of windows, offsets in viewports, slider positions in scrollbars, or selections in lists. Users will constantly modify such stateful, visual composites when interacting through mouse clicks, key strokes, or touch gestures. — Yet, also consider unsaved contents in a text field or the not-yet-committed state of a checkbox. The most recent part of the domain model’s undo/redo log is typically managed by the application model. Only committed changes leave a tool’s realm and are then logged through a shared mechanism so that other tools in the environment can participate. For example, files will only get a new version once the text editor has written new contents, which does usually not happen per typed character.

As exemplified in Figure 2, modal locks disrupt the user’s flow. Modal dialogs try to soften such locks but remain disruptive because they call for often impossible decisions. We claim that no user really wants to discard recent efforts but expects the tool’s ongoing support, obliquely at best. Composite views try to integrate common workflows, yet quickly grow in complexity of what is possible. Especially such browse-and-edit arrangements must account for users starting to edit. How to deal with not-yet-committed contents when starting to browse again? Tool designers have learned that manually re-creating a domain log is hard, which yields those confirmation dialogs or non-disruptive attempts such as new windows (or tabs) popping up. Tool designers have also learned that browsing a deeply structured domain is hard, which makes a navigation history grow per mouse click. Finally, tool designers have accepted that there are breaks between working days, which allows users to shut down the working environment and then come back the next day to find all windows as they were the day before.

But how can tool designers support (incidentally) complex scenarios without bearing the risk of unforeseen modal locks? We think that the environment should give users more power over view compositions, which entails a more direct way of integrating (or separating) application models. Inspired by the “modeless” interaction with domain models through (undo-able) cut/copy-and-paste gestures, we want to encourage users to reduce friction losses by manually opening up modal locks as they occur — without discarding unsaved work. Most importantly, they should be able to separate (or integrate) browsers and editors. Once part of the tool’s interface, users can non-disruptively explicate their intents about exploration and note taking.

3 DIRECT WINDOW MANIPULATION

In this section, we propose a more direct way for manipulating composite views (or tool windows) so that users can resolve modal locks themselves. We assume that programmers are prevalent in our target audience since manual re-composition might require a willingness for experimenting with one’s work efficiency. And tool-constructing programmers would enable our proposal in the first place.

First, we will follow the concept of noun-verb interaction [21, pp. 59–62] and present four new verbs for window manipulation, including their relation to the established ones: OPEN, CLOSE, COLLAPSE, and EXPAND. Second, we will describe...
which of our proposed verbs existing systems already offer — some of them successfully for decades.

3.1 The New Verbs

Basically all of the following operations should be logged and easily reversible. While this requirement would likely increase the environment’s resource consumption, it would make user actions feel forgiving and hence encourage experimentation. The new verbs are as follows; we list alternative names to establish a possible relation to familiar concepts:

**Stash.** De-prioritize contents in a view and remove the visual container to make room on screen. Unsaved changes are not deleted but out of the user’s sight. Reverting this operation will make the container reappear as is, including view-specific state such as text selection or scroll position. Alternative names could be close, hide, collapse, or discard.

**Duplicate.** Branch the exploration path to follow up on a new idea by getting two identical representations of the same scenario on screen. Further interaction with the copy will make its visual appearance diverge. Reverting this operation will simply stash the “duplicate” in its most recent form. Alternative names could be copy, multiply, or mirror.

**Split.** Re-use a window’s component in a different context by visually and semantically separating it from its current neighbors. Users can then link it to another component or stash and duplicate at a more fine-granular level. Reverting this operation will simply link the component again. Alternative names could be cut, partition, disconnect, or divide.

**Link.** Re-use (an already split) component in a different context by visually and semantically integrating it into its neighbor windows. Users can populate existing views with new data coming from freshly linked exploration paths. Alternative names could be paste, glue, join, or connect.

Tools and their views would open as usual through button clicks or keyboard shortcuts. Users would naturally resize (or expand) views to accommodate the available screen space among other tools. Now, making room on the screen would be different: users could choose to (a) stash a tool’s entire view composition or (b) split it up first to only hide obsolescent parts to not inadvertently increase cognitive load. Since these operations should be reversible, there would be no confirmation dialog blocking the user’s flow when they would hit a close button. They already know this kind of safety net from hitting a collapse (or minimize) button. Thus, we think they could easily adapt to this new behavior.

The verbs split and link challenge the degree of modularity [15, pp. 39–64] in the affected application models’

![Figure 3: Squeak (top) can duplicate all kinds of morphs, including tool windows and models. Sublime Text (middle) does implicitly link and split between file list and file contents. Vive (bottom) offers an explicit way for view composition.](image-url)
found a possible architecture for purely object-oriented sys-
tems (such as Squeak/Smalltalk in the form of VivIDE [25, 27]): a data-driven, script-based, interactive tool-construction
framework. There might be other approaches to realize the
verbs SPLIT and LINK.

3.2 Known Uses
While we think that our proposal for direct window manip-
ulation is novel in its packaging, we did derive those verbs
from existing systems. Following the design-patterns com-
munity [7], we thus summarize our original observations (as
depicted in Figure 3) to further substantiate feasibility and
applicability.

In virtually any window manager, users can (kind of)
stash visual containers. There is usually a COLLAPSE (or MINI-
MIZE) operation for overlapping layouts or a STACK operation
for tile-based layouts. Both variants then offer (button-like)
tabs in a row so that users can easily retrieve the hidden con-
ten. As an example for STACK, there is a sketch in Figure 1,
complemented through browse-and-edit combinations in
Pharo (Figure 2) and Sublime Text (Figure 3). As an example
for COLLAPSE, the window managers in Squeak and Pharo
each behave as generalized, while Sublime Text depends on
the operating system’s manager. — Yet, we argue that the
typical close button does not invoke a STASH operation be-
cause it is not reversible, which makes it disruptive. We think
that there is no need for such manual “garbage collection,”
which we discuss later in this paper.

A DUPLICATE operation for visuals has been around since
the conception of Self’s Morphic [13] and its adaptation for
Squeak [12, 26]. Every morph offers this verb in a meta-menu,
called halo, as depicted in Figure 3. In their simplest form,
morphs are like shapes in presentation tools (such as Power-
Point). Yet, morphs are not just structural composites, they
can exhibit any complex behavior. Being integrated in the
Smalltalk system, morphs can represent any kind of interac-
tive view (or window). Unsurprisingly, users can DUPLICATE
not only visual structure but also a tool’s behavior-driving
configuration: the application model. Like users cut/copy-
and-paste shapes on presentation slides, programmers in
Squeak naturally duplicate windows to handle modal locks
in browse-and-edit tools (Figure 2).

We think that, in most cases, when a window does SPLIT
or LINK its components, users do not notice — and should
not care by design. As exemplified through the Sublime Text
editor in Figure 3, users can browse file contents in the same
container until they start to edit. Any existing container in
EDIT MODE cannot be re-purposed for browsing (mode) but
only closed. While this might be okay for simple widgets
such as text fields, we argue that more configurable views
deserve longer lifetimes. In VivIDE, we experimented with
explicitly connecting and disconnecting views as means for
switching between exploration paths. In many cases, the end
of a path was represented through one of many selectable
and configurable views. For new paths, closing those views
would mean that programmers must repeat redundant steps
because the DEFAULT view may not be supportive. — Conse-
sequently, we think that an environment with many different
views should offer the verbs SPLIT and LINK as part of basic
user-to-window interaction.

4 DISCUSSION ABOUT CONSEQUENCES
Reversible close operations are typically implemented by
logging the identifier for a particular domain artifact, which
excludes unsaved changes. For example in many text editors,
users can retrieve efforts through a list of recently-opened
file paths. In Web browsers, for another example, users can
undo closing a tab, but the Web page’s contents will be re-
loaded from the URL, which represents already committed
domain data. In such sophisticated browsers for structured
information, successively touched artifact identifiers are of-
ten logged into a navigation history. The environment might
keep that history across multiple sessions. Yet, users can
still not mix note taking and wanting to STASH windows
with uncommitted changes: a disruptive dialog will ask for
discarding those changes. They must plan ahead.

However, extensive logging would raise concerns about re-
source consumption and data privacy. In Squeak, it is rather
easy to hold on to the objects that represent application mod-
els. Given that there is memory paging and enough disk
space, a strategy such as least-recently-used could not only
de-emphasize visuals [22] but also clean up hidden, tool-
specific “trash bins” automatically. After the right amount of
time, users may just have moved on and saved the changes
they actually wanted to commit. — But maybe users get anx-
iou about unfinished thoughts sitting around somewhere
in a cache, waiting to be exposed. If the click on a close
button would not immediately discard a text buffer’s confidential
contents, users might require complementary tools like there
is “securely delete” for files.

Given that we advocate the DUPLICATE verb, we want to
finish this discussion with a brief (hypothetical) look on clut-
erred screens. Programmers often need more screen space
than they have available to lay out all relevant information.
Overlapping windows can be annoying to constantly re-arrange; a close-all button may be no cost-effective alter-
native. A layout strategy with TILES looks neat and tidy, but
often hides too much information in stacks. We see (virtually)
endless tapes [2, 4, 8] as a good fit for onward exploration: the

\[2\] Note that cookies might configure page loading with (transient) informa-
tion from the current session.
path can deepen horizontally and branch vertically. Yet, a full-screen viewport has no visually stable content (i.e., pixels) when scrolling (or zooming) back-and-forth. A combination of overlapping tapes could be a step forward. Programmers could flexibly manage multiple exploration paths.

5 CONCLUSIONS

We explained our perspective on multi-tool environments as a plentitude of domain models and application models, which glue interactive graphics to data-under-exploration. We argued that tool users should be empowered by manipulating windows (or any visual container) through the (often novel) operations STASH, DUPLICATE, SPLIT, and LINK to work around occasional, yet disruptive, modal locks. We think that tool builders cannot anticipate all possible, complex scenarios for integration and should thus follow a more generic approach. Leave it to the users, which are programmers in our case, because they are probably immersed into their task and do not expect to be patronized through superfluous interruptions. Instead, make window management more open and configurable — yet safe — so that users can learn to efficiently accommodate domain-specific tasks.

We did not stress our analogy to cut/copy-and-paste gestures. There must be some kind of "trash bin" for stashed windows. But should there be a "clipboard" for components that were copied or cut out? From personal experience [25], we know that flexible view composition can feel direct and non-disruptive by just doing a click-and-drag, completed with a drop. Yet, with fidgeting fingers on a cluttered screen, a simple cut-and-hide might be more user friendly. Thus, we see more potential for investigating better interaction paradigms for window-based exploration environments.

TOOLS AND SYSTEMS

In this paper, we presented arguments and screenshots based on the following tools and systems, which we all accessed on 2021-02-10:

- Squeak 5.3 (https://www.squeak.org/)
- Pharo 8.0 (https://www.pharo.org/)
- Sublime Text 3.1.1 (https://www.sublimetext.com/)
- Vivide 2021-02-10 (https://github.com/hpi-swa/vivide/)

ACKNOWLEDGMENTS

Sincere thanks go to all PX workshop participants, who provided valuable feedback by discussing this topic thoroughly. We gratefully acknowledge the financial support of the HPI Research School "Service-oriented Systems Engineering" (https://hpi.de/en/research-schools/hpi-sse.html) and the Hasso Plattner Design Thinking Research Program (https://hpi.de/en/dtrp/).

REFERENCES


Exploring Modal Locking in Window Manipulation

265–284.


