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.

CCS CONCEPTS

• **Software and its engineering** → **Integrated and visual development environments**; *Patterns*; *Object oriented development*; • **Human-centered computing** → *Interface design prototyping*.

KEYWORDS

Program comprehension, window management, direct manipulation, tool building, exploration, interface design

ACM Reference Format:
Marcel Taeumel and Robert Hirschfeld. 2021. Exploring Modal Locking in Window Manipulation: Why Programmers Should Stash, Duplicate, Split, and Link Composite Views. In *Companion Proceedings of the 5th International Conference on the Art, Science, and Engineering of Programming (<Programming> '21 Companion), March 22–26, 2021, Virtual, UK*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3464432.3464433

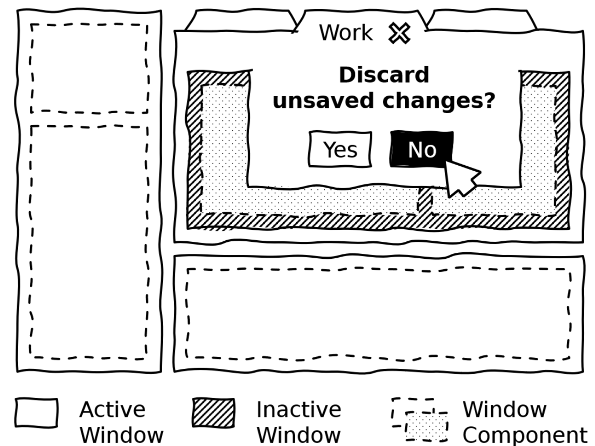

Figure 1: A good start. Modal (confirmation) dialog, blocking only one of the environment's available (tool) windows. But how to avoid modal locks among window components?

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 the cost of switching between exploration paths. They have to guess which paths lead to valuable information, and which ones are a waste of time. The theory on information foraging [6, 22] suggests that programming tools are especially helpful when they offer *cues* that help making such decisions. Looking at the entire programming environment, there is a combination of tools (and tool windows) forming that exploration path, offering tempting cues, hosting informational *prey*. And every tool switch will be costly if it interrupts the current task [21, 24] with new information, new visuals, or new ideas. But what about exploring *familiar* things such as when backtracking or assessing the current goal's overall progress? That *must be* cheap; it is a matter of effective window management.

Unfortunately, today's window managers provoke superfluous decision-making, which interrupts the programmer's flow of exploration [4]. As depicted in Figure 1, even for easily reversible (or "undo-able") gestures such as closing a view, environments expect users to confirm the loss of unsaved work. While there usually is a "trash bin" (or command history) for *domain models*, uncommitted changes in *application models* have no such safety net. For example, users can easily revert changes in a text file (or paragraph), but trying to close a document's view may be rewarded with a blocking "Are you sure? There is no going back." This conceptual nuisance can propagate into a window's components, where *browser views* are locked until the programmer finishes (i.e., commits!) the current edits. That is, exploration tools can be *modally locked* [26, pp. 37–47] as soon as programmers start writing (unpolished) thoughts into a nearby text field. Such modal locks negatively affect costs and cost estimation in the underlying exploration activity.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

<Programming> '21 Companion, March 22–26, 2021, Virtual, UK

© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-8986-0/21/03…$15.00
https://doi.org/10.1145/3464432.3464433

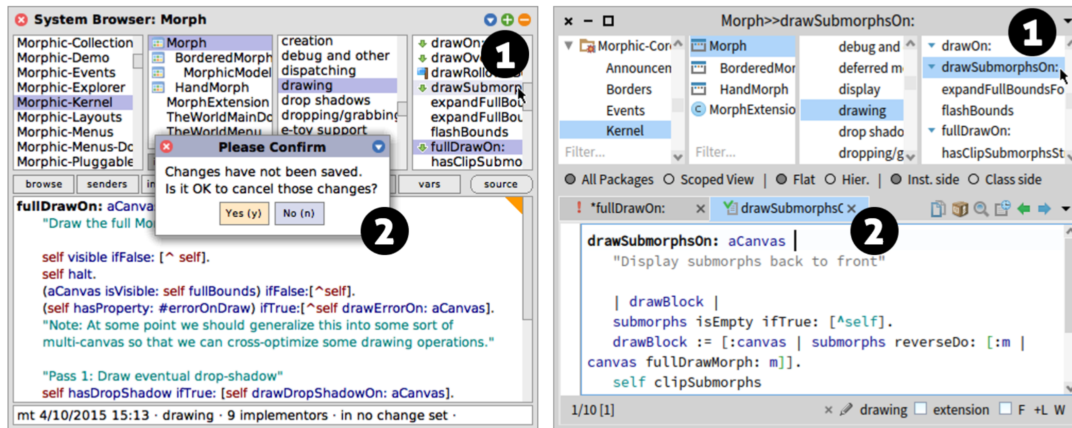

Figure 2: Modal lock in a code browser. (1) After adding (not saving) "`self halt`" to the method `fullDrawOn:` and clicking on `drawSubmorphsOn:`, (2) Squeak 5.3 (left) requests confirmation while Pharo 8.0 (right) does not block; it embeds a new tab. Programmers in Squeak get used to duplicating the window first to not discard work.

Given that programmers cannot decide on *committing or discarding* informational prey while still gathering:

> *How can the environment's window management support a programmer's train of thought during exploration?*

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* [11] windows to immediately mirror updates from their mental model [23, pp. 12–17][16].

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 to also offer a flexible UNDO and REDO. Following a pattern-finding approach [8], 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 [15, 28, 33]. 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, 7, 8]. Following this paper's theme, we combine the descriptions of all verbs' intents, known uses, and consequences side-by-side instead of separated per verb.*

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 already implement some of them. There are related attempts only addressing the domain model, which we discuss in section 4, including concerns on extensive logging and visual clutter. We conclude our thoughts in section 5.

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 locking in an hierarchical code browser.

Note that a tool's technical artifacts (i.e., classes/methods) may serve either model or both, which affects modular decomposition [20, pp. 39–64]. The command history for a domain model (e.g., text undo) is typically closely coupled to the view (e.g., text field) and thus the application model, which also holds other state (e.g., text selection).

We think that distinguishing between two (kinds of) models does *not* indicate that language constraints or implementation details are leaking upwards. Instead, a program's design should always account for (1) a suitable representation of domain artifacts in terms of a language's primitives and (2) a configurable mapping of artifact properties in terms of a system's (or application's) interactive visuals. At best, this conceptual dualism would be realized by clearly separating the roles as distinguished technical artifacts. There are simplifications of this dualism, such as by limiting (almost hard-coding) the application model [25], but the underlying design challenge remains.

2.1 "Modeless" Domain Models

Many tools represent their primary domain-artifacts in a way that supports direct manipulation [11]. 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 [26, 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 [2], fine-granular logging [29] can facilitate inadvertent recovery, while transactional changes [19] 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 [25] as distinguished views representing particular domain artifacts. The interaction feels "modeless" and directly supports the user's 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 (most likely) 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 switch between built-in modes.

Note that there is often more than one view per domain artifact, each varying in directness, conciseness, expressiveness, or configurability. It is the responsibility of application models to provide the means for selecting, configuring, and interacting with such views. With that role comes one of the major challenges around interface modes and modal locks.

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 list views. 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.[1]

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, oblivious support. Composite views try to integrate common workflows, yet quickly grow in complexity of what is possible.

[1] Auto-save can mitigate this issue if (1) users will not be noticeably interrupted and (2) artifact changes will never trigger unwanted side effects, both of which are open challenges in Squeak/Smalltalk.

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?

Looking at existing systems, we realized that there are already *best practices* for tool designers to accommodate more complex multi-tool scenarios. For example, manually tracing and repeating one's interactions can be hard and annoying, which yields – in critical places – confirmation dialogs or *non-disruptive attempts* such as new windows (or tabs) popping up. Candid experimentation embraces failure; loss of work would be demoralizing. For another example, browsing a deeply structured domain can be time-consuming, which makes a *navigation history* grow as users explore a path. Finally, there are tool designs that acknowledge breaks between working days, which allows users to shutdown 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 — not having to discard unsaved work, not even be asked about it. Most importantly, they should be able to separate (or integrate) browsers and editors. Once part of the tool's interface, users could 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* [26, 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 obsolete 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 [20, pp. 39–64] in the affected application models' design. Tool builders would have to establish a *direct mapping* between domain artifacts and visual representations (like Naked Objects [25]). When users would try to integrate or separate nearby windows, the underlying mechanics (or technical artifacts) would need to follow clearly defined rules of *composition* and *decomposition*. We already found a possible architecture for purely object-oriented systems (such as Squeak/Smalltalk) in the form of Vivide [30, 32], which is 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 manipulation is novel in its packaging, we did derive those verbs from existing systems. Following the design-patterns community [8], 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 MINIMIZE) 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 content. As an example for STACK, there

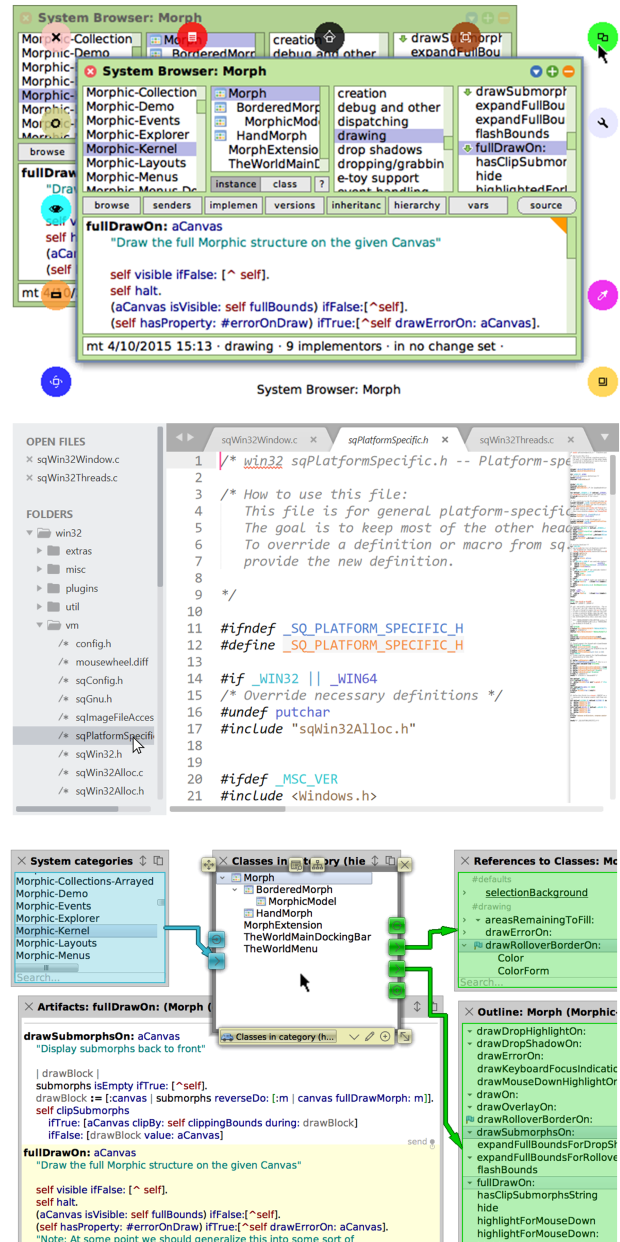

Figure 3: Squeak (top) can DUPLICATE all kinds of morphs, including tool windows and models. Sublime Text (middle) does implicitly LINK and SPLIT between file list and file contents. Vivide (bottom) offers an explicit way for view (de-)composition.

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 because 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 [18] and its adaptation for Squeak [17, 31]. 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 PowerPoint). Yet, morphs are not just structural composites, they can exhibit any complex behavior. Being integrated in a Smalltalk system [9, 13, 14], morphs can represent any kind of interactive view (or window). Unsurprisingly, users can DUPLICATE not only visual structure but also a tool's behavior-driving configuration, which corresponds to a *deep copy* of 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[2] 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 effortfully configured views deserve longer lifetimes. Programmers might want to try out those views with different kinds of data. 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. Consequently, 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

In this section, we discuss known workarounds for *missing* verbs, current struggles with *naming* the verbs, and potential consequences when *realizing* all verbs in a single environment.

4.1 Anti-patterns and Workarounds

Through Figure 2 and Figure 3, we illustrate that programmers in Squeak can work around the missing SPLIT (and confirmation dialogs) by duplicating the windows of interest. In general, many environments somehow "compensate" for "missing" verbs, which typically interrupts the programmer's train of thought and hence increases the cognitive load during exploration. Especially the absence of "undo" demands mental simulation of critical steps.

If there is no STASH operation, users can often MINIMIZE, COLLAPSE, or STACK windows (or views) to make room for other information. If that is still too effortful, a layout of overlapping windows is typically forgiving when users just leave it be, letting tools fade into the background, slowly forgetting about them.

[2]Many file-oriented, text-based programming environments offer a similar browse-and-edit interface, such as Eclipse, IntelliJ IDEA, and Visual Studio.

If there is no DUPLICATE operation, users can often manually re-open and re-configure tools until their appearance matches the original. Yet, multiple views per domain artifact require non-exclusive access to domain artifacts and a way to communicate changes triggered from outside a particular view. While many (read-only) exploration tools anticipate such outside changes, two loggers that, for example, are appending contents to the same file would be in conflict.

If there is no SPLIT or LINK operation, users can choose to ignore or re-purpose parts of a tool's composite views. Yet, having multiple instances of the same tool with redundant information is a waste of screen space. Manually "linking" information between tools typically involves copy-and-paste (text) or drag-and-drop (objects), which can also be time-consuming and error-prone.

4.2 Naming the Verbs

When documenting best practices in pattern form, a pattern's name serves as a concrete handle to refer to and talk about with field experts. Unfortunately, finding good names for our verbs can be challenging. Short names typically have an established meaning and thus might entail inappropriate co-notations. Longer names (or even descriptions) might not be suitable as reference point. Since the pattern language in this paper is considered a "draft," we are still looking for good names.

Looking at STASH (and its alternatives), we are concerned about negative co-notations and a conflict of meaning with well-known operations. First, using *stashed changes* in version control (e.g., Git) might too often lead to a situation where almost forgotten work cannot be easily applied to a project's current form. Also negative, the alternative DISCARD might be associated with the loss of work. Second, the common operations CLOSE and COLLAPSE are quite similar to what STASH intents to cover. Re-using those names with extended meaning might be elegant yet confusing; expecting users to learn that "STASH also means CLOSE" might be too troublesome to accept.

Looking at DUPLICATE, our proposed alternatives might not fit the domain or scope of window manipulation. First, MULTIPLY is a math operator and not associated with making copies. Second, a MIRROR might be expected to turn things upside down. Finally, the effect of DUPLICATE or COPY is expected to fade quickly as users directly manipulate the result when continuing exploration. Maybe there are names that better reflect the timeliness and intent of not having to repeat prior steps to revise certain choices. Squeak uses the name SPAWN to extract changes of one tool into a new one, cleaning up the original one.

Looking at SPLIT, there are tools that support *duplicating* a window to gain two independent viewports, effectively *splitting* the original screen space. For example, Emacs text buffers or file views in other text-based environments can be split this way to look at multiple (distant) sections that would otherwise not fit on the screen. Our intention of splitting is more like a CUT through a view composition to re-purpose sub-views.

Looking at LINK and SPLIT, we wonder whether all verbs around window manipulation should occur in opposing pairs. Our first impression was, that a generic UNDO should be the opposite of any operation. Yet, every CLOSE had its OPEN. And what about a DE-DUPLICATE to clean up a messy desktop without losing work? For

manipulating view compositions, we collected many alternatives in pairs: ATTACH/DETACH, COUPLE/DECOUPLE, LINK/UNLINK, and INTEGRATE/EXTRACT.

4.3 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.[3] In such sophisticated browsers for structured information, successively touched artifact identifiers are often 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, because a disruptive dialog will ask for discarding those changes. They must plan ahead.

However, extensive logging would raise concerns about resource consumption and data privacy. In Squeak, it is rather easy to hold on to the objects that represent application models. Given that there is memory paging and enough disk space, a strategy such as least-recently-used could not only de-emphasize visuals [27] 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 anxious 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 cluttered 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 might be too careless and potentially costly. A layout strategy with *tiles* (e.g., Eclipse, IntelliJ IDEA, Visual Studio) looks neat and tidy, but often hides too much information in stacks. We see (virtually) endless tapes [3, 5, 10] as a good fit for onward exploration; the 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 then flexibly manage multiple exploration paths.

5 CONCLUSIONS

We explained our perspective on multi-tool environments as a plenitude 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 [30], 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.

Overall, we see more potential for investigating better interaction paradigms for window-based exploration environments. Regarding this "draft" of a pattern language, a valuable next step would be to validate our proposed verbs in a user study to let field experts check their personal experience [12]. Then, each pattern's name, intent, known uses, consequences, and relation to other patterns could be clarified.

[3]Note that cookies might configure page loading with (transient) information from the current session.

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/21 reviewers and 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

[1] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. 1977. *A Pattern Language - Towns, Buildings, Construction.* Oxford University Press. ISBN 978-0-19-501919-3.
[2] Oscar Nierstrasz Damien Pollet Damien Cassou Marcus Denker Christoph Thiede Andrew Black, Stéphane Ducasse and Patrick Rein. 2020. *Squeak by Example* (5.3 ed.). lulu. ISBN 978-1-716-26297-5.
[3] Andrei Chiş, Oscar Nierstrasz, Aliaksei Syrel, and Tudor Gîrba. 2015. The Moldable Inspector. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* (Pittsburgh, PA, USA). ACM, 44–60. https://doi.org/10.1145/2814228.2814234
[4] Mihaly Csikszentmihalyi. 2008. *Flow: The Psychology of Optimal Experience.* Harper Perennial Modern Classics. ISBN 978-0-06-133920-2.
[5] Robert DeLine, Andrew Bragdon, Kael Rowan, Jens Jacobsen, and Steven P. Reiss. 2012. Debugger Canvas: Industrial Experience with the Code Bubbles Paradigm. In *2012 34th International Conference on Software Engineering (ICSE)* (Zurich, Switzerland). IEEE, 1064–1073. https://doi.org/10.1109/ICSE.2012.6227113
[6] Scott D. Fleming, Chris Scaffidi, David Piorkowski, Margaret Burnett, Rachel Bellamy, Joseph Lawrance, and Irwin Kwan. 2013. An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, 2 (3 2013), 14:1–14:41. https://doi.org/10.1145/2430545.2430551

[7] Richard P. Gabriel. 1996. Repetition, Generativity, and Patterns. In *Pattern Languages of Program Design 2*. Addison-Wesley, ix–xiii. ISBN 978-0-201-89527-8.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Abstraction of Reusable Object-oriented Software*. Addison-Wesley. ISBN 978-0-201-63361-0.

[9] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley. ISBN 978-0-201-11371-6.

[10] Austin Z. Henley and Scott D. Fleming. 2014. The Patchworks Code Editor: Toward Faster Navigation with Less Code Arranging and Fewer Navigation Mistakes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Toronto, ON, Canada). ACM, 2511–2520. https://doi.org/10.1145/2556288.2557073

[11] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. 1985. Direct Manipulation Interfaces. *Human-Computer Interaction* 1, 4 (12 1985), 311–338. https://doi.org/10.1207/s15327051hci0104_2

[12] Takashi Iba and Taichi Isaku. 2016. A Pattern Language for Creating Pattern Languages: 364 Patterns for Pattern Mining, Writing, and Symbolizing. In *Proceedings of the 23rd Conference on Pattern Languages of Programs (PLoP)*. 1–63. https://doi.org/10.5555/3158161.3158175

[13] Daniel H. H. Ingalls. 2020. The Evolution of Smalltalk: From Smalltalk-72 Through Squeak. In *Proceedings of the 4th ACM SIGLAN History of Programming Languages Conference (HOPL IV)*. ACM, 1–101. https://doi.org/10.1145/3386335

[14] Daniel H. H. Ingalls, Ted Kaehler, John H. Maloney, Scott Wallace, and Alan C. Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (Atlanta, GA, USA). ACM, 318–326. https://doi.org/10.1145/263700.263754

[15] Jeff Johnson, Teresa L. Roberts, William Verplank, David C. Smith, Charles H. Irby, Marian Beard, and Kevin Mackey. 1989. The Xerox Star: A Retrospective. *IEEE Computer* 22, 9 (9 1989), 11–26. https://doi.org/10.1109/2.35211

[16] Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining Mental Models: A Study of Developer Work Habits. In *Proceedings of the 28th International Conference on Software Engineering* (Shanhai, China). ACM, 492–501. https://doi.org/10.1145/1134285.1134355

[17] John H. Maloney. 2002. *An Introduction to Morphic: The Squeak User Interface Framework*. Prentice Hall, Chapter 2, 39–67. ISBN 978-0-13-028091-6.

[18] John H. Maloney and Randall B. Smith. 1995. Directness and Liveness in the Morphic User Interface Construction Environment. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology* (Pittsburgh, PA, USA). ACM, 21–28. https://doi.org/10.1145/215585.215636

[19] Toni Mattis, Patrick Rein, and Robert Hirschfeld. 2017. Edit Transactions: Dynamically Scoped Change Sets for Controlled Updates in Live Programming. *The Art, Science, and Engineering of Programming* 1, 2 (4 2017), 13–1–13–32. https://doi.org/10.22152/programming-journal.org/2017/1/13

[20] Bertrand Meyer. 1998. *Object-oriented Software Construction* (2 ed.). Prentice Hall. ISBN 978-0-13-629155-8.

[21] Yoshiro Miyata and Donald A. Norman. 1986. Psychological Issues in Support of Multiple Activities. In *User Centered System Design: New Perspectives on Human-Computer Interaction*, Donald A. Norman and Stephen W. Draper (Eds.). Lawrence Erlbaum Associates, Inc., 265–284. ISBN 978-0-89859-872-8.

[22] Tahmid Nabi, Kyle M. D. Sweeney, Sam Lichlyter, David Piorkowski, Chris Scaffidi, Margaret Burnett, and Scott D. Fleming. 2016. Putting information foraging theory to work: Community-based design patterns for programming tools. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 129–133. https://doi.org/10.1109/VLHCC.2016.7739675

[23] Donald A. Norman. 2002. *The Design of Everyday Things*. Basic Books. ISBN 978-0-465-06710-7.

[24] Chris Parnin and Spencer Rugaber. 2012. Programmer Information Needs After Memory Failure. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)* (Passau, Germany). IEEE, 123–132. https://doi.org/10.1109/ICPC.2012.6240479

[25] Richard Pawson and Robert Matthews. 2002. *Naked Objects*. John Wiley & Sons, Ltd. ISBN 978-0-470-84420-5.

[26] Jef Raskin. 2000. *The Humane Interface: New Directions for Designing Interactive Systems*. Addison-Wesley. ISBN 978-0-201-37937-2.

[27] David Röthlisberger, Oscar Nierstrasz, and Stéphane Ducasse. 2009. Autumn Leaves: Curing the Window Plague in IDEs. In *2009 16th Working Conference on Reverse Engineering* (Lille, France). IEEE, 237–246. https://doi.org/10.1109/WCRE.2009.18

[28] David C. Smith, Charles Irby, Ralph Kimball, Bill Verplank, and Erik Harslem. 1982. Designing the Star User Interface. *BYTE - The Small Systems Journal (ISSN 0360-5280)* 7, 4 (4 1982), 242–282.

[29] Bastian Steinert, Damien Cassou, and Robert Hirschfeld. 2012. CoExist: Overcoming Aversion to Change. In *Proceedings of the 8th Symposium on Dynamic Languages* (Tuscon, AZ, USA). ACM, 107–118. https://doi.org/10.1145/2480360.2384591

[30] Marcel Taeumel. 2020. *Data-driven Tool Construction in Exploratory Programming Environments*. Ph.D. Dissertation. University of Potsdam, Digital Engineering Faculty, Hasso Plattner Institute. https://doi.org/10.25932/publishup-44428

[31] Marcel Taeumel and Robert Hirschfeld. 2016. Evolving User Interfaces From Within Self-supporting Programming Environments: Exploring the Project Concept of Squeak/Smalltalk to Bootstrap UIs. In *Proceedings of the Programming Experience 2016 (PX/16) Workshop* (Rome, Italy). ACM, 43–59. https://doi.org/10.1145/2984380.2984386

[32] Marcel Taeumel, Michael Perscheid, Bastian Steinert, Jens Lincke, and Robert Hirschfeld. 2014. Interleaving of Modification and Use in Data-driven Tool Development. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Portland, Oregon, USA) *(Onward! 2014)*. ACM, New York, NY, USA, 185–200. https://doi.org/10.1145/2661136.2661150 ISBN 978-1-4503-3210-1.

[33] Larry Tesler. 1983. Object-oriented User Interfaces and Object-oriented Languages (Keynote Address). In *Proceedings of the 1983 ACM SIGSMALL Symposium on Personal and Small Computers* (San Diego, CA, USA). ACM, 3–5. https://doi.org/10.1145/800219.806644