# One Way to Select Many*

**Jaakko Järvi**[1] **and Sean Parent**[2]

1 **Texas A&M University**
**College Station, TX, USA**
`jarvi@cse.tamu.edu`

2 **Adobe Systems Inc.**
**San Jose, CA, USA**
`sparent@adobe.com`

## Abstract

Selecting items from a collection is one of the most common tasks users perform with graphical user interfaces. Practically every application supports this task with a selection feature different from that of any other application. Defects are common, especially in manipulating selections of non-adjacent elements, and flexible selection features are often missing when they would clearly be useful. As a consequence, user effort is wasted. The loss of productivity is experienced in small doses, but all computer users are impacted. The undesirable state of support for multi-element selection prevails because the same selection features are redesigned and reimplemented repeatedly. This article seeks to establish common abstractions for multi-selection. It gives generic but precise meanings to selection operations and makes multi-selection reusable; a JavaScript implementation is described. Application vendors benefit because of reduced development effort. Users benefit because correct and consistent multi-selection becomes available in more contexts.

**1998 ACM Subject Classification** D.2.11 Software Architectures: Domain-specific architectures; D.2.13 Reusable Software: Reusable libraries

**Keywords and phrases** User interfaces, Multi-selection, JavaScript

**Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2016.14

**Supplementary Material** ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.2.1.7`

## 1 Introduction

Many, perhaps most, interactive software applications present their users one or more collections of elements in the form of lists, trees, grids, or otherwise arranged views, of which a user can select one or more elements. Examples include selecting files and folders in a file explorer; mail folders or mail messages in a mail client; music tracks in a media player; thumbnail images in a photograph organizer; "to do" list items, hours, days, weeks, or months in a calendar application; pages organized into "tabs" in a web browser; and electronic books or videos on a digital library or store. These tasks are typical daily activities for many computer users—we select elements from collections dozens of times per day.

Regardless of which set of modern applications a user chooses for mail, music, photos, calendar, web browsing, books, and videos, the features for selecting elements are likely to differ across applications—even within a single application the selection features for different collections, such as the list of mail folders and list of mail messages, are likely to be different.

* This work was supported in part by NSF grants CCF-0845861 and CCF-1320092.

The differences could presumably stem from optimizing the feature for the best possible user experience in different kinds of selection contexts, but this is not the case. The selection features of modern, widely used applications are different in quite arbitrary ways and the results they produce can be unexpected; Section 2 presents examples. A more plausible reason for the variability is that the software industry has not managed to establish precise standard semantics for selecting multiple elements from collections and, consequently, to produce easily reused implementations of selection features. Each development team therefore designs and implements selection features anew. This repeated design and development effort is a strain on development resources, and a cause for what we have today:

- At a cursory level, most applications offer similar multi-element selection capabilities (*multi-selection* for short). The detailed semantics, however, are "artifacts of the implementation"; instead of being designed for the best user experience, they are what falls out of particular implementation choices.
- User interfaces have many contexts where multi-selection would be useful but it is not provided because of the considerable development effort that would be required to do so. To a user there is no good reason why multi-selection cannot be used with browser tabs, applications in the OS X dock, or icons in the "flyout folders" that open from the dock.
- Even when multi-selection is provided, the feature is incomplete. We draw attention to one particular deficiency. Selections are constructed and modified with key-commands and mouse-clicks that are issued almost reflexively. They can grow to become complex objects, but remain brittle. For example, constructing a selection of thumbnail photos in a photo organizer can take minutes—yet at all times the selection is just one mis-click away from vanishing. Almost no applications provide any kind of undo facility for selections.

Many users may consider a less than ideal selection feature a small problem since the inconvenience experienced at any one selection task is usually minor. Due to the omnipresence of selection tasks in all computer use, however, the aggregate effect is not small. Lack of familiar and predictable multi-selection semantics can cause confusion and steer users towards more repetitive and perhaps more frustrating ways to perform tasks, such as repeating "single-element selection followed by a command" interactions multiple times. Frustrating experiences with computer use impact individuals, organizations, and societies negatively [10].

To better the user experience of future software applications, we put forth a proposal for a set of common multi-selection features. We present the justification for the proposed features but do not compare them to other possible feature sets in user studies.

The primary contribution of the paper is not the proposed selection features, which build significantly on well-known guidelines [2, 12], but rather the abstractions that capture the essence of multi-selection features and enable their precise and concise specification, and their generic implementation. We describe a JavaScript library that provides a rich multi-selection feature with support for keyboard and mouse operations, undo/redo commands, and selecting elements based on a predicate. The implementation is not tied to a specific framework or widget. With little per-instance development effort, developers can reuse the feature in different contexts and provide users the same consistent selection semantics everywhere. The paper thus shows how to make a ubiquitous user interface behavior reusable; it replaces a user interface implementation guideline with a precise specification and a reusable implementation.

## 2 Multi-selection today

This section discusses the selection features of a sample of popular applications. Our intent is not to criticize particular applications, vendors, or developers, but rather to point out the variation in these features and that even applications with large user bases, after numerous releases, exhibit problems in their selection features.

For a detailed understanding of how multi-selection works in an application, one must resort to reverse engineering through experimenting with the feature. The descriptions in help systems or user guides, if they exist, tend to be cursory. For example, Apple's documentation for OS X El Capitan [4], primarily describing Finder, states the following:

- To move, copy, and make changes to items, you usually have to select them first.
- Select an item: Click the icon for any document, folder, app, or disk.
- Select multiple items: Hold down the Command key, then click the items.
- Select multiple items that are listed together: Click the first item, then press the Shift key and click the last item. All items in between are included in the selection.
- You can also click near the first item, hold down the trackpad or mouse button, then drag over all of the items.
- Select all items in a window: Click a window to make it active, then press Command-A.
- When you selected multiple items and want to deselect one of them, Command-click the item.

This description is vague and incomplete. Aspects not described include selecting multiple disjoint regions with command-click shift-click pairs, drags started with a command-click (they toggle), keyboard selection, the nuances of anchor (explained below) placement, and that shift-click can sometimes deselect. Further, the help document applies only to views that are sequentially ordered; in other views the commands have significantly different meanings.

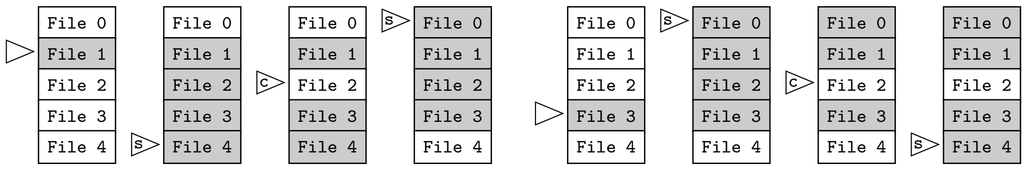
The above description is of course not intended to be a detailed documentation of Finder's multi-selection features. Such documentation in user guides would be of little benefit—few users today turn to documentation to learn how to use an application, let alone a seemingly trivial feature such as selection. Indeed, users should not need to consult a manual to use multi-selection. Multi-selection gestures and their meaning should be part of a common "user interface language" that we as users know and expect applications to understand.

Pieces that can serve as beginnings of such a language exist. In most applications, a mouse *click* with no modifier keys means selecting an individual element, a *command-click* (on a Mac, *control-click* on a PC) toggling an element, and a *shift-click* selecting elements (from somewhere) "up to" an element. Dragging a mouse to indicate a rectangular area usually means selecting the elements that fall on that rectangle. There are also established key bindings for selecting elements using the keyboard. The precise meanings of these actions, however, are not established, as evidenced by the below review of sample applications.

## 2.1 OS X Finder multi-selection

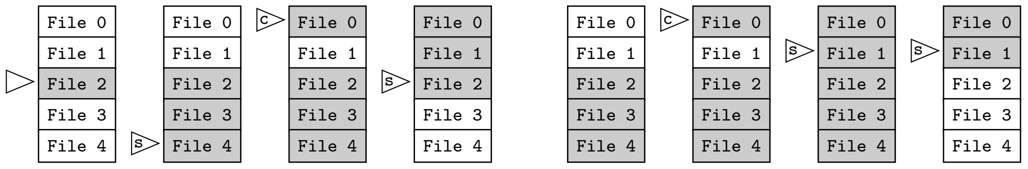
Finder supports different kinds of views. A sequential view shows each file as a single line. A two-dimensional view shows files as icons, placed at arbitrary locations. Selection operations are different for different views. We first discuss selection in sequential views. We refer to the objects being selected as *elements*, whether they are files, lines, or icons.

At every click or command-click, Finder establishes an *anchor* at the clicked element. A subsequent shift-click selects the range of elements from the anchor to the clicked element, the *active end* of the range. The terminology comes from the Macintosh guidelines [2]. We refer to the set of elements indicated by the anchor and the active end as the *active (selection) domain*. Further shift-clicks change the active end, and thus also the active domain; a shift-click farther away from the anchor adds elements to the active domain, closer to the anchor removes elements from it, and on the opposite side of the anchor both removes (from



**(a)** **(b)**

**Figure 1** Selection outcomes in Finder. The triangles mark the elements the user clicks: `s` is a shift-click and `c` a command-click. Each column shows the selection state after the click. The darker elements are selected. The series of operations in the diagram (b) is symmetric with that of the diagram (a). The symmetry axis is the element `File 2`. The selection results are not symmetric.



**(a)** **(b)**

**Figure 2** Selection outcomes in Finder. The series (a) shows that when a shift-click occurs on an already selected element, deselections outside the active domain can take place. The series (b) shows that the effect of shift-clicking is not idempotent.

the anchor to the previous active end) and adds (from the anchor to the new active end). Finder does not remember the prior selection state of the elements in the active domain; when a shift-click shrinks the active domain, the elements that were dropped from the domain will unconditionally become deselected, rather than restored to their prior state.

To foresee the effect of a shift-click, the user needs to be aware of the anchor's location. The anchor is not, however, visible to the user, and the rules for where it resides are complex. For example, if a command-click deselects a previously selected element, the anchor is placed on the first selected element downward from that element; or if no such element exists, on the first selected element upward; or if no such element exists, on the first element of the view.

The anchor placement rules lead to surprises, as the two series of operations in Figure 1 demonstrate. It seems reasonable to expect that a command-click followed by a shift-click has no effect outside the range between the locations of these two clicks. Yet in the first series, an element outside of this range toggles from selected to not selected. The second series is by itself not surprising but considered together with the first one it is—two symmetric selection operations lead to asymmetric results.

Another surprising part of the selection rules is the effect of shift-clicking an element that is currently selected: if a shift-click occurs on an element that belongs to a contiguous range of selected elements, all elements in that range are deselected prior to selecting the elements in the new active domain. Figure 2 shows two cases of such overly eager deselection.

It is possible to understand Finder's rules, as our reverse engineering effort demonstrates. We doubt that many will go through a similar exercise. After all, Finder is only one of many contexts of multi-selection and knowing its peculiar set of rules helps little elsewhere.[1]

[1] Finder's multi-selection behavior appears in a few contexts outside of Finder, such as in the File selection dialog shared by many applications and in Apple Mail's (version 6.5) folder and message lists—though not in the list of messages within a thread. Incidentally, the selection features of Apple's iCloud Mail

## 2.2 Microsoft's File Explorer multi-selection

Microsoft's File Explorer's multi-selection feature is similar, yet subtly different from that of Finder. Click behaves as in Finder: an anchor is established on the clicked element, the element is selected, and all other selections are cleared. Control-click roughly corresponds to command-click on Finder: it toggles the current element and establishes a new anchor. The anchor is always placed on the clicked element, be it selected or not. This is different from Finder, where the anchor is never placed on an unselected element. In addition to setting the anchor, control-click establishes a *mode* of selecting: if the control-click occurred on a selected element, the mode is to deselect, otherwise to select elements. The mode affects subsequent *control-shift-clicks* (Windows' selection documentation does not mention control-shift-click [11]), each of which establishes a new active end and either selects or deselects according to the mode. Shift-click sets the active end, selects the elements in the active domain, and clears all other selections.

File Explorer's model is relatively consistent, but different from most anything else; differences manifest even within File Explorer itself. For instance, changing to a hierarchical view of files disables multi-selection altogether.

## 2.3 Gmail multi-selection

Gmail's web interface supports multi-selection in the list of mail messages and single-selection in all other collections: mail folders, contacts, and category tabs. Each element on the message list has a checkbox for receiving the mouse clicks that control selection.

Click and command-click are interchangeable: they toggle the element that was clicked and set the anchor to that element. Shift-click sets the active end and either (1) selects the elements from the anchor to the active end, if the active end is on an element that was not selected, or (2) deselects those elements, if the active end is on an element that was selected. Shift-click then places a new anchor, at the active end.

The behavior that results from this rule set differs in at least two significant ways from that of Finder and File Explorer. First, click does not discard the current selection. For that effect, one needs to resort to the keyboard shortcut `*n`, or a click on or above the first selected element followed by a shift-click on the last selected element. Second, whether shift-click selects or deselects elements depends not on the anchor but on the active end; in File Explorer's control-shift-click the anchor's selection status determines the mode of selecting. As a result, in Gmail a shift-click always either grows or shrinks the current selection, not both. The notion of an active domain modified by shift-clicking does not apply.

Gmail's selection rules are relatively simple, but their effect may still surprise. If the current selection is a contiguous region, a user may get the impression that shift-click always grows or shrinks the selection either at the top or bottom of a selected region. If, however, a shift-click that shrinks the region at one end is followed by a shift-click beyond the other end, the region grows at both ends. Figure 3 illustrates such a sequence that arises, for example, in the following scenario: a user (1) clicks to select some message $m$, (2) shift-clicks to select a few more messages above $m$, (3) realizes that too many were selected and backs down one or more messages by shift-clicking some message $k$, still above $m$, and finally (4) shift-clicks in an attempt to extend the selection with messages below $m$. That $k$ becomes selected is peculiar, and might go unnoticed as $k$ could reside outside the current viewport.

app, which is made to look quite similar to the desktop mail client, are significantly different.

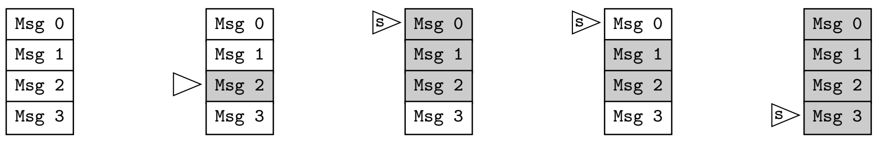


**Figure 3** A series of mouse events and their effect in Gmail's multi-selection semantics. That the last shift-click extends the selected region at both ends may surprise.

Gmail exposes the anchor location, which makes it possible to predict the effect of a shift-click in all cases. Gmail also maintains a cursor indicating the current message for keyboard navigation and commands. Clicks move that cursor too, but we have not been able to understand the exact rules guiding those movements.

Gmail's selection behavior is a departure from the conventions in OS X and Windows. The differences manifest concretely when a Gmail user selects files to attach to a message and is subjected to the OS's multi-selection semantics.

## 2.4 Multi-selection with keyboard

Intuitive keyboard commands for selection are important. The preferred and effective methods to perform selections vary from one user to another. Karat et al. compared touch, mouse, and keyboard in selecting one (menu) element at a time, and concluded that none of the devices was the most effective for or most preferred by all users [8].

File Explorer's keyboard bindings map rather consistently to mouse clicks so that it is possible to use the mouse and the keyboard interchangeably. Space corresponds to a click and arrow keys to a cursor movement followed by a click. The effect of modifiers is the same as for mouse clicks, except that control combined with an arrow is just movement. That control suppresses the automatic click action is useful, so that moving around without altering the current selection is possible. Such capability is missing from, e.g., the MS Word text editor, even though it supports selecting multiple disjoint regions with a mouse.

Finder's keyboard bindings are less complete than File Explorer's. Arrow keys correspond to moving to the neighboring element followed by a click, or shift-click if the shift key is held down. Command-click has no counterpart; keyboard commands thus seem to be insufficiently expressive for selecting disjoint regions.

Gmail's selection key commands are off by default and limited when on: arrow keys move the cursor that indicates the current element, `x` toggles, `*a` selects all, and `*n` deselects all.

## 2.5 Rubber band selection

A mouse drag can be used to select a range: the anchor is established at the *mousedown* and the active end at the *mouseup* event's position. While a drag is ongoing, the current mouse position is interpreted as the active end. Often user interfaces show the changing minimum bounding rectangle of the points of the anchor and active end during the drag. File Explorer and Finder support this kind of *rubber band selection*, whereas Gmail does not.

In File Explorer, rubber band selection without modifier keys clears the current selection and selects all elements in the specified region; the shift modifier suppresses the clearing of the current selection. Control and shift-control toggle the elements inside the rubber band.

Finder's rubber band selection without modifiers is as in File Explorer, but both the shift and command modifiers toggle elements. No command unconditionally adds elements to or removes them from the current selection. In both OSs, the meaning of modifier keys is different in rubber band selection than in selection through clicking or using the keyboard.

It is curious that in both OSs rubber band selection supports toggling, but not deselecting. Deselecting has obvious uses, such as selecting a range save a few elements, but use cases for a single mouse drag to flip some elements from unselected to selected and others to the opposite direction are hard to come by. Perhaps toggling is implemented to approximate deselection: when the anchor and active end are wholly within an already selected region, toggling and deselecting produce equal results.

Another curious aspect is that the active domain is treated differently when rubber band selecting than when selecting with mouse clicks. In the former case, when the active domain shrinks, elements revealed from under the active domain are unconditionally deselected. In the latter case, however, their selection status prior to their inclusion to the active domain is remembered and restored. Section 4.8 describes the drawbacks of the former behavior.

## 2.6 Touch applications

In recent years, the popularity of touch devices has risen dramatically. Apple's iOS, Google's Android and Microsoft's Windows 8 are all operating environments designed from the outset for multi-touch displays. Given the long history these companies have with user interfaces, one would have expected the basic concepts, such as selection, to have been refined and carried forward. Instead we find even more disparity and limitations in touch based selection.

In iOS, in Springboard (the application launcher) long-pressing on an icon enters *selection mode*. Only one application can be moved or deleted at a time. In the Mail application, tapping an Edit button (top right of screen) moves into selection mode, where tapping selects messages. After selecting, one chooses an operation to perform on the selection (Trash, Move, Mark All Read, ...). The Photos application works similarly, except that the Edit button is named Select. In the Messages application, again an Edit button (now top left of the screen) enters selection mode. A message is selected by tapping on an indicator to its left. Only one conversation at a time can be selected; the only operation available is Delete.

In Windows 8.1, long-pressing on a title or swiping up from the bottom of the screen and tapping "Customize" on the far right enters selection mode, where multiple tiles can be tapped. In Windows 8.0, selecting a tile was done by swiping down (perpendicular to the horizontal scroll direction). Apparently this approach was abandoned, likely because it was not very discoverable. In the Mail application when one taps on an item, a checkbox appears to its left. Tapping on the check box toggles it on and enters a selection mode where checkboxes appear next to all messages; tapping anywhere on a message toggles its checkbox. Swiping up from the bottom of the screen and tapping "Select" on the left also enters selection mode. In the Photos application a swipe down on a photo toggles its selection state.

In Android 4.4 on the Launcher screen one can only operate on icons individually with a long-press. In the GMail application, one can select multiple email messages by long-pressing on each one individually. In Google Photos one enters selection mode by long-pressing a photo. A drag will then select multiple photos, tap will toggle one photo.

Except for the Google Photos application, none of the above selection models allow quickly selecting a large range of elements. Where multiple selection is supported, each element must be selected one at a time.

## 3 Selection concepts

The above review of multi-selection features in a handful of well-known applications confirms that the semantics of multi-selection are not well established, and that users are faced with a wide variety of behaviors, some surprising. This section describes such semantics, making no assumptions about what kinds of elements are being selected, how they are organized and visually presented to the user, and what kind of a selection tool or device the user is using.

The multi-selection task is to select a subset of the elements of a collection presented to the user. For an abstract view of the task, it is not essential what these elements are. It is thus convenient to think of them as an indexed family $x : I \to M$, where $M$ is the set of elements, and $I$ a suitable index set. With this setup, we can talk about the $i$th element of a collection, where $i \in I$, and mean the element $x_i$. A set of selected elements in $I$ can be represented as a mapping $s : I \to \{\mathsf{T}, \mathsf{F}\}$, where $s(i) = \mathsf{T}$ means that $x_i$ is selected and $s(i) = \mathsf{F}$ that it is not. We call $s$ a *selection mapping*. Below, we use $\mathbf{2}$ for the set $\{\mathsf{T}, \mathsf{F}\}$.

## 3.1 Primitive selection operations

A user constructs a selection mapping with a sequence of selection commands, such as a rubber band selection, followed by a shift-click, press of shift-right arrow, and a command-click. Each user command modifies the current selection mapping, and can thus be modeled as a function of type $(I \to \mathbf{2}) \to (I \to \mathbf{2})$. We call functions of this type *selection operations*. We define *primitive* selection operations, from which all selection operations can be built:

▶ **Definition 1.** Let $x : I \to M$ a collection, $J \subseteq I$, and $f : \mathbf{2} \to \mathbf{2}$ a mapping. $J$ and $f$ (uniquely) determine a *primitive selection operation*

$$\mathsf{op}_J^f : (I \to \mathbf{2}) \to (I \to \mathbf{2}), s \mapsto \lambda i. \begin{cases} f(s(i)), & i \in J \\ s(i), & i \notin J \end{cases}$$

In other words, the primitive selection operation $\mathsf{op}_J^f$ produces a new selection mapping that applies $f$ to the selection state of every element in $J$, and has no impact on elements not in $J$. We call $J$ the *selection domain* and $f$ the *selection function* of $\mathsf{op}_J^f$.

Selection operations compose. Starting from the "no elements selected" selection mapping $e : I \to \mathbf{2}, i \mapsto \mathsf{F}$, the selection that results after applying a series of primitive selection operations $\mathsf{op}_{J_1}^{f_1}, \mathsf{op}_{J_2}^{f_2}, \ldots, \mathsf{op}_{J_n}^{f_n}$ is $(\mathsf{op}_{J_n}^{f_n} \circ \mathsf{op}_{J_{n-1}}^{f_{n-1}} \circ \ldots \circ \mathsf{op}_{J_1}^{f_1})(e)$.

Any change to the selection state of elements can be expressed as a composition of primitive selection operations. The task of implementing a multi-selection feature can thus be viewed as the task of providing the user convenient means to specify primitive selection operations. Since they are uniquely determined by the selection function $f$ and domain $J$, it is these two components that must be derived from user interactions. We first review common ways in which user actions give rise to particular selection functions.

There are four functions of type $(\mathbf{2} \to \mathbf{2})$, and thus four selection functions: $\lambda x.x$, $\lambda x.\mathsf{T}$, $\lambda x.\mathsf{F}$, and $\lambda x.\neg x$. Primitive selection operations of the form $\mathsf{op}_J^{\lambda x.x}$ are identity functions. Operations of the form $\mathsf{op}_J^{\lambda x.\mathsf{T}}$ select all elements in the selection domain, and arise as a result of many actions, including clicking an element, shift-clicking, or rubber band selection. Operations of the form $\mathsf{op}_J^{\lambda x.\mathsf{F}}$ deselect all elements in the selection domain. They can model, e.g., File Explorer's or Gmail's "deselect a region" operation. Operations of the form $\mathsf{op}_J^{\lambda x.\neg x}$ toggle the selection status of the elements in $J$. Finder's command-click on some element $i$ could be modeled as $\mathsf{op}_{\{i\}}^{\lambda x.\neg x}$. Toggling is often limited to singleton sets, but as discussed in Section 2.5, Finder's and File Explorer's rubber band selection can toggle many elements.

## 3.2 Deriving selection domains from user events

Selection domains are determined based on points a user indicates with a pointing device, keyboard, or by tapping. Often only two points, the anchor and active end, are of interest but

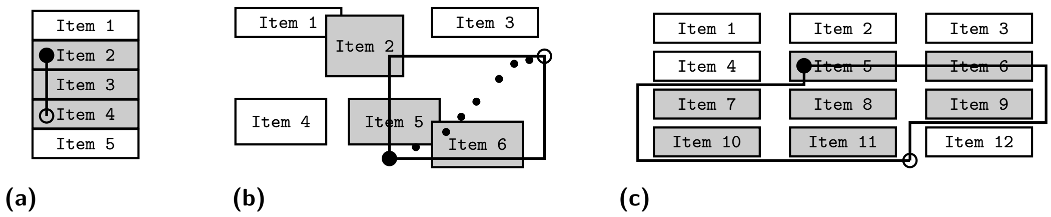


**Figure 4** Illustration of concepts encountered in determining the selection domain. The solid circles indicates where the user has placed the anchor (with a click) and the hollow circles the active end (with a shift-click or a rubber band selection). Dots indicate other points on the selection path, detected during a mouse drag. In these examples only anchor and active end are used.

in general arbitrarily many points may be used. For example, in "lasso" selection, discussed briefly in Section 4.9, the user-indicated points are the vertices of a polygon and the selection domain consists of the elements that intersect with that polygon.

We call the sequence of points that determine a selection domain the *selection path*, and define the *anchor* as the first and the *active end* the last element of this path. In a one-element selection path, the anchor and active end coincide.

How the selection path maps to a set of elements depends on the application. Figure 4 shows three examples. In Figure 4a, the anchor and active end indicate the endpoints of a range of elements; in 4b, the opposite corners of a rectangle with which the selected elements intersect; and in 4c, a range in a row-wise ordering. The last behavior appears, e.g., in text editors, photo organizers, and File Explorer's two-dimensional views.

The anchor and active end, and in general the points on the selection path, are not necessarily mouse coordinates but rather points in some space $V$ that we call the *selection space*. This space can be any coordinate system to which the visual locations of the elements and the user-indicated points can be translated to. For example, in Figure 4a, $V$ is a totally ordered set of element indices, and a mouse coordinate maps to the index of the element that contains it; in 4b, $V$ is the window's coordinate system; and in 4c, the points in $V$ could be tuples that store both a mouse coordinate and the element index, as some points lie outside of any element. We name the function that performs the conversion from mouse coordinates to selection space coordinates m2v; it is provided by the application programmer.

We can now capture all the variation in how different applications allow the user to indicate elements into one function $\mathsf{sdom} : V^* \to \mathcal{P}(I)$ that computes a selection domain from the selection path. Similarly to m2v, sdom is provided by the application programmer. We use the term *selection geometry* for the set of functions that define the application or context-specific aspects of multi-selection.

In Figure 4a, sdom computes the range of indices between the anchor and active end; in 4b, sdom maps a rectangle to the indices of other rectangles it intersects with; and in 4c, sdom finds two elements, one closest (by some suitable definition) to the anchor and the other to the active end, and returns the range of indices between those elements.

Viewing collections of elements as indexed families lets us "abstract away" the elements. The sdom function in turn abstracts away the indices: since the selection domain $J$ is determined by some sequence of points $P$ in the selection space, the primitive selection operations $\mathsf{op}_J^f$ can be defined in terms of $P$, as $\mathsf{op}_{\mathsf{sdom}(P)}^f$. A sequence $\langle f_1, P_1\rangle, \ldots, \langle f_n, P_n\rangle$ of selection function–selection path pairs thus determines the composition of primitive selection operations $\mathsf{op}_{\mathsf{sdom}(P_n)}^{f_n} \circ \ldots \circ \mathsf{op}_{\mathsf{sdom}(P_1)}^{f_1}$ that can compute a new selection mapping.

$$
\begin{aligned}
\mathsf{click}_p : \langle s, op, \_, \_\rangle &\mapsto \langle s, \mathsf{op}^{\lambda x.\,\mathsf{T}}_{\mathsf{sdom}(\cdot|p)} \circ \mathsf{op}^{\lambda x.\,\mathsf{F}}_{\{i \in I \,|\, op(s)(i) = \mathsf{T}\}} \circ op, \cdot|p, \mathsf{ae}(\cdot|p)\rangle \\
\mathsf{c\text{-}click}_p : \langle s, op, \_, \_\rangle &\mapsto \langle s, \mathsf{op}^{\lambda x.\neg\mathsf{onsel}(p, op(s))}_{\mathsf{sdom}(\cdot|p)} \circ \mathsf{op}^{\lambda x.x}_{\varnothing} \circ op, \cdot|p, \mathsf{ae}(\cdot|p)\rangle \\
\mathsf{s\text{-}click}_p : \langle s, \mathsf{op}^{f}_{\_} \circ op, P, \_\rangle &\mapsto \langle s, \mathsf{op}^{f}_{\mathsf{sdom}(P|p)} \circ op, P|p, \mathsf{ae}(P|p)\rangle \\
\mathsf{s\text{-}click}_p : \langle s, op, \_, \_\rangle &\mapsto \langle s, \mathsf{op}^{\lambda x.\,\mathsf{T}}_{\mathsf{sdom}(\cdot|p)} \circ \mathsf{op}^{\lambda x.x}_{\varnothing} \circ op, \cdot|p, \mathsf{ae}(\cdot|p)\rangle
\end{aligned}
$$

**(a)** The three basic selection commands.

$$
\begin{aligned}
\mathsf{undo} : \langle s, o_n \circ o_{n-1} \circ op, \_, p\rangle &\mapsto \langle s, op, \bot, p\rangle \\
\mathsf{undo} : t &\mapsto t \\
\mathsf{bake} : \langle s, op \circ o_4 \circ o_3 \circ o_2 \circ o_1, R, p\rangle &\mapsto \langle \mathsf{store}(s, o_2 \circ o_1), op \circ o_4 \circ o_3, R, p\rangle \\
\mathsf{bake} : t &\mapsto t
\end{aligned}
$$

**(b)** Undo and bake operations.

$$
\begin{aligned}
\mathsf{space}' : \langle s, op, P, p\rangle &\mapsto \mathsf{click}_p(\langle s, op, P, p\rangle) & \mathsf{arrow}'_{dir} : \langle s, op, P, p\rangle &\mapsto \langle s, op, P, \mathsf{step}_{dir}(p)\rangle \\
\mathsf{s\text{-}space}' : \langle s, op, P, p\rangle &\mapsto \mathsf{s\text{-}click}_p(\langle s, op, P, p\rangle) & \mathsf{s\text{-}arrow}'_{dir} &= \mathsf{s\text{-}space}' \circ \mathsf{arrow}'_{dir} \\
\mathsf{c\text{-}space}' : \langle s, op, P, p\rangle &\mapsto \mathsf{c\text{-}click}_p(\langle s, op, P, p\rangle) & \mathsf{c\text{-}arrow}'_{dir} &= \mathsf{arrow}'_{dir} \circ \mathsf{c\text{-}space}'
\end{aligned}
$$

**(c)** Keyboard selection operations.

**Figure 5** The basic commands of the selection language.

## 4 Selection language

We give precise meanings to multi-selection features by defining the components that constitute the state of a selection and the functions that define the valid transformations on that state. These definitions are a language for implementing multi-selection features. The language is parametrized by a selection geometry, a handful of functions that the programmer defines. As a heads up, in addition to the already introduced sdom and m2v, the other functions of the selection geometry are step, default, and $|$, and a function to filter elements of the index set $I$ based on a predicate.

Figure 5 defines the basic commands of the selection language. Each command is a function that maps the current selection state to a new state. The function definitions rely on pattern matching to express concisely the complete effect that a command has on the selection state. If more than one pattern matches, the first matching definition applies.

The selection state is a tuple of the form $\langle s, op, P, p\rangle$, where $s : I \to \mathbf{2}$ is the base selection mapping; $op$ the composition of primitive selection operations that have not yet been applied and stored to $s$; $P$ the selection path; and $p$ the *keyboard cursor*, explained in Section 4.4.

The third element of the tuple has two uses. In lieu of a selection path, it can contain a *selection predicate*. The metavariable $P$ always signifies a path and the metavariable $Q$ a selection predicate; selection predicates are explained in Section 4.5. The third element can also have an undefined value $\bot$. Neither metavariable $P$ nor $Q$ matches $\bot$. The metavariable $R$ matches any of these three kinds of values.

The selection mapping that reflects the currently selected elements is obtained from $op$ and $s$ as $op(s)$: the selected elements are $\{i \in I \mid op(s)(i) = \mathsf{T}\}$. We consider the empty composition, denoted as $op = \cdot$, to be the identity function, so that $\cdot(s) = s$. The metavariable $o$ ranges over primitive selection operations and $op$ over compositions of zero or more primitive selection operations. While function composition has its regular meaning, we also assume that the composition structure is accessible, so that pattern matching can extract individual primitive selection operations. For example, the pattern $o_1 \circ o_2$ matches compositions that have exactly two primitive selection operations, binding $o_1$ to the first and $o_2$ to the second. The metavariable $op$ can appear on either end of a pattern: $op \circ o$ and $o \circ op$ both match compositions that have one or more primitive selection operations. Finally, the metavariable _ binds to anything and signifies an unused value.

A selection path is a sequence of points in $V$. The symbol $\cdot$ denotes the empty sequence. The operator $|$ is one of the context-dependent functions whose meaning can vary, but to simplify, we assume for now that it extends a sequence with a new point. For example, if $P = p_1, \ldots, p_n$, then $P|p$ is the sequence $p_1, \ldots, p_n, p$. The function $\mathsf{ae}(P)$ returns the active end, the last element of a sequence, or the special undefined value $\perp$ if $P$ is the empty sequence. We thus overload $\perp$ to signify either an undefined point or an undefined path/selection predicate. We assume that we can query whether a value is undefined by matching against the pattern $\perp$.

All functions of the selection language maintain the invariant that the composition of primitive selection operations has an even number of elements. This is a technicality that makes the definition of the language more concise at the expense of sometimes adding extra empty operations to the composition. A possible valid initial empty selection state that satisfies the invariant is $\langle e, \cdot, \perp, \perp \rangle$, where $e$ is the empty selection mapping.

## 4.1 Meaning of mouse clicks

Figure 5a shows our definitions for the three basic operations for constructing selections: click, shift-click, and command-click. The point parameter $p$ to each operation is written as a subscript, as in $\mathsf{click}_p$, to keep it notationally separate from the selection state parameter. The click functions do not modify the base selection mapping $s$; the current selection mapping $op(s)$ changes because the composition $op$ is modified.

The $\mathsf{click}_p$ function adds two primitive selection operations to $op$. The first operation clears the current selection; its selection domain is the currently selected elements and not $I$, which might be difficult to express, even unbounded in some selection geometries. The second operation selects the element(s) indicated by $p$, if any. The previous selection path is discarded and $p$ becomes the only element of the new selection path, and thus both the anchor and active end. It also becomes the keyboard cursor. If elements can overlap in $V$, the definition of $\mathsf{sdom}$ may need to handle this one-element selection path specially. The established behavior is that a click selects at most one element. Hence, $\mathsf{sdom}(\cdot|p)$ should either return the singleton set whose element is the index of the "topmost" element that $p$ could indicate, or the empty set if $p$ indicates no elements.

The $\mathsf{c\text{-}click}_p$ function is similar to click in that it clears the selection path and sets a new anchor. If $p$ indicates an element, command-click should toggle that element. Instead of using the toggling selection function $\lambda x.\neg x$, $\mathsf{c\text{-}click}_p$ uses either $\lambda x.\,\mathsf{F}$ or $\lambda x.\,\mathsf{T}$, depending on whether $p$ is on a selected or unselected element. This is to establish the mode of selecting, i.e., whether subsequent shift-clicks should select or deselect. The helper function $\mathsf{onsel}(p, s) = \textbf{if}\, \mathsf{sdom}(\cdot|p) = \{i\}\, \textbf{then}\, s(i)\, \textbf{else}\, \mathsf{F}$ determines whether $p$ is on a selected element. One primitive selection operation would suffice to capture the effect of a command-click, but to maintain the invariant of an even number of primitive selection operations, command-click first adds the identity operation $\mathsf{op}_{\varnothing}^{\lambda x.x}$ to the composition.

Shift-click is defined by two cases. The first matches if both the $op$ composition is not empty and the selection path is defined. The second matches all tuple values; it is thus selected if $op$ is empty or if the tuple's third component is not a path (if it is $\perp$ or a selection predicate). In the first case, the point $p$ is added to the selection path as the new active end. A new selection domain is computed from this path to replace the selection domain of the outermost primitive selection operation in the composition. Shift-click thus cancels the current active domain and establishes a new one. There may be no selection path to extend or no outermost primitive selection operation, so in the second case a new path is established and a new pair of primitive selection operations added. The effect is the same as that of a command-click, except that the mode of selecting is always set to select.

All three click functions extend the path with $p$, then use ae to access the path's active end and make it the new keyboard cursor. This may seem like a roundabout way of setting the cursor to $p$, and indeed it usually is. However, since the $|$ operator is a parameter to the selection language, it might be defined to sometimes not make $p$ the active end. For example, in some selection contexts it is useful to ignore certain points, such as coordinates outside of any selectable elements. This is easily modeled by defining $|$ to keep the path unchanged when called with such a point. To enable this customization point, all selection path manipulation is performed with the $|$ operation. The reason for setting the cursor using ae is now plain: when a point is ignored, we get the desired effect that click and command-click set the keyboard cursor to $\perp$ and shift-click keeps it unchanged.

The $|$ function can actually be defined to modify the selection path in arbitrary ways. For example, when only the anchor and active end are of interest, $|$ can limit the length of the selection path to two elements, throwing away the intermediate points. As another example, one of our example applications defines a selection geometry where $|$ may remove points from the end of the selection path to get the effect of erasing points of a lasso selector.

We consider the above three operations to be a sufficient set for convenient manipulation of the selection mapping with a pointing device. Compared to File Explorer and Finder, an operation for toggling many elements simultaneously is missing. It would not be problematic to specify or implement but, as discussed in Section 2.5, its use cases are not obvious.

## 4.2 Meaning of rubber band selection

In existing practice, rubber band selection and selecting via mouse clicks often have different semantics. For example, Finder's rubber band selection always toggles, instead of selecting or deselecting. We equate mouse drag to shift-click, following old apparently forgotten advice [2, 12]. More precisely, the meaning of "drag mouse to point $p$" is exactly that of s-click$_p$; every *mousemove* event until a *mouseup* should be interpreted as a shift-click. The result of this equivalence is the interchangeability of the two selection mechanisms.

To appreciate the benefits, consider a rubber band selection that selects a region that extends beyond the current window's visible area. Many user interfaces scroll the window when the mouse is dragged past the window's edge. It is easy to "overshoot", then be forced to scramble back by dragging the mouse to the opposite edge of the window, possibly overshooting again, and at all times being careful not to release the mouse button. When shift-click has the same meaning as mouse drag, one can let go of the mouse at any point, scroll by whatever method is convenient (scrollbar, page down key, two-finger scroll, etc.), and use shift-click to finish the selection or pick up the rubber band and continue the drag. MS Word supports this behavior in text selection and Excel in selecting spreadsheet cells.

## 4.3 Undo, redo, and baking

In the presented framework, undoing selection commands means simply removing outermost primitive selection operations from the composition. Figure 5b shows the undo function.

Every undoable selection command adds two operations to the composition, so undo removes two operations at a time.

The undo function sets the selection path to the undefined value $\perp$. This is because undoing exposes a new primitive selection operation whose selection domain was not computed from the current selection path (or predicate), and thus keeping the current path would make a subsequent shift-click command very unpredictable. An undefined path forces shift-click to add a new pair of primitive selection operations, avoiding the surprising behavior. Note that undo preserves the keyboard cursor; there are no surprises in doing so. The second case of undo is an identity function; it is applied if there are no operations to undo.

To support redo, the operations removed from the composition by undo can be stored to a stack of redoable selection operations, to be composed back when requested. We do not show the definitions. Our reference implementation, described in Section 5, implements redo.

There are no established key bindings for undo/redo of selections, since such a feature is seldom, if ever, provided. Adopting typical undo/redo bindings, such as command-Z/shift-command-Z, would mean coalescing the selection undo stack and the application's command undo stack, making selection undo subordinate to command undo: executing a command on the selection (such as deleting or copying the selected files) would clear the selection undo stack. Selections would be undoable only until the most recent command. We have not explored all the ramifications, and thus keep selection undo separate from command undo. We bind option-Z and shift-option-Z to selection undo and redo, respectively.

Every new click or command-click operation grows the composition of primitive selection operations. For reducing the size of the composition, without changing which elements are selected, we introduce the bake function in Figure 5b. It extracts the two least recently added primitive selection operations from $op$ and "bakes" their effect permanently to the base selection mapping $s$; we assume that $\mathsf{store}(op, s)$ is an operation that constructs this new base selection mapping from $op$ and $s$. The second case of bake is an identity function; it is applied if fewer than two primitive selection operations would remain in $op$ after baking. Emptying the composition completely would bake the operation that defines the active domain and thus change, e.g., shift-click's behavior. To impose a limit on the size of the composition, an implementation calls bake whenever a predefined upper limit is reached.

## 4.4 Meaning of keyboard selection operations

When navigating with arrow keys is meaningful, constructing selections with the keyboard should be possible and convenient. As discussed in Section 2.4, applications' support for keyboard selection varies considerably. This is expected since selection features are not the only candidates competing for the relatively small set of convenient key combinations. We choose a particular set of key bindings and give a few arguments to justify them below, but recognize that they are not suitable everywhere. Again, the contribution is the precise meanings of the keyboard selection operations, not particular key bindings.

The keyboard selection functions, named to suggest our key bindings, are shown in Figures 5c and 6. The functions do not manipulate the selection path or call sdom directly; they are all defined in terms of the three click functions. The point argument they pass to the click functions is the current keyboard cursor, which is a point in the selection space $V$.

Assuming $p$ is the keyboard cursor, $\mathsf{space}'$ has the effect of $\mathsf{click}_p$, $\mathsf{s\text{-}space}'$ the effect of $\mathsf{s\text{-}click}_p$, and $\mathsf{c\text{-}space}'$ the effect of $\mathsf{c\text{-}click}_p$. Note the "primed" function names in Figure 5c. These functions assume that the keyboard cursor is defined. If, however, no click operations have yet been applied, the cursor may have its initial undefined value. Figure 6 shows the corresponding functions that accept $\perp$ as the keyboard cursor; these are the functions to

$$\begin{aligned}
\mathsf{space}'\colon \langle s, op, P, p\rangle &\mapsto \textbf{if } \mathsf{crs_s}(p) \neq \bot \textbf{ then } \mathsf{space}(\langle s, op, P, \mathsf{crs_s}(p)\rangle) \textbf{ else } \langle s, op, P, p\rangle\\
\mathsf{s\text{-}space}'\colon \langle s, op, P, p\rangle &\mapsto \textbf{if } \mathsf{crs_s}(p) \neq \bot \textbf{ then } \mathsf{s\text{-}space}(\langle s, op, P, \mathsf{crs_s}(p)\rangle) \textbf{ else } \langle s, op, P, p\rangle\\
\mathsf{c\text{-}space}'\colon \langle s, op, P, p\rangle &\mapsto \textbf{if } \mathsf{crs_s}(p) \neq \bot \textbf{ then } \mathsf{c\text{-}space}(\langle s, op, P, \mathsf{crs_s}(p)\rangle) \textbf{ else } \langle s, op, P, p\rangle
\end{aligned}$$

$$\mathsf{arrow}_{dir}\colon \langle s, op, P, p\rangle \mapsto \textbf{if } p \neq \bot \textbf{ then } \mathsf{arrow}'_{dir}(\langle s, op, P, p\rangle) \textbf{ else} \langle s, op, P, \mathsf{crs}_{dir}(p)\rangle$$

$$\mathsf{s\text{-}arrow}_{dir}\colon \langle s, op, P, p\rangle \mapsto \begin{cases} \mathsf{s\text{-}arrow}'_{dir}(\langle s, op, P, p\rangle) & \text{if } p \neq \bot\\ \langle s, op, P, \bot\rangle & \text{if } \mathsf{crs}_{dir}(p) = \bot\\ \mathsf{s\text{-}space}'(\langle s, op, P, \mathsf{crs}_{dir}(p)\rangle) & \text{otherwise} \end{cases}$$

$$\mathsf{c\text{-}arrow}_{dir}\colon \langle s, op, P, p\rangle \mapsto \begin{cases} \mathsf{c\text{-}arrow}'_{dir}(\langle s, op, P, p\rangle) & \text{if } p \neq \bot\\ \langle s, op, P, \bot\rangle & \text{if } \mathsf{crs}_{dir}(p) = \bot\\ \mathsf{c\text{-}space}'(\langle s, op, P, \mathsf{crs}_{dir}(p)\rangle) & \text{otherwise} \end{cases}$$

$$\mathsf{crs}_{dir}\colon p \mapsto \textbf{if } p = \bot \textbf{ then } \mathsf{default}_{dir} \textbf{ else } p$$

**Figure 6** Arrow and space commands with default keyboard cursor positions.

call from client code. We discuss the primed functions first, as they explain the meaning of keyboard commands in relation to the basic click functions without confusion.

We adopt spacebar as the keyboard equivalent of clicking, following the example of File Explorer (which does not, however, bind the unmodified spacebar to a selection command). Many applications bind space to other events: Finder applies the “open” command to the selected file(s) and Gmail delegates to the default binding in browsers to scroll down the screen. It is not critical if spacebar is too contested to be used for selection. The arrow keys with the bindings we propose are sufficient for selecting multiple disjoint ranges of elements.

The arrow functions are parameterized by a direction, where $dir \in \{\mathsf{left}, \mathsf{right}, \mathsf{up}, \mathsf{down}\}$, so that one definition covers all four arrow keys. The $\mathsf{arrow}'_{dir}$ functions are for arrow keys without modifiers. They do not change the selection state of any element. They move the keyboard cursor to the specified direction according to the step function discussed below. We find it natural that arrow keys move the cursor freely. This is the convention in text editors and it is analogous to the mouse moving the mouse pointer. This choice, however, does not optimize for selecting a single element, which is arguably the most common need.

Arrows with modifiers predictably combine a keyboard cursor move with a similarly modified selection operation: $\mathsf{s\text{-}arrow}'$ function is the composition of the $\mathsf{s\text{-}space}'$ and $\mathsf{arrow}'$ functions, and $\mathsf{c\text{-}arrow}'$ the composition of $\mathsf{arrow}'$ and $\mathsf{c\text{-}space}'$ functions. Importantly, with the command modifier the selection operation is applied first and move second; with the shift modifier move is applied first. The intuition is that with the command modifier one indicates where a selection region starts and with shift where it ends.

### 4.4.1 Keyboard selection and coordinates

The arrow function uses the function $\mathsf{step} : \langle\{\mathsf{left}, \mathsf{right}, \mathsf{up}, \mathsf{down}\}, V\rangle \to V$ to determine how the cursor moves in response to arrow keys. This function depends on the coordinate system of the selection space $V$ and on the placement of elements in $V$. It is one of the functions of the selection geometry and must be defined by the application programmer.

In a grid layout, the definitions of all directions are obvious. In a row-wise sequential ordering (text editors, for example), left and right are trivial, up moves to a position on the line above the current cursor position and down on the line below. Rules for which position on a line to move to depend on the application. E.g., text editors typically choose the position with the smallest difference along the $x$-axis, except when the cursor is on the last position of a line, in which case the last position of the new line is chosen as the new cursor location. When elements can be positioned in arbitrary locations, the step function can be rather complex. It may, for example, examine a sector from the cursor towards the selected direction for candidate elements and choose one according to some criteria. In Finder, an element's fitness seems to increase when the angle from the requested direction and distance from the current element decrease.

### 4.4.2 Keyboard cursor defaults

A selection context usually starts with no keyboard cursor set, and thus a mechanism for defining default values for the cursor is necessary. The most natural default may vary depending on the issued keyboard command. For example, in a vertical list of elements, arrow down might set the cursor to the first element, arrow up to the last element, and spacebar might not have a default at all. We rely on the selection geometry's function $\mathsf{default}_{dir}$ to provide the default keyboard cursor value for each of the four arrow keys, and additionally for the direction s, which we use to mean the spacebar. Thus here $dir \in \{\mathsf{left}, \mathsf{right}, \mathsf{up}, \mathsf{down}, \mathsf{s}\}$. If some direction $d$ has no default, then $\mathsf{default}_d$ should return $\perp$.

Figure 6 defines the keyboard selection functions to be used by the client. They inspect the cursor, and, if it is undefined, try to establish it by invoking the default function. If the cursor remains undefined, the keyboard functions have no effect. The space functions are simple wrappers over the corresponding primed functions in Figure 5c. The arrow functions are a bit more complex; they only delegate to the corresponding primed function if the keyboard cursor is defined. If the cursor is initialized with a default value, the corresponding primed function is not invoked; it could be confusing if the cursor was first set to the default, then immediately moved by the primed function. This is why s-arrow and c-arrow, after establishing a cursor from a default value, forward to $\mathsf{s\text{-}space}'$ and $\mathsf{c\text{-}space}'$, respectively.

Supporting default cursor values adds some complexity to the keyboard functions, but the additional definitions are mostly repetition of the same wrapping pattern. Importantly, it is very easy for the application programmer to add default cursor positions to a multi-selection feature: the default function is essentially a lookup table of at most five key-value pairs.

## 4.5 Selecting based on a predicate

Apart from selecting elements with a pointing device, a user may wish to select or deselect elements based on their properties. For example, a user may want to select all files whose name matches a particular string. To integrate a predicate-based selecting or deselecting into the selection language, the *selection predicate* takes a similar role as the selection path, and determines a selection domain. Selecting or deselecting is then accomplished in the usual manner of adding primitive selection operations to the composition.

Figure 7 defines two operations, $\mathsf{predicate\text{-}select}^b_Q$ and commit, where $Q : I \to \mathbf{2}$ is a predicate on the indices, and $b \in \mathbf{2}$ determines whether the operation selects or deselects. The selection domain that $Q$ defines is $\{i \in I | Q(i)\}$. The selection language does not dictate the representation of $I$, so we assume that the selection geometry implements an efficient way to iterate over $I$ and find the elements that satisfy $Q$.

The predicate-select command works analogously to shift-click: if a predicate has already been defined (the previous command was predicate-select), the topmost primitive selection operation is replaced with one whose selection domain is computed using the new predicate $Q$. The first rule matches when a predicate is defined because the metavariable $Q'$ only

$$
\begin{aligned}
\mathsf{predicate\text{-}select}^b_Q &: \langle s, \_ \circ op, Q', p\rangle \mapsto \langle s, \mathsf{op}^{\lambda x.b}_{\{i \in I \mid Q(i)\}} \circ op, p\rangle \\
\mathsf{predicate\text{-}select}^b_Q &: \langle s, op, \_, p\rangle \mapsto \langle s, \mathsf{op}^{\lambda x.b}_{\{i \in I \mid Q(i)\}} \circ \mathsf{op}^{\lambda x.x}_{\varnothing} \circ op, p\rangle \\
\mathsf{commit} &: \langle s, op, \_, p\rangle \mapsto \langle s, op, \bot, p\rangle
\end{aligned}
$$

**Figure 7** The commands for selecting with a predicate.

matches predicates, not paths or ⊥. If there is no predicate (the third tuple component is either ⊥ or a selection path), the second rule applies and a new pair of primitive selection operations is added.

The commit function makes the effect of a predicate-selection permanent, in the sense that a new undoable state is created. This is accomplished by setting the current predicate to ⊥, so that a subsequent predicate-selection command will add a new pair of primitive selection operations before applying a predicate. Note that even though shift-clicking behaves quite similarly to predicate-selection, only the latter needs a dedicated commit command; an active domain resulting from shift-clicks can be made permanent with a command-click.

Using the same component of the selection state tuple for both selection paths and predicates may seem confusing. It rather naturally guarantees, however, that shift-clicks and predicate-selection commands do not interfere with each other. Since predicate-selection considers anything other than a predicate the same as undefined, and shift-click anything other than a selection path the same as undefined, both a predicate-selection command after a shift-click and a shift-click command after predicate-selection will always add a new pair of primitive selection operations, as they should.

## 4.6 Summarizing the selection language

The definitions above are the complete selection language. They define the meaning of selection operations to be bound to the click, shift-click, and command-click mouse events, as well as to the space, shift-space, command-space, arrow, shift-arrow, and command-arrow keyboard events. They define the undo and redo operations and selection based on elements' properties. The language is parametrized by the functions sdom, m2v, step, default, and |, and a filtering function. Implementing these six functions provides a selection model with all the features expected of a full-fledged selection feature, and probably more (e.g., undo).

Section 5 describes a concrete implementation of the selection language, but below we first touch on a few additional aspects of it.

## 4.7 Touch gestures

As discussed in Section 2.6, multi-selection in touch platforms is a wild-west, with a wide variety of gestures and meanings for those gestures. Adapting our model to touch platforms is mostly future work, but we see no reasons not to provide the same basic selection operations with the same semantics in both touch and non-touch platforms. Due to no good way of modifying taps (as with shift or command), the challenge is to identify natural gestures that can be given the meanings of click, shift-click, and command-click operations of our selection language, and that can continue as a rubber band selection.

### 4.8 Active domain

Section 2.5 pointed out two different responses to the shrinking of the active domain. The question is what should be the selection state of the elements that were in the active domain before the active end changed but are not in it anymore. Finder's answer is "unselected" if the active end was changed as the result of a shift-click, and "whatever the state was before" if it was changed via a mouse drag. This difference seems arbitrary. Our view is that regardless of the means of selection, the prior selection state should always be restored. Otherwise it is easy to unintentionally destroy existing selection state by shift-clicking or dragging the mouse "too far". Furthermore, the use case for not remembering the prior selection state under the active domain is contrived. It can be distilled to the user specifying some active domain $J_1$, then shift-clicking to change it to $J_2$, and expecting all elements in $J_2$ to become selected and those in $J_1 \setminus J_2$ deselected—this is not an intuitive operation.

It is easy to see that our selection operations remember and restore elements' prior selection state. Let $s$ be some base selection mapping, and $\mathsf{op}^f_{J_1} \circ op$ the current composition of selection operations. The current selection mapping is then $(\mathsf{op}^f_{J_1} \circ op)(s)$ and the active domain is $J_1$. If the user changes the active end to $J_2$, either by shift-click, mouse drag, shift-arrow, shift-space, or predicate-selection, then the new selection mapping is $(\mathsf{op}^f_{J_2} \circ op)(s)$; any effect that $\mathsf{op}^f_{J_1}$ had on the selection state is completely canceled.

### 4.9 Selections that use the entire selection path

All our examples thus far have assumed that the selection domain is determined by at most two selection space points: the anchor and active end. Yet the functions in Figure 5 maintain an arbitrarily long selection path, with each shift-click operation adding more points to it We wrote the formalism, and our implementation, in this manner to be able to support selection geometries where more than two points are needed to define the selection domain. For example, in "lasso" selection the points of an entire mouse drag are taken as the vertices of a polygon; items that intersect with the polygon belong to the selection domain. Since the entire selection path is passed as input to the sdom function, supporting lasso selection is simply a matter of defining sdom appropriately.

## 5 Packaging selection behavior into a library

We provide a reference implementation of the selection language as a JavaScript library at `http://hotdrink.github.io/multiselectjs`. It is a faithful implementation of the specification, though on the surface there are a few differences; we point them out below. The library comes with a tutorial and example applications. The examples show concretely the division of labor: the tasks required of the application programmer vs. the services that the library provides. Further, they make it clear that the selection semantics remains the same in all selection contexts, even if the selection geometry is different. The library allows switching between selection geometries on-the-fly without losing the current selection state. Figure 8 shows three snapshots from an example application packaged with our library; each snapshot is taken when a different selection geometry object was in use.

### 5.1 SelectionState class

The central abstraction in the library is the SelectionState class (we emulate classes by objects with the usual JavaScript conventions). The elements of the selection state tuple are member

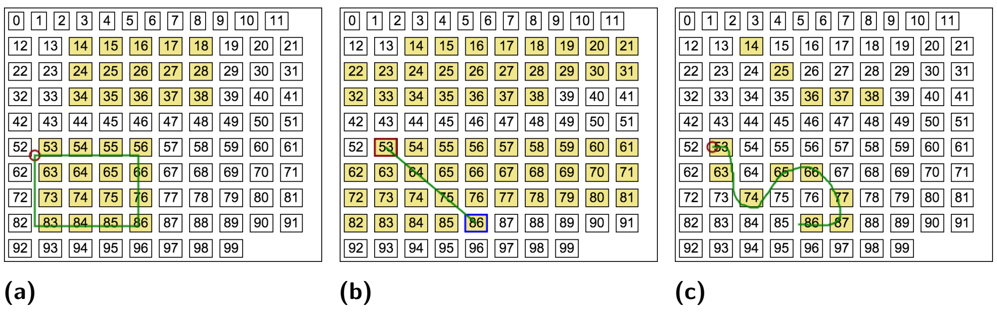


**Figure 8** Snapshots of selecting using three different selection geometries: (a) rectangular, (b) row-wise, and (c) "snake" (all elements that touch the selection path are included in the selection domain). User's mouse events were the same in each case: click at element 14, shift-click at 36, shift-click at 38, command-click near or at 53, and mouse drag to near or at 86. The rubber band indicator is visible because the snapshots were taken prior to releasing the mouse to end the drag.

variables of this class. Every command of the selection language is a method of SelectionState. Unlike the functions of the selection language that each map a selection state tuple to a new selection state tuple, SelectionState class's methods modify the encapsulated state, as is conventional in object-oriented libraries. The implementation includes a few optimizations:

- The current selection state of an element $i$ is $op(s)(i)$, where $op$ is the current composition and $s$ the base selection mapping. If $op$ contains many primitive selection operations, evaluating $i$'s selection state would require many function calls. For example, if the composition object represents the value $\mathsf{op}^f_{J_1} \circ \mathsf{op}^g_{J_2} \circ \mathsf{op}^h_{J_3}$, the straightforward way to compute $(\mathsf{op}^f_{J_1} \circ \mathsf{op}^g_{J_2} \circ \mathsf{op}^h_{J_3})(s)(i)$ is $\mathsf{op}^f_{J_1}(\mathsf{op}^g_{J_2}(\mathsf{op}^h_{J_3}(s)))(i)$. If, however, $f$ is a constant function ($\lambda x.\,\mathsf{T}$ or $\lambda x.\,\mathsf{F}$) and $i \in J_1$, then it suffices to compute $\mathsf{op}^f_{J_1}(i)$. As another example, if $i \notin J_1$ and $i \notin J_2$, then it suffices to compute $\mathsf{op}^h_{J_3}(i)$. The implementation of the composition maintains additional information about which selection functions are constant and which selection domains include a particular index, so that unnecessary evaluations like the ones above are avoided.
- The semantics of shift-clicking guarantees that the effect of two consecutive shift-clicks at points $p_1$ and $p_2$ is the same as first extending the selection path with $p_1$, then shift-clicking at $p_2$. The library takes advantage of this property when many shift-click events happen in rapid succession, e.g., during a rubber band selection, and coalesces several consecutive shift-click calls into one so that the selection domain is not computed after every mouse location visited during a drag.

The size of the selection state object is proportional to the total number of elements in all of the primitive selection operations' domains. The number of primitive selection operations is limited by the number of allowed undo states set by the programmer.

We chose to maintain the composition of primitive selection operations as functions, which is faithful to the presented formalism. Another feasible alternative is to store, as sets of indices, the "diffs" between the applications of each two consecutive primitive selection operations. To represent selection domains and the baked selection mapping, we use JavaScript's Map container, where each element's selection status is a separate entry. In some applications (selecting pixels in images, for example), other representations, such as bitvectors or run-length encoded sets, may lead to a smaller memory footprint and faster manipulation of selections. The library is parameterized on the above two aspects, that is, on how the composition and selection domains are represented. These parameterizations are purely for performance reasons and do not affect semantics. They are not discussed in this paper.

## 5.2 Selection Geometry Classes

Selection state objects are parameterized by a selection geometry object that captures all context-specific aspects of multi-selection. A selection geometry object thus has a corresponding method for each of the functions sdom, m2v, step, default, and |, introduced in Section 4, as well as for selecting elements based on a predicate:

- selectionDomain(path) to compute a set of indices from the selection path;
- m2v(point) to translate mouse coordinates to selection space;
- step(direction, point) to determine the cursor movement after an arrow key press;
- defaultCursor(direction) to provide the default cursor value for each direction;
- extendPath(path, point) to append point to path; and
- filter(predicate) to find the set of indices that satisfy predicate.

Figure 9 shows a complete implementation of a selection geometry for the case of rectangular elements that can be positioned arbitrarily. The selectionDomain function iterates over all elements and returns those that intersect with the rectangle formed by the anchor and active end. If the selection path has exactly one element, at most one element is returned. This is so that a click or command-click only selects the topmost element when the clicked point is on two or more overlapping elements. We assume the topmost element is the one with the smallest index. The m2v function is the identity function, since the selection space and mouse coordinates coincide. The extendPath function discards the selection path's intermediate points, since only the first and last points (the anchor and active end) are used in computing the selection domain. The step and defaultCursor are not be needed since keyboard commands are not supported; we include stubs for them for completeness. In practice, geometry classes would inherit from ms.DefaultGeometry, which provides default definitions that apply for many geometries. For RectangularGeometry it would suffice to implement just the selectionDomain method, and filter if predicate-selection needed to be supported.

One might want to optimize selectionDomain so that it does not iterate over all elements on every call. E.g., the selectable area could be divided into segments and the iteration limited to the elements in segments that intersect with the selection path. Also, the library makes the previous selection domain object available to selectionDomain, so that selection domain computations can be incremental. We do not discuss these aspects further in this paper.

Figure 10 shows another selection geometry class. A geometry like this could be used to implement multi-selection in a list box, Finder's list view, list of emails on a mail client, etc. The selection space coordinates are now indices of the elements in the elements array. The selectionDomain function thus returns the elements within the range between the anchor and the active end. The m2v function finds the index of the element that contains a given mouse coordinate. Clicks outside of all elements might be possible; we choose to return the selection space coordinate **null** for such mouse coordinates, and define extendPath to ignore **null** coordinates. Otherwise extendPath's implementation is the same as in RectangularGeometry above. To implement keyboard navigation, the step method recognizes the ms.UP and ms.DOWN directions, and decrements and increments the keyboard cursor respectively. The defaultCursor implements cursor defaults so that up-arrow starts from the bottom and down-arrow from the top. Filtering is as in RectangularGeometry.

```
function RectangularGeometry(elements, bbox) {
  this.selectionDomain = function(path) {
    var J = new Map();
    var r1 = makeRectangle(ms.anchor(path), ms.activeEnd(path));
    for (var i=0; i<elements.length; ++i) {
      var r2 = bbox(elements[i]);
      if (rectangleIntersect(r1, r2)) { J.set(i, true); if (path.length == 1) return J; }
    }
    return J;
  };
  this.m2v = function(p) { return p; };
  this.extendPath = function (spath, p) { if (spath.length == 2) spath[1] = p; else spath.push(p); };
  this.step = function (dir, p) { return undefined; };
  this.defaultCursor = function(dir) { return undefined; };
  this.filter = function (pred) {
    var J = new Map();
    for (var i=0; I<elements.length; ++i) { if pred(i) J.set(i, true); }
    return J;
  };
};
```

**Figure 9** An implementation of a selection geometry for selecting arbitrarily positioned rectangular elements. The constructor's elements parameter is the indexed family of elements, implemented as an array. The bbox function is assumed to compute a bounding box given an element. The helper functions makeRectangle and rectangleIntersect are what their names suggest. The ms namespace prefix indicates functions and constants in our library's API. Their names explain their purpose.

## 5.3 Visualizing selections

How selections are visualized is application dependent, when these visualizations take place is controlled by the library. Concretely, SelectionState's methods invoke a callback when the elements' selection states may have changed. Assuming CSS code defines suitable rendering, the callback can simply toggle elements' class attribute based on their selection state:

```
function refresh(sel) {
  var s = sel.selected();
  for (var i=0; i<elements.length; ++i) elements[i].className = s.get(i) ? "selected" : "unselected";
};
```

The parameter to the callback is the selection state object itself, whose selected method gives the currently selected indices. A selection state object can be requested to track changes to the elements' selection state. The refresh function will then receive the set of changed indices as a second argument.

## 5.4 Constructing a selection state object and registering event handlers

Assuming elements is an array of DOM objects, we can construct the selection geometry, then use that and the refresh function to construct a selection state object:

```
var geometry = new RectangularGeometry(elements, function(e) { return e.getBoundingClientRect(); });
var selection = new SelectionState(geometry, refresh);
```

To put the selection object to use, what remains to do is to recognize mouse and keyboard events and handle them by invoking selection's methods. The same mouse operations are often harnessed for many different tasks, which makes their event handling code complex and difficult to reuse. For example, a mouse click may invoke a command (such as opening

```
function VerticalListGeometry(elements, bbox) {
  this.selectionDomain(path) {
    var J = new Map(), a = ms.anchor(path), b = ms.activeEnd(path);
    for (var i=Math.min(a, b); i<=Math.max(a, b); ++i) J.set(i, true);
    return J;
  };
  this.m2v = function(p) {
    for (var i=0; i<elements.length; ++i) if (pointInRectangle(p, bbox(elements[i]))) return i;
    return null;
  };
  this.extendPath = function (spath, p) {
    if (p == null) return;
    if (spath.length == 2) spath[1] = p; else spath.push(p);
  };
  this.step = function (dir, p) {
    if (dir == ms.UP) return Math.max(p − 1, 0);
    if (dir == ms.DOWN) return Math.min(p + 1, elements.length−1);
    return p;
  };
  this.defaultCursor = function (dir) {
    if (dir == ms.UP) return elements.length − 1;
    if (dir == ms.DOWN) return 0;
  };
  this.filter = function (pred) {
    var J = new Map();
    for (var i=0; I<elements.length; ++i) { if pred(i) J.set(i, true); }
    return J;
  };
};
```

**Figure 10** An implementation of a selection geometry for vertically stacked elements (a list box, for example). The conventions are the same as in Figure 9. Here, we rely on one additional helper function, pointInRectangle, and use the constants UP and DOWN from our library.

a file), start a drag-and-drop operation, or indicate a selection. Distinguishing between different commands can be a nuanced task: a *mousedown* event followed by a *mouseup* could mean selection if the mouse did not move (too much) in between, otherwise a start of a drag-and-drop. We thus expect that the event handling that triggers selection commands may require some application-specific code. The SelectionState class's onSelected(p) method that determines whether the selection space point p is on a selected element or not helps in writing this code. For example, the method makes it easy to distinguish between a *mousedown* event on a selected element and a *mousedown* event elsewhere. The former is usually a possible start of a drag-and-drop of the selected elements and the latter a start of a selection.

Once the mouse interaction has been interpreted to mean a selection command, the code that effects the command is trivial: a call to a method in the SelectionState class. Figure 11 shows possible implementations of the event handlers for the *mousedown*, *mousemove*, and *mouseup* events. For simplicity, we assume that drag-and-drop is not supported. Keyboard event handlers are equally straightforward. They merely have to recognize the set of supported key presses and map each of them to an appropriate method call. To give one example, the press of a command-modified down-arrow would call selection.cmdArrow(ms.DOWN).

The handlers may also call functions to draw or clear the anchor, cursor, and rubber band indicators as appropriate. As these aspects vary from one selection context to another, implementing these functions are the application programmer's responsibility—the library merely makes readily available the necessary information, the selection path and cursor. As

```
function mousedownHandler (evt) {
  var p = selection.geometry().m2v(getMousePos(evt));
  switch (ms.modifierKeys(evt)) {
    case ms.NONE: selection.click(p); break;
    case ms.SHIFT: selection.shiftClick(p); break;
    case ms.CMD: selection.cmdClick(p); break;
  }
  drawAnchor(selection.selectionPath()); drawCursor(selection.cursor());
  document.addEventListener('mousemove', mousemoveHandler);
  document.addEventListener('mouseup', mouseupHandler);
}
function mousemoveHandler (evt) {
  var p = selection.geometry().m2v(getMousePos(evt));
  selection.shiftClick(p);
  drawCursor(selection.cursor()); drawRubber(selection.selectionPath());
}
function mouseupHandler (evt) {
  document.removeEventListener('mousemove', mousemoveHandler);
  document.removeEventListener('mouseup', mouseupHandler);
  clearRubber();
}
```

**Figure 11** An example implementation of mouse event handlers for multi-selection: selection is the selection state object, getMousePos extracts the mouse position from the event data, ms.modifierKeys inspects the event data and determines whether the shift or command modifiers were held down, and the draw* and clear* functions do what their names suggest. The current geometry is accessible as selection.geometry(). So that the handlers are not too aggressive, only the handler for *mousedown* is registered continuously, the others are activated at *mousedown* and deactivated at *mouseup*. We omit event handling boilerplate that stops event propagation and prevents default actions.

explained in Section 5.1, the selection domain may not be updated for each mouse move and shift-click event. By invoking the cursor/anchor/rubberband drawing functions directly in the handler functions, these visual indicators are, however, always updated instantaneously.

## 5.5 Summary

The recipe to implement multi-selection in a particular context is as follows: (1) select a data structure for the indexed family of elements; (2) implement a "refresh" function to visualize the selection state of the elements; (3) implement a selection geometry class; (4) implement functions to draw the anchor, cursor, and rubber band; and (5) register event handlers for mouse and keyboard events and map them to calls to a selection object's methods.

Many of the "implement" tasks can be expected to be "reuse" tasks: visualizing selected/unselected status is perhaps nothing more than a function call to change the style or color property of an element; some of the geometry's functions could be reused from another geometry and the code for drawing the anchor and cursor indicators from another selection context; and the same event registration code can likely be used repeatedly.

In the best case scenario, implementing multi-selection for a given context means selecting the geometry and visualization functions to use, and turning the feature on. But even if nothing but the SelectionState class can be reused, the selection library and its abstractions simplify the application programmer's work in a fundamental way. Each item above is a well-specified and confined task. Each task involves defining one or more functions that perform a (usually straightforward and side-effect free) computation from inputs to outputs, realize a visual effect, or dispatch events to pre-defined handler functions. Such pieces of code are predictable to develop and testable. There is no need to think about how selections change, how they should be updated, undone or redone; where anchors or cursors are located, when they should be moved, cleared, or drawn; or how or if mouse clicks, mouse dragging, and keyboard selection should work together. The findings described in Section 2 are evidence that these less tangible aspects are where the trouble lies in today's selection implementations.

## 6 Related work

One of the early guidelines for uniform multi-selection was for the Apple Lisa's user interface [1]. The operations of shift-click and mouse drag for selecting contiguous regions were defined, though not in much detail. The document that perhaps set the baseline for modern multi-selection, on Macintosh at least, is Apple's *Macintosh Human Interface Guidelines* [2, Ch. 10]. The notion of an anchor, how shift-click extends the selection by setting the active end of a range, interplay between mouse clicks, dragging, and keyboard selection operations are explained. Command-click is just presented as a toggling operation, not as establishing a new anchor, and nothing is said about restoring the selection state when the current selection shrinks. Similar guidelines for Windows [12] established File Explorer's selection behavior, though using shift-control-click for extending the selected range without disturbing other selections was not mentioned; that operation was bound to shift-click.

During the past three decades, instead of further progress towards a consistent and uniform multi-selection feature, the baseline set in the 80's has gradually deteriorated.

Today, there is no shortage of implementations of multi-selection features. Practically every GUI framework includes widgets for lists, tables, or trees of selectable items. The implementations, however, are intrusive and tightly coupled with a particular view or widget class. Typically, the programmer must wrap elements as a collection of a particular type of objects and place those objects in a particular view object, in order to get the "stock" multi-selection behavior "for free". Unless this is done, and in many cases it is not feasible due to the limitations of the provided view classes, the programmer is largely on their own.

For example, Apple's NSTableView and NSCollectionView classes provide Finder-like[2] list and icon multi-selection behavior [3], and for collections that fit to these views, their use is convenient. Their selection behavior, however, exhibits the problems described in Section 2. The corresponding iOS classes, UITableView and UICollectionView, are less useful, as their stock multi-select functionality is so minimal; as discussed in Section 2.6, each selected element requires an individual tap. But even if the multi-selection behavior provided by these classes were perfected, the main problem remains: the selection functionality is only available to the views implemented as instances of these specific classes. For example, Adobe Revel provides a "slot machine" view of images, where images are arranged in rows, each row representing an event. Each row can be of a different length and each row scrolls independently of other rows. We do not think it is possible to model this view in any of the predefined view classes.[3]

The situation is essentially the same with other widely used GUI frameworks. JavaFX's ListView and TableView have a pre-packaged selection feature. To build your own, the library provides the MultipleSelectionModel class [13], but it is not much more than an array of booleans where value changes trigger observable events. There are methods to select an individual

[2] The behavior is not exactly the same in icon view, since Finder itself does not use NSCollectionView.
[3] More generally in Adobe's software, which uses several GUI frameworks and Adobe's own platform-independent widgets, the lack of a reusable multi-selection feature has lead to many re-implementations. Symptoms are visible: e.g., the sequence of commands in Figure 1a would lead to different selections in Photoshop's "Paths" and "Layers" panel, and they would be different from the one produced in Finder.

index or a range of indices, and to clear all selections. The TableSelectionModel class adds the ability to arrange and select indices in a grid. There is a notion of the "current index", but no notion of an anchor or cursor that would not be an index of an element. Further, the current index's update rules are questionable at times: selecting a range sets the current index at the end of the range, inviting the programmer to implement a selection feature that treats range selections that grow downwards or right differently from those that grow upwards or left. There is no notion of a selection path that could support lasso-like selection tools, no notion of an active domain, and no support for implementing an undo operation.

Qt's [16] QListView, QTableView, and QTreeView classes provide a pre-packaged selection feature. The QItemSelectionModel class serves the same role as JavaFX's MultipleSelectionModel, and has most of the same limitations. QItemSelectionModel, however, supports deselecting regions (rectangular areas in a grid). It also separates the "current selection" from the permanent (baked in our terminology) selection, which is useful for implementing a shift-click feature that remembers the selection state under the active domain, as in our model.

The story with Web GUI frameworks is similar. Taking one popular example, jQuery UI library [15] provides "Selectable" as one of its user interface interaction widgets. Its multi-selection feature of DOM elements supports rectangular rubber band selection and toggling with command-click. Shift has no function and keyboard selection is not supported. Command-click on a selected element is quite surprising: the element is first deselected, but even the smallest mouse move reselects the just deselected element, and starts a rubber band selection. The library is of little use when the pre-packaged selection is not adequate.

To summarize, today's GUI frameworks provide some widgets and views with multi-selection, but they do not provide a reusable multi-selection feature. At best they provide a set of utility classes that can be useful building blocks in implementing such a feature. They leave arguably the most difficult tasks to the programmer: gaining a thorough understanding of how multi-selection should work and mapping that understanding to an implementation.

That our work models multi-selection formally differentiates it further from prior works on multi-selection. More generally, modeling user interaction and interfaces formally has not found a particularly strong foothold in practical GUI programming. This point has been made by many over the years [7, 17, 5], before introducing new techniques aimed at changing the status quo. The result is a diverse set of approaches, none in the mainstream. We do not propose a new approach for formally modeling user interfaces, but instead apply common tools and notations from the domain of programming languages for defining the operational semantics of an abstract "multi-selection machine".

Developing the formal model in the manner we did certainly required effort and iterations, but the effort is justified by the prospect of reusing the result widely and by the resulting clear understanding of the modeled feature. We suspect that there are other programming challenges in user interfaces that would benefit from an analogous formal inspection; some prior work on those lines exists. Krishnamurthi et al. [9] model the user's interaction with a web browser communicating with a server. They recognize that defects manifest frequently when users mix the browser's page navigation tools with those provided by a web application. Practical evidence shows that coordinating web interaction is clearly complex enough that programmers do not consistently implement it correctly. Analogously to multi-selection, a formal model clears the confusion and provides a precise specification. Concretely, their "language transition relation" serves a similar role than our selection language.

As another example of formally modeling oft-occurring user interface components, Zhang [17] defines models for buttons, menus, textual input fields, etc. combining modeling interaction with finite state machines and algebraic specification. Formal models of user interaction are often finite state machines and use notation designed for their concise representation (See Bowen et al. [5] for a survey of many techniques). Our selection language can also be seen as a definition of a state machine, but the abstract machine view we chose is more useful; the state changes are conveniently captured by changes to the selection state tuple and enables also algebraic reasoning (e.g., for developing optimizations).

The early PIE-models [6] would offer a setup for inspecting properties such as the reachability of all selection states, and aspects of undo, but these questions in our model are rather straightforward.

Finally, we note that there is a wealth of CHI literature about innovative means of selecting groups of elements with different kinds of pointing devices. A recent article by Strothoff et al. [14] contains a summary and categorization of such works. The connection to our work is tangential; our contribution is not in novel means of selection, but rather in how to cost-effectively and correctly implement the widely known and established multi-selection features. Perhaps our work could help in designing and implementing novel selection schemes.

## 7 Conclusion

Multi-selection is a mundane feature, yet evidently too complex to design well and implement correctly as part of a typical application development project. Practically every implementation manifests at least minor problems, some are flawed in major ways, and all are different. Even the most established applications do not stand up to close scrutiny.

We would like to see this change. Users should be able to rely on a full-fledged, correct, and uniform multi-selection feature in all contexts where selecting many elements would be useful. Consequently, this paper describes the abstractions and software architecture that makes multi-selection reusable. With a reusable multi-selection library, practically any kind of collection of elements can be made selectable with little programming effort, and every context where selection is supported will provide the same rich set of selection commands.

This article is not meant to discourage innovation and experimentation in how best to interact with collections of elements. Redesigning and reimplementing the same features many times over with predictably unpredictable outcomes, however, is an effort misplaced.

## References

1 Apple Computer, Inc. Lisa user interface starndards. Internal technical document, September 1980. URL: `www.guidebookgallery.org/articles/lisauserinterfacestandards`.

2 Apple Computer, Inc. *Macintosh Human Interface Guidelines*. Addison-Wesley, USA, 1992.

3 Apple Inc. Mac Developer Library, 2015. URL: `developer.apple.com/library/mac/navigation`.

4 Apple Inc. OS X El Capitan: Select items, 2015. URL: `support.apple.com/kb/PH21888`.

5 Judy Bowen and Steve Reeves. Formal models for user interface design artefacts. *Innovations in Systems and Software Engineering*, 4(2):125–141, 2008.

6 A. J. Dix and C. Runciman. Abstract models of interactive systems. In *HCI'85: People and Computers: Designing the Interface*, pages 13–22. Cambridge University Press, 1985.

7 Michael D. Harrison and H. Thimbleby, editors. *Formal Methods in Human Computer Interaction*. Cambridge University Press, 1990. Chapter 1.

8 John Karat, James E. McDonald, and Matthew P. Anderson. A comparison of menu selection techniques: Touch panel, mouse and keyboard. *International Journal of Man-Machine Studies*, 25(1):73–88, 1986.

9 Shriram Krishnamurthi, Robert Bruce Findler, Paul Graunke, and Matthias Felleisen. *Interactive Computation: The New Paradigm*, chapter Modeling Web Interactions and Errors, pages 255–275. Springer, Berlin, Heidelberg, 2006. `doi:10.1007/3-540-34874-3_11`.

10 Jonathan Lazar, Adam Jones, Mary Hackley, and Ben Shneiderman. Severity and impact of computer user frustration: A comparison of student and workplace users. *Interacting with Computers*, 18(2):187–207, March 2006. `doi:10.1016/j.intcom.2005.06.001`.

11 Microsoft. Select multiple files or folders. MS Windows support topic. Accessed: 2015-11-31. URL: `windows.microsoft.com/en-us/windows7/select-multiple-files-or-folders`.

12 Microsoft. *Windows Interface Guidelines for Software Design.* Microsoft Press, Redmond, WA, USA, 1st edition, 1995.

13 Oracle. JavaFX, MultipleSelectionModel. Accessed: 2015-12-07. URL: `docs.oracle.com/javase/8/javafx/api/toc.htm`.

14 Sven Strothoff, Wolfgang Stuerzlinger, and Klaus Hinrichs. Pins 'n' touches: An interface for tagging and editing complex groups. In *Proc. of the 2015 Int. Conf. on Interactive Tabletops & Surfaces*, ITS '15, pages 191–200, New York, NY, USA, 2015. ACM.

15 The jQuery Foundation. jQuery user interface. Accessed: 2015-12-03. URL: `jqueryui.com`.

16 The Qt Company. Qt Documentation. Accessed: 2015-12-03. URL: `doc.qt.io`.

17 Weishi Zhang. *Formal description and development of graphical user interfaces.* Utz, 1996.