

BACHELOR THESIS  
COMPUTING SCIENCE



RADBOD UNIVERSITY

---

**The anatomy of the modern window manager**  
A case study in developing a novel top-level reparenting list-based  
tiling window manager for X in an Agile manner

---

*Author:*  
Max van Deurzen  
s4581903

*First supervisor/assessor:*  
dr. Cynthia Kop  
c.kop@cs.ru.nl

*Second supervisor:*  
dr. Peter Achten  
p.achten@cs.ru.nl

## Abstract

At the heart of the modern personal computer experience lies the desktop environment, an implementation of the desktop metaphor comprising a variety of programs that together provide common graphical user interface components. Arguably the most complex and, in some systems, the most important component of the desktop environment, is the window manager. A window manager is system software that controls the placement and appearance of windows within a graphical user interface.

In this thesis we describe the functioning of top-level window managers. We showcase what common features window managers may consist of, and what their interaction with other system software looks like. In doing so, we will discuss the X Window System<sup>[1]</sup>, along with the Inter-Client Communication Conventions Manual<sup>[2]</sup> (ICCCM) and Extended Window Manager Hints<sup>[3]</sup> (EWMH), two standard protocols that define policy on top of the X Window System. Together, they form an environment that facilitates the interoperability between window managers and other, regular programs.

As a proof of concept, we provide and discuss *kranewm*, a complete C++ implementation of an ICCCM and EWMH compliant top-level reparenting, tiling window manager, built on top of the X Window System, using Xlib as client-side programming library. With it, we illustrate the low-level functioning of modern top-level window management. Leading up to an extensive demonstration of the product, we review several Agile software development concepts that were applied throughout, before describing the process undergone to realize the implementation.

### **Acknowledgements**

I would like to thank both of my supervisors, dr. Cynthia Kop and dr. Peter Achten, for guiding me through the research process, for the many incremental proofreads, and for the suggestions made during the development phase of *kranewm*.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Window Management</b>	<b>5</b>
2.1	Windowing systems . . . . .	5
2.1.1	Graphical user interface . . . . .	5
2.1.2	Display server . . . . .	5
2.1.3	Widget toolkits . . . . .	6
2.2	The window manager . . . . .	6
2.2.1	Responsibilities . . . . .	7
2.2.2	Stacking . . . . .	7
2.2.3	Tiling . . . . .	7
2.2.4	Compositing . . . . .	11
2.2.5	Reparenting . . . . .	12
<b>3</b>	<b>The X Window System</b>	<b>14</b>
3.1	History . . . . .	14
3.2	Client-server architecture . . . . .	14
3.3	System model . . . . .	15
3.4	Core protocol . . . . .	15
3.5	Client-side libraries . . . . .	15
3.6	Graphical environment . . . . .	16
3.7	Resources and identifiers . . . . .	17
3.8	Messages . . . . .	18
3.8.1	Events . . . . .	18
3.9	Color . . . . .	19
3.10	Graphics operations . . . . .	20
3.10.1	Images . . . . .	20
3.10.2	Text . . . . .	20
3.10.3	Exposures . . . . .	21
3.11	Fonts . . . . .	21
3.12	Input . . . . .	21
3.13	Properties and atoms . . . . .	23
3.14	The X window manager . . . . .	24
3.15	Wayland . . . . .	25
3.15.1	Architecture . . . . .	25
<b>4</b>	<b>Policy in X</b>	<b>27</b>
4.1	ICCCM . . . . .	27
4.1.1	Client properties . . . . .	28
4.1.2	Window manager properties . . . . .	30
4.2	EWMH . . . . .	35
4.2.1	Non-ICCWM features . . . . .	35
4.2.2	Root window protocol definitions . . . . .	37
4.2.3	Client window protocol definitions . . . . .	40
<b>5</b>	<b>Agile Software Development</b>	<b>44</b>
5.1	Agile Manifesto . . . . .	44
5.2	Extreme Programming . . . . .	44
5.3	Personal Software Process . . . . .	45
5.4	Personal XP . . . . .	46
<b>6</b>	<b>Implementation Process</b>	<b>48</b>
6.1	The planning game . . . . .	48

6.2	Small releases . . . . .	49
6.3	Metaphor . . . . .	49
6.4	Simple design . . . . .	50
6.5	Testing . . . . .	50
6.6	Refactoring . . . . .	51
6.7	Pair programming . . . . .	51
6.8	Collective ownership . . . . .	51
6.9	Continuous integration . . . . .	51
6.10	40-hour week . . . . .	52
6.11	On-site customer . . . . .	52
6.12	Coding standard . . . . .	52
<b>7</b>	<b>Implementation Product</b>	<b>53</b>
7.1	Design . . . . .	53
7.2	Reparenting . . . . .	53
7.3	Workspaces . . . . .	54
7.4	Sidebar and struts . . . . .	55
7.5	Key bindings . . . . .	56
7.6	Mouse bindings . . . . .	57
7.7	General usage . . . . .	57
7.7.1	Terminal . . . . .	57
7.7.2	Program launching . . . . .	57
7.7.3	Closing clients . . . . .	57
7.8	Window states . . . . .	57
7.8.1	Floating state . . . . .	57
7.8.2	Fullscreen state . . . . .	58
7.9	Window manipulation . . . . .	58
7.9.1	Inner-workspace focus movement . . . . .	58
7.9.2	Inner-workspace window movement . . . . .	59
7.9.3	Cross-workspace window movement . . . . .	60
7.10	Layouts . . . . .	60
7.10.1	Floating mode . . . . .	60
7.10.2	Tile mode . . . . .	60
7.10.3	Stick mode . . . . .	60
7.10.4	Deck modes . . . . .	61
7.10.5	Grid mode . . . . .	61
7.10.6	Pillar mode . . . . .	62
7.10.7	Column mode . . . . .	62
7.10.8	Monocle mode . . . . .	62
7.10.9	Center mode . . . . .	63
7.11	Rules . . . . .	63
7.11.1	Centering . . . . .	64
7.11.2	Autoclosing . . . . .	64
7.11.3	Workspace hint blocking . . . . .	64
7.12	Process jumping . . . . .	64
7.13	Marking . . . . .	65
7.14	Summary . . . . .	65
<b>8</b>	<b>Related Work</b>	<b>66</b>
<b>9</b>	<b>Conclusions</b>	<b>67</b>
<b>A</b>	<b>List-based Tiling Window Managers</b>	<b>71</b>
A.1	wmii . . . . .	71
A.2	dwm . . . . .	72
A.3	awesome . . . . .	73

A.4	xmonad . . . . .	73
<b>B</b>	<b>Tree-based Tiling Window Managers</b>	<b>75</b>
B.1	i3 . . . . .	75
B.2	bspwm . . . . .	76
B.3	herbstluftwm . . . . .	77
<b>C</b>	<b>X.Org Foundation Distribution</b>	<b>79</b>

# Chapter 1

## Introduction

Windowing systems and window managers alike have assumed crucial roles in guiding the look and feel of graphical user interfaces. In fact, without them, the concept of a graphical user interface would not even exist. The windowing system allows for the drawing of graphical primitives to the screen, and creates the notion of *windows*. Windows are presented on screen, and convey important information from applications to the user, but how are they to be manipulated? The windowing system merely provides drawing capabilities, and does not in itself allow for the resizing or moving of windows, or other forms of interaction for that matter. The window manager, a different subset of the desktop environment, provides this functionality. The window manager interacts with the windowing system in such a way as to allow the user to be in complete control of the windows and their contents.

Window managers come in many different shapes and forms. Some only enable users to move and resize windows, whereas others have predefined layouts along which windows can be arranged. In general, we speak of three types: *stacking*, *tiling*, and *compositing* window managers. Stacking window managers are the traditional and most widely used type, as they do not impose any constraints on the user's interaction with windows, and hence allow them to be moved around and resized freely. Tiling window managers organize windows into (mutually non-overlapping) partitions of the screen, and often allow the user to alternately activate predefined arrangement schemes. Compositing window managers, often called *compositors*, perform additional processing on the windows' buffers, possibly applying 2D or 3D (animated) effects to them. These different types of window managers are not mutually exclusive, as some or all of them may be combined in what are referred to as a *dynamic* window manager.

In most commercial system software, like Windows and macOS, both the windowing system and the window manager are indistinguishable parts of the desktop environment. These systems usually do not empower the user to replace one or other, and as such offer a fixed set of features. Most of the more popular systems employ stacking window management functionality due to its ease of use. Tiling functionality is seen as more advanced, and requires a user to anticipate an arrangement that best suits their current workflow.

The *X Window System* is the most widely used non-commercial windowing system. It does not man-

date a particular kind of window manager to be running and thereby allows its users to swap out the window manager if they see fit, facilitating customizability. To make such customizability possible, the X Window System does not define policy, only mechanism. To ensure that applications are able to work well together and can communicate with the window manager, several standard protocols were designed to define a layer of policy atop of the X Window System's mechanism. The two most widely implemented protocols are the *Inter-Client Communication Conventions Manual* (ICCCM) and the *Extended Window Manager Hints* (EWMH).

The development of an X window manager is no small task. To aid in the development process, an *Agile software development methodology* can be applied. Agile methodologies assure that developers anticipate changing requirements, circumvent integration problems, and avoid defects altogether. One such methodology is *Extreme Programming*, which aims to facilitate team synergy and better software quality. To furthermore improve performance, an individual developer may benefit from the *Personal Software Process*, which helps bring discipline to the way they develop software, and tracks predicted versus actual development progress. These two techniques have together been used to design an Agile methodology geared towards one-person teams, called *Personal Extreme Programming*.

In this thesis, we aim to answer several questions. How do modern top-level window managers allow for the displaying and positioning of, and user interaction with programs' graphical components? What are a window manager's rights and responsibilities? What is the X Window System, and how is policy defined on top of it? What does a window manager in X look like, and are there alternatives to X?

As we shall see, all of the aforementioned come together nicely as we discuss *kranewm*, an implementation of an ICCCM and EWMH compliant reparenting, tiling X window manager, all the while applying the Personal Extreme Programming methodology. In particular, we aim to show what a modern tiling window manager looks like, and what the process gone through to realize such an implementation entails. We finally aim to assess the applicability and effectiveness of the Personal Extreme Programming methodology in its strict application throughout the development process of *kranewm*.

# Chapter 2

## Window Management

As its name would suggest, a window manager is a program that manages windows. That is, it controls the placement and appearance of programs' graphical components in a graphical user interface. It is generally the most important part of a collection of programs together implementing the desktop metaphor, called the desktop environment<sup>[4]</sup>.

Before we begin our discussion on window management (Section 2.2), we must introduce the topic of windowing systems.

### 2.1 Windowing systems

A windowing system is system software that realizes a graphical model that is suitable for constructing and presenting graphical user interfaces<sup>[5]</sup>. Its main task is to assign display real-estate to currently running applications' graphical components<sup>[5]</sup>. These real-estate assignments usually take the form of resizable, rectangular surfaces of the display, or *windows*, used by the applications to present their own graphical interfaces to the user<sup>[5,6]</sup>. Windowing systems provide the functionality to construct and draw these windows<sup>[5]</sup>.

Oftentimes, windowing systems also integrate one or more widget toolkits, such as to provide developers using the system with a means to more conveniently decorate client graphical interfaces<sup>[5]</sup>.

#### 2.1.1 Graphical user interface

A *graphical user interface* (or *GUI*) is a type of computer-human interface. It solves the blank screen problem that confronted early computer users who only had a prompt to interact with<sup>[7]</sup>.

Conceptually, a computer-human interface is the point of communication between the human and the computer. The analogy can be made with a car and its steering wheel<sup>[8]</sup>. The steering wheel is the connection between the driver and the operation and functionality of the vehicle. When driving, a driver should not have to concentrate on the steering wheel, let alone the internals of the machine. Similarly, the GUI binds the user of the computer system to the operation and potential of that system. Good GUI design removes the encumbrance of communication with the computer system and allows the user to work directly on the problem at hand<sup>[7]</sup>.

In technical terms, the graphical user interface is a visual operating display that is presented on the monitor to the computer operator<sup>[9]</sup>, or, more specif-

ically, a specification for the look and feel of the computer system<sup>[7]</sup>.

The windowing system contains several important interfaces<sup>[1]</sup>. First and foremost, the *programming* interface, a library of routines and types provided in a programming language for interacting with the windowing system<sup>[1]</sup>; both low-level (e.g. line drawing) and high-level (e.g. menus) such interfaces are usually provided. Second, the *application* interface, which takes care of the mechanical interaction with the user and the visual appearance specific to an application (and thus not to the system as a whole)<sup>[1]</sup>. The *management* interface provides another mechanical interaction with the user<sup>[1]</sup>. It deals with overall control of the desktop and the input devices. The management interface typically defines how applications are arranged and rearranged on the screen, and how the user switches between application windows; individual application interfaces define how information is presented and manipulated within an application's own graphical components<sup>[1]</sup>. The *user* interface is the sum total of all application and management interfaces<sup>[1]</sup>.

#### WIMP

Graphical user interfaces often have common characteristics, such as windows, icons, menus, and push-buttons<sup>[8]</sup>. This is what is known as the *WIMP-interface*. It is a style of interaction that includes a specific set of elements of the user interface<sup>[7]</sup>.

Collectively, WIMP are pictures that bring forth a certain action or an action space. The user issues commands via the GUI to applications<sup>[7]</sup>.

Throughout the thesis, in our discussion on graphical user interfaces, we will adhere to the naming conventions described by the WIMP paradigm.

#### 2.1.2 Display server

The term *display server* is often used synonymously with *windowing system*, with the technicality that a windowing system normally incorporates a display server, and hence the former really belongs to the latter<sup>[10]</sup>. In fact, the display server is usually the main component of a windowing system<sup>[10]</sup>.

The display server interacts with the operating system and the underlying hardware to draw graphics to the screen and to relay input and output from graphical applications to and from the rest of the system<sup>[10]</sup>. Communication between the display server and its clients happens along the display server



protocol, a communications protocol that is often network-transparent<sup>[10]</sup>.

Any program that presents its application interface in a window is a client of the display server. The display server functions as an intermediary between the clients and the user<sup>[1,7]</sup>.

The display server receives its input from the kernel, that in turn gets notified by the system's input devices, such as the keyboard and the mouse. After perhaps processing an input event itself, the display server passes it off to the correct client<sup>[11,12]</sup>.

Another important responsibility of the display server is to draw the client windows' output to the computer monitor. The display server enables the user to work with multiple graphical programs at the same time, allowing each program to display its application interface in their respective window<sup>[10]</sup>.

As we shall see later on, from a developer's point of view, the windowing system, through use of its display server, implements graphical primitives, such as font rendering and the drawing of simple shapes to the screen<sup>[11,12]</sup>. It provides a layer of abstraction above the graphics hardware for use by higher-level components of the graphical user interface, such as the window manager<sup>[11,12]</sup>.

### 2.1.3 Widget toolkits

A widget toolkit is a library or collection of libraries containing a set of graphical control elements (or *widgets*)<sup>[5,6]</sup>. These widgets are elements of interaction used to more easily and consistently construct the graphical interface of a program—they define the application interface<sup>[5,6,8]</sup>.

User interface libraries such as *Windows Presentation Foundation* (for Windows-based applications), *GTK+* and *Qt* (cross-platform), and *Cocoa* (for macOS), contain a collection of controls and the logic to render them<sup>[6]</sup>.

Examples of widgets include sliders, buttons, and even windows themselves (for a window can contain multiple other windows)<sup>[5,6,8]</sup>.

Each widget facilitates a specific type of human-computer interaction, and appears as a visible part of the application's graphical user interface, as defined by the theme and rendered by the toolkit's rendering engine<sup>[8]</sup>. A rendering engine is nothing more than another programming interface that uses the rudimentary hooks provided by the display server to draw, or *render*, its elements on the screen. The theme makes all widgets adhere to a unified aesthetic design and creates a sense of overall cohesion. Some widgets support interaction with the user (e.g. labels, buttons, and check boxes). Others act as containers that group the widgets added to them (e.g. windows, panels, dividers, and tabs)<sup>[6,7]</sup>.

## 2.2 The window manager

The window manager is system software that interacts with the windowing system to seize control of the display space—by taking away sizing and positioning responsibilities from individual windows—and to form a single point of interaction for the user<sup>[6]</sup>. With a window manager in place, applications need not worry about their designated area of the screen, or whether or not the user's interaction with its graphical components is consistent with how other applications handle interaction<sup>[4]</sup>. The window manager, and only the window manager, is to take care of the placement and appearance of the windows under its reign, with the user as its conductor. In essence, the window manager is in sole control of the management interface, while the applications retain control of their own application interfaces.

Few desktop environments are designed with a clear distinction between the windowing system and the window manager<sup>[5]</sup>. Proprietary system software, like Windows and macOS, generally only comprise a single monolithic, largely non-replaceable desktop environment<sup>[5]</sup>. Within it, there is often no option to change core components, such as the windowing system or the window manager<sup>[5]</sup>. In such systems, the role of the window manager is tightly coupled with the kernel's graphical subsystems<sup>[5]</sup>. The window manager is essentially an indistinguishable part of the windowing system<sup>[5]</sup>.

Window managers usually allow the user to open, close, minimize, maximize, move, and resize running applications' graphical components<sup>[4]</sup>. Other window manager functionality is the decorating of windows to indicate, for instance, which window has input focus<sup>[5,11]</sup>. Many window managers also come with or support various utilities and features, such as docks, status bars, program launchers, desktop icons, and a desktop wallpaper.

Conventionally, a system has only one window manager active at a time<sup>[1]</sup>. Depending on the windowing system's policy, however, it may be permitted to have multiple different window managers running concurrently. Because window managers assume to be in control of *all* top-level windows, having multiple active window managers often leads to problems, and thus it is generally discouraged.

Because our proof of concept (Chapter 7) is implemented to interact with X, and because the definition of the window manager within the context of X is the most stand-alone, whenever there is inconsistency between the definitions of window manager functionality within the various platforms (proprietary Windows and macOS, X), we will adhere to those expounded by the X Window System.

### 2.2.1 Responsibilities

Unlike windowing systems such as those of Windows and macOS, X does not impose a window manager, nor does it dictate how a window manager is to behave<sup>[10]</sup>. X does not mandate even a particular type of window manager<sup>[10]</sup>. Its developers have tried to make X itself as free of window management or user interface policy as possible<sup>[13]</sup>. In fact, X does not require a window manager to be running at all<sup>[10]</sup>. Applications must therefore be developed to work with any type of window manager, and with none at all. This approach is quite different from that of windowing systems on different platforms. On Windows and macOS, for instance, an application could not possibly function without the window manager. In Unity, the window manager is even responsible for rendering part of programs' application interfaces<sup>[14]</sup>.

As we shall see, window managers have varying responsibilities. Some window managers are only concerned with the arrangement of windows on screen, whereas others solely deal with effects and transition animations. Then there are window managers that incorporate the functionality of both of the aforementioned, and more. Often, the category a window manager belongs to may be vague. For instance, many stacking window managers have some (very) limited tiling features, and almost all tiling window managers have a floating state for windows not suited for tiling. This floating state can usually be toggled on a per-window basis by the user.

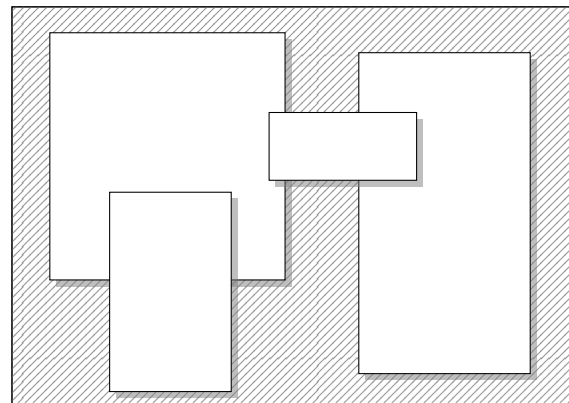
As we discuss the different forms of window management, note that a window manager can belong to just one of them, or to multiple at once. Window managers that take on the responsibilities of more than a single type of window manager are generally referred to as *dynamic* window managers.

### 2.2.2 Stacking

A *stacking* window manager (or *floating* window manager) is a window manager that allows windows to overlap, and to be moved around and resized freely by the user. In essence, no predefined arrangement is imposed by the window manager.

Windows in a stacking window management scheme can be seen as pieces of paper on a desk. Windows may (partially) overlap, and only the top-most window on a stack of overlapping windows is guaranteed to be entirely visible, given it is confined within the boundaries of the screen. Effectively, a stacking window manager attempts to fully emulate the desktop metaphor. In drawing the windows to the screen, *painter's algorithm* is most commonly used, referring to the technique used by painters,

for painting distant objects before those that are nearer<sup>[15]</sup>. Knowing which windows are to be drawn in front of or behind which others, is the window manager's task, and is often handled differently, depending on layout policy<sup>[11]</sup>.



Windows freely positioned on screen

Most commercial systems, such as macOS and Windows, incorporate this coordinate-based stacking window management scheme by default.

Depending on the implementation, stacking may be a wasteful process, requiring every window's application interface to be continually redrawn, even those of programs that are entirely covered by other windows. Because of this, many stacking window managers do not redraw windows that are not visible. Some window managers handle redrawing on a per-program basis, instructing the rerendering of a program's application interface only when that program's content has changed.

When windows are drawn over one another, the content of windows being covered is overwritten (due to single-buffer rendering). Those windows must be redrawn when they are brought back into view, or when visible parts of them change. When a window's content has changed, or when its position on the screen has changed, the window manager gets notified by the windowing system, and may decide to restack all windows, and have them all redraw their application interface. Applications that stop responding will be unable to redraw their content. This usually causes their application interface to retain parts of windows previously covering it.

### 2.2.3 Tiling

Most contemporary window managers employ multiple overlapping windows, and leave the spatial arrangement of the individual windows to the user. As a consequence, important information may be hidden in occluded windows, and users spend a significant amount of time switching between windows to reveal obscured content<sup>[16,17]</sup>.

An alternative approach to a stacking window manager and its overlapping windows is the *tiling* window manager that *tiles* windows beside each other, in a grid-like format. In a tiled window arrangement, the screen is often partitioned to be filled completely by windows. Usually, depending on the tiling scheme, no two windows will overlap each other. In certain workflows, tiling window management has proved to be superior<sup>[18,19,20]</sup>.

Whenever there are more than a few windows on the screen, managing them for effective viewing can be an annoyance to the user<sup>[21]</sup>. Users end up spending much of their time manipulating window position and structure, rather than working with the content of the windows, and on the problems at hand<sup>[21]</sup>. Tiling offers the option of automatic control of windows, benefiting users who do not want or need to arrange and rearrange windows<sup>[21]</sup>.

The desktop metaphor inherently explains the problem with a stacking approach. It is all too familiar, the problem of locating a piece of paper that is not immediately visible on a messy desk. The same situation arises when windows are hidden behind others<sup>[21]</sup>. In a tiled environment, all windows are visible and readily accessible at all times<sup>[21]</sup>.

Partially overlaid windows are often simply wasting space, causing unnecessary distraction<sup>[21]</sup>. In tiled window arrangements, screen space is always going to be allocated to some useful purpose<sup>[21]</sup>.

Certain types of windows, such as pop-up windows, dropdown menus, and fixed-size windows, are not suited for tiling. These often require a predefined position on screen, and mostly have a static application interface. Good tiling window managers will detect such windows, and have them operate in a floating state instead. If these windows were to be sent to the background, they would be fully covered by the windows in tiling mode. As such, windows in the floating state should always be rendered in front of those in tiled mode.

There are many different ways to categorize tiling window managers. Some incorporate techniques that are comparable, while others have little to nothing in common. Amongst contemporary tiling window management, two tiling techniques are generally employed: *list-based* and *tree-based* tiling<sup>[22]</sup>. The names of these techniques refer to the underlying data structures used to store and manipulate windows. It is worth noting that there are also tiling window managers that integrate both of these techniques, and there are tiling window managers that do neither.

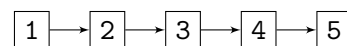
## List-based tiling

A list-based tiling window manager uses an *ordered list* to keep track of its windows. Windows are all assigned a unique number based on their position in the ordered list. The number associated with each window determines where it is to be tiled on screen. The list-based tiling window manager will have several predefined *layouts* that regulate the size and position of each window in the ordered list. Layouts do not stipulate a specific number of windows to control, and are, in that sense, dynamic. That is, adding a new window to the list will usually—depending on the layout and its policy—cause all windows to be rearranged. This is a direct consequence of the tiling philosophy: the screen is to be fully filled with windows, and no window is to be covered by any other window.

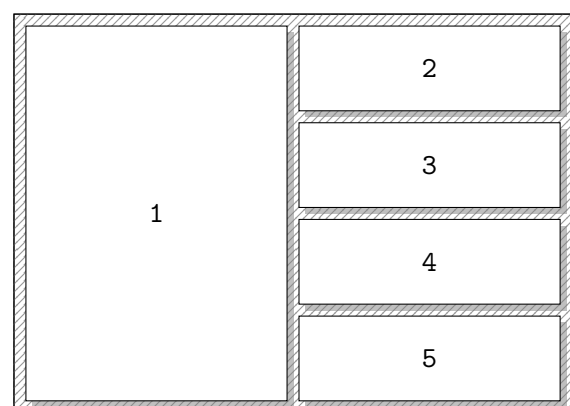
List-based tiling window managers offer users the functionality to change the currently active layout *on the fly*. This allows for highly efficient workflows, in which the user continually switches to the layout that best suits their latest needs. Users are also able to change the order of the windows in the list, therein directly manipulating their placement on screen.

Layout nomenclature is not standardized; each window manager will have their own naming scheme. The underlying concepts, however, are mostly the same.

The most common layout found in list-based tiling window managers today is *stack mode*, sometimes called *master-stack mode*, *master mode*, or *tile mode*. In it, one window—the first window in the ordered list—is appointed to be the *master* window. The master window is considered to be the most important window, in that it gets assigned the largest partition of the screen. All other windows in the ordered list—those deemed *non-master* windows—are part of the so-called *stack*.



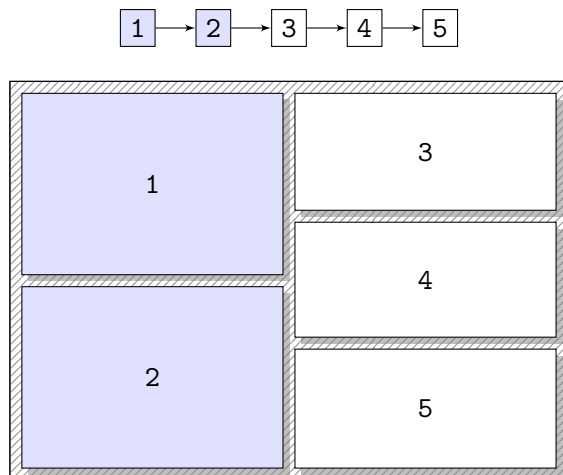
Ordered list containing 5 windows



Window arrangement on screen

Master windows fill the left half of the screen, while windows in the stack are *stacked* on top of each other on the right. Non-master windows all take up an equal part of the portion of the screen devoted to the stack.

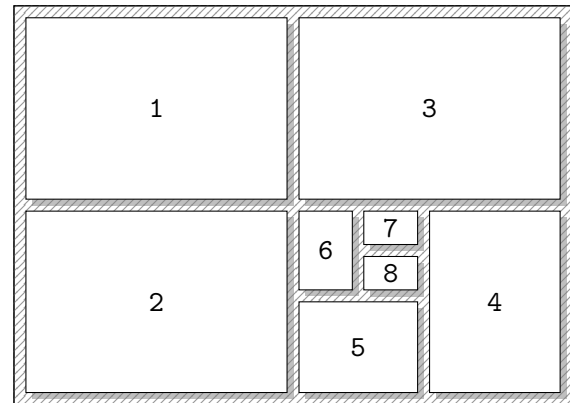
A variant of the stack mode layout is *n-master mode*. Window managers employing this layout will often provide the ability to dynamically change the number of master windows in stack mode. For example, if  $n = 2$ :



Just as in stack mode, non-master windows all have the same size. When there are multiple master windows, those too take up equal parts of their dedicated portion of the screen. If ever there are no master windows, or no windows in the stack, all windows in the ordered list are stretched along the entire width of the screen.

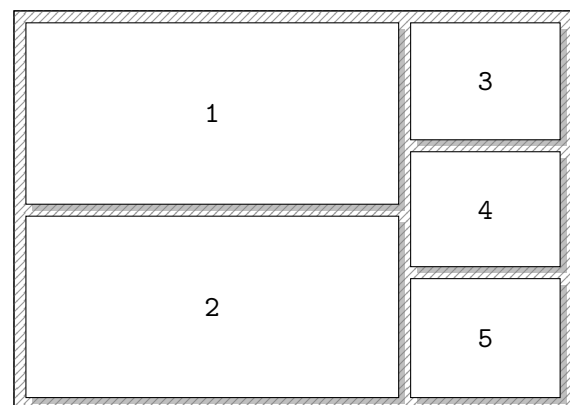
A third common layout is aptly named *monocle mode*, *fullscreen mode*, or sometimes *max mode*, in which all windows in the ordered list fill the entire screen. Of course, technically, this does not comply with our definition of tiling, for *all* windows overlap each other. Notwithstanding, this layout can prove tremendously useful, especially in situations where every program that is being worked with has a more involved application interface.

Some window managers allow its users to define their own layouts. Usually, this type of customization is more geared towards developers, as adding layouts mostly involves altering source code. Most layouts will revolve around the master-stack disposition, allowing users to dynamically change the number of master windows, like in *n-master mode*, and updating the geometry of the windows on screen accordingly. An example of such a custom layout, *spiral mode*, with two master clients:



There are many more layouts employed amongst the vast choice of list-based tiling window managers.

It is common for list-based tiling window managers to offer the ability to dynamically change the so-called *master factor*, or, the factor of master portion width to screen width. In the examples presented above, in which there are both master clients and non-master clients, the master factor is 50%.



Master factor of 70%

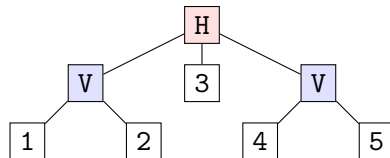
When a new window is added to the environment, most list-based tiling window managers will prepend it to the ordered list. Some are implemented to insert the new window behind the window that currently has input focus, and others will append it to the list.

List-based tiling can be extremely useful if the available layouts tailor exactly to the user's every whim. If they do not, it can become a frustration, and even a deterioration of productivity. This inflexibility has prevented mainstream adoption, as list-based tiling has only seen popularity amongst developers and hobbyists.

Refer to Appendix A for a discussion on and comparison between contemporary list-based tiling window managers.

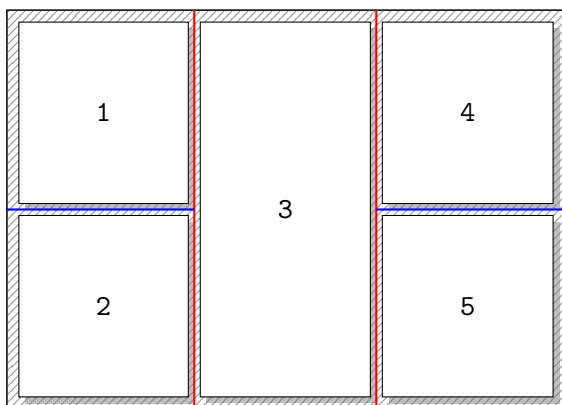
## Tree-based tiling

Tree-based (or *manual*) tiling window managers internally represent windows as leaves of a tree. Internal nodes and the root of the tree are *dividers* (sometimes called *split containers* or *splits*). Dividers recursively split up the set of windows and dividers passed along by its parent into constituents. For this reason, the tree can be seen as a *nested container*. Dividers lay out their direct children into either a horizontal local arrangement, or a vertical one. Consider the following tree.



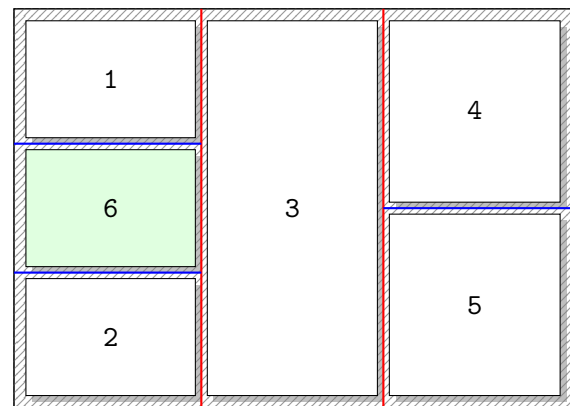
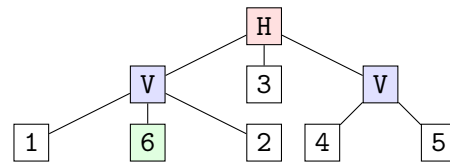
Tree with 3 dividers and 5 windows

It corresponds to the following window arrangement on screen.

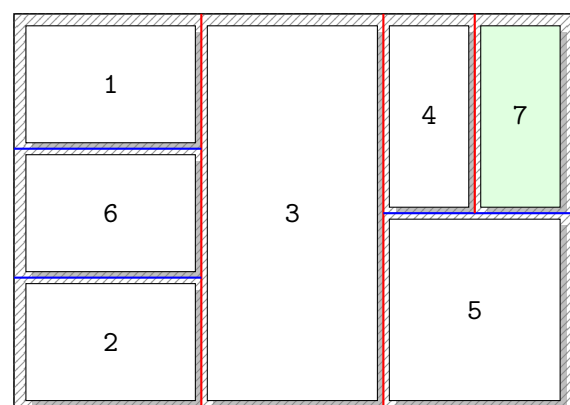
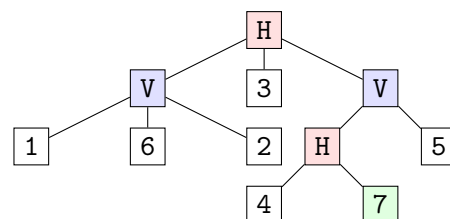


Here, the red lines represent the horizontal divider specified by the root of the tree. It has three children, two of which are themselves dividers. It divides the screen into three equally sized partitions that are evenly distributed amongst its children. The blue lines represent the two vertical dividers; both have two children, all of them being windows. These two dividers hence each divide their assigned portion of the screen into two. Windows fill the portion of the screen allocated to them by their parent (which is always a divider).

Whereas with list-based tiling, the user has little to no control over where a new window is placed, with tree-based tiling, a user can specify the exact position and local tiling arrangement. Say we wanted to add a window to the arrangement shown above, between windows 1 and 2. The user would direct input focus to window 1, and spawn the new window. The resulting arrangement is the following.



Because the local arrangement was vertical (window 1 has a vertical divider as parent), the new window inherits this, and is tiled along with its siblings in a vertical layout. The user may want to tile windows horizontally, while the local arrangement is vertical. Tree-based tiling window managers allow users to specify this, internally calling for a new divider to be created. If we wanted to add a new window to the right of window 4, we would navigate to window 4, instruct the window manager that upon adding a new window, we want a horizontal divider to be created, and we would get the following.



Often, such window managers will also allow the user to adjust the width and height of a window

within its local arrangement. And, as with list-based tiling, the user can swap the positioning of two windows within the tree (and hence, on screen).

Just as list-based tiling, tree-based tiling is deterministic, in that the representation is never ambiguous with regard to the corresponding window arrangement.

Tree-based tiling offers the user the ability to fine-tune the arrangement, essentially allowing for custom layouts to be constructed as the user sees fit. For this reason, tree-based tiling window managers tend to garner more widespread popularity than their list-based counterparts, though still not nearly as much as traditional stacking window managers do.

Refer to Appendix B for a discussion on and comparison between contemporary tree-based tiling window managers.

### 2.2.4 Compositing

A *compositing* window manager, or *compositor*, is a window manager (or a component of one) that registers an *off-screen buffer*—sometimes called a *surface* or *canvas*—for each of an application's windows. The window manager blits these window buffers into an image that represents the screen, and writes the result into display memory. Compositing has little to do with window arrangement, such as is the responsibility of stacking and tiling window managers. A window manager can be both a compositor and a stacking or tiling window manager. More commonly, following the Unix philosophy, compositors are run alongside other window managers to provide arrangement and compositing functionality separately.

To form the image shown on screen, windows' application interfaces are composited together. In order to do this, each window needs a dedicated buffer to which to draw<sup>[23]</sup>. As a result, compositors require more memory (system memory or graphics memory) for each new window that will be displayed. The primary benefit to compositing over traditional window management is compartmentalization. Each window has its own buffer to draw to, and the compositor has complete control over the final image<sup>[23]</sup>. As a consequence, artefacts typical of traditional single-buffer rendering, such as screen tearing, are not seen in compositing window managers, since the window manager does not need to wait for or request that windows repaint areas previously clipped or covered when they are moved.

Compositing window managers may perform additional processing on buffered windows, such as animated effects and fading. Computer graphics technology allows for visual effects to be rendered in real time, such as drop shadows, live previews, and com-

plex animations. If a screen is double buffered—which most modern monitors are—flickering does not occur during image updates.

Mac OS X 10.0 was the first mainstream operating system to feature software-based 3D compositing and effects, through use of its Quartz component (display server and compositor).

Microsoft first presented the Desktop Window Manager (DWM), which is Windows' compositing window manager, in Project Longhorn, at the 2003 Windows Hardware Engineering Conference. There, the compositor's *wobbly windows* feature was demonstrated. Due to delays in the development of Longhorn, Microsoft did not debut its then 3D-compositing window manager until the release of Windows Vista in January 2007.

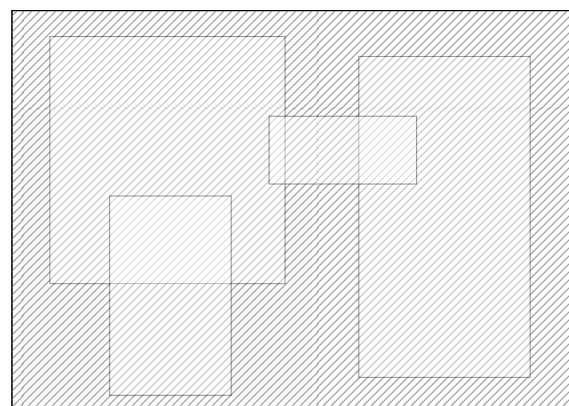
Of course, because buffers are converted to images before being rendered on the screen, any processing done to imagery can also be done to a window and its encompassed application interface. Here, we present several features commonly implemented in modern compositors.

#### Duplication

A window manager may want to duplicate images to create effects, or to mirror a window's content on another display. Compositors can achieve this merely by copying the window's buffer to another display.

#### Alpha compositing

Alpha compositing is the process of combining an image with a background to create the appearance of partial or full transparency. Alpha compositing is often used by window managers to emulate transparency on the windows of applications that otherwise do not support it.

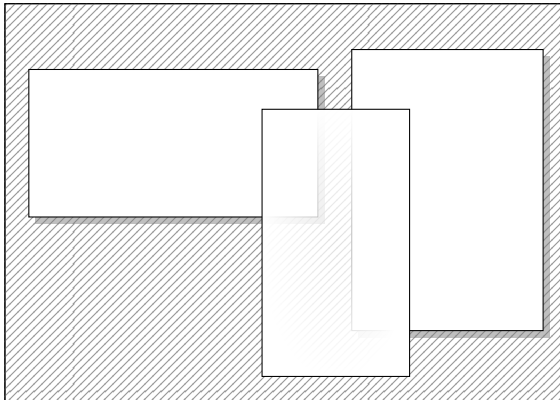


Window buffers given an opacity of 40%

#### Fading

Compositors can produce a fading effect when certain operations are performed on a window, such as

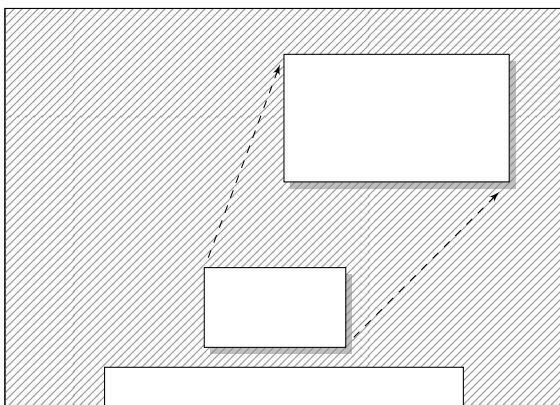
when it is opened, closed, minimized, or maximized. Upon closing a window, for instance, the compositor may perform an animation on the window, having it seemingly fade out, into the background.



A window fading out into the background

### Scaling

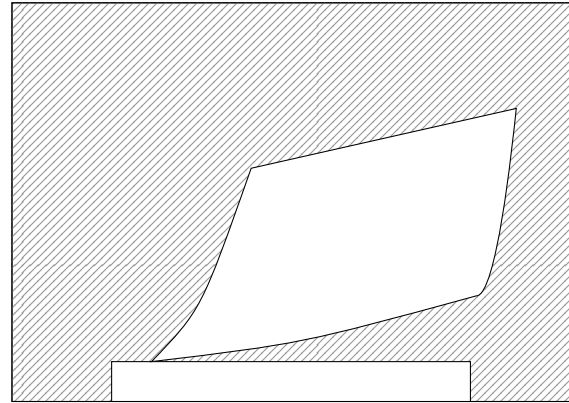
Scaling is the gradual resizing of an image. Window managers can produce zoom effects by scaling a window's buffer before it is displayed in full size on the screen. Upon opening an application from the dock, a zoom-in effect can be simulated by having the application's buffer image resized from the size of the dock icon to full scale, and by at the same time having it moved from the dock to its initial position.



A window being scaled up to full size, and moved to its initial position

### Contortion

Of course, buffer images can also be rotated and contorted. Certain effects can be simulated by gradually rotating or contorting a buffer image, such as having a window seemingly being sucked into the dock upon minimizing it, by stretching it out. macOS' genie effect is an example of this.



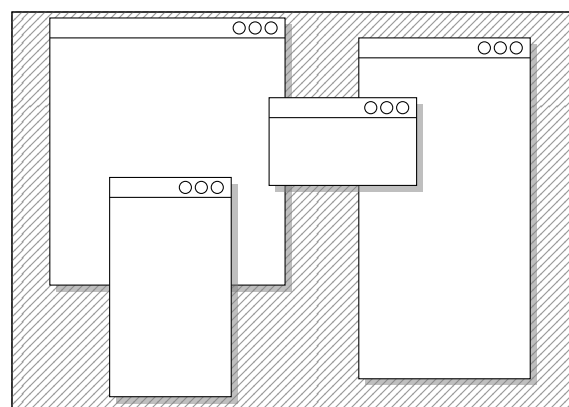
A window being stretched into the dock

### Blurring

A blur effect can be achieved on (part of) a window's buffer by applying a Gaussian function. The visual effect of this technique is a smooth blur that resembles that of viewing the image through a translucent screen. Image blurring is sometimes used by window managers to visually distinguish an input-focused window from windows that are not in focus.

### 2.2.5 Reparenting

A *reparenting* window manager is a window manager that adds its own graphical elements to a program's windows. These elements are added as an addition to the program's own application interface. DWM, Windows' compositor, is a reparenting window manager, and so is macOS' Quartz. On most proprietary platforms, applications have a border and title bar rendered around their content. Programmers that develop graphical applications for Windows or macOS, for example, need not draw these borders and title bars themselves; they are added and maintained by the window manager.



Windows given a frame that contains buttons

The term stems from the X environment, in which every window has a parent window which it is a part of. When a window manager *reparents*

an existing window, what it is really doing, is creating a new window to act as a *frame* around it. The window manager then sets this frame to be the existing window's parent, and any decoration, such as the title of the window, is drawn to the frame by the window manager. The application interface remains unaltered.

Frames may also contain control elements, such as buttons used for closing, minimizing, and maximizing the window. The program knows nothing of these control elements, and is not notified when one of them is pressed. The window manager is to respond to any interaction with them, by, for example, destroying or resizing the window.



# Chapter 3

## The X Window System

The X Window System is a network-transparent windowing system that runs on a wide variety of machines<sup>[1,24]</sup>. It is most common on Unix-like operating systems, though it is relatively straightforward to build the X.Org Foundation software distribution, which it is part of, on most ANSI C and POSIX-compliant systems<sup>[25]</sup>. Commercial implementations are also available for a wide range of platforms.

Throughout most of this chapter, we will be heavily referencing *The X Window System* by Robert Scheifler and Jim Gettys, *The X Window System, Version 11* by Jim Gettys, Philip Karlton, and Scott McGregor, and the books in the series *The Definitive Guides to the X Window System*.

The proof of concept presented and discussed in Chapter 7 is a window manager implementation that sits on top of the X Window System. Therefore, familiarity with the notions detailed in this chapter is assumed.

### 3.1 History

The X Window System was created in the mid-1980s at the Massachusetts Institute of Technology<sup>[26]</sup>. In 1988, MIT formed a member-funded consortium that was to provide the leadership required to support further development of the X Window System<sup>[26]</sup>. In 1992, MIT and the membership formed an independent organization with the purpose to move the consortium out of MIT. The rights to the X Window System were assigned to X Consortium, Inc. by MIT in 1994. In 1996, the X Consortium, Inc. closed its doors and the rights to the X Window System were transferred to The Open Group (then known as the Open Software Foundation)<sup>[26]</sup>.

In 2004, the X.Org Foundation was formed as the successor to the X.Org Group at The Open Group. The X.Org Foundation was meant to foster the development, advancement, and maintenance of the X Window System as a comprehensive set of vendor-neutral, system architecture independent, network-transparent windowing and user interface standards<sup>[26]</sup>. Membership in the X.Org Foundation is free and open to anyone<sup>[26]</sup>.

The X.Org Foundation requests that the following names be used when referring to the windowing system: *X Window System*, *X*, *X Version 11*, *X Window System Version 11*, and *X11*<sup>[1,25]</sup>.

Despite its age, the X Window System is still highly relevant. Even as the release of Wayland

(Section 3.15), another windowing system and display server protocol that is meant to replace the X Window System, draws closer, it is believed that the transition from X to Wayland may take many years, and that a great number of systems will continue to use the X Window System in the future<sup>[27]</sup>.

The X Window System servers target computers with bitmap displays<sup>[25]</sup>. The server sends user input to and accepts output requests from various client programs through a variety of different inter-process communication channels<sup>[1,25]</sup>. The most common case is for the client programs to be running on the same machine as the server, though clients can also be run transparently from other machines, including machines with different architectures and operating systems<sup>[1,25]</sup>.

X supports overlapping hierarchical subwindows, and text and graphics operations, on both monochrome and color displays<sup>[25]</sup>.

The number of programs that use X is quite large<sup>[25]</sup>. The core X.Org Foundation distribution alone includes a plethora of X programs. Refer to Appendix C for the list of programs that are part of the standard X.Org Foundation distribution.

Other user-contributed software is available in the X.Org Foundation distribution, and is otherwise available on the Internet<sup>[25]</sup>.

### 3.2 Client-server architecture

The X Window System operates in a client-server model. The *X server* controls one or more physical display devices, underlying input devices, and service requests from the clients<sup>[1,24,28]</sup>.

An application that wishes to interact with these devices takes on the role of an *X client*. Clients communicate with the server in that they issue requests and receive information (back) from the server<sup>[1,14,24,28]</sup>. Generally, every application with a graphical interface is a client.

The communication protocol used between server and clients can work over any inter-process communication mechanism that provides a reliable octet stream<sup>[1]</sup>. Of course, an X server and its clients may run on the same computer, in which case they communicate via domain sockets<sup>[1,24]</sup>. If server and clients are on different machines, communication usually happens over TCP/IP<sup>[1,24,28]</sup>.

The greatest advantage of X's design is the fact that clients only need to know how to communi-

cate with the server, and need not be concerned with the details of interacting with the underlying hardware<sup>[1,24,28]</sup>. Programmers of client graphical application interfaces are not required to deal with rudimentary graphics operations; at the most basic level, a client directs the server to “draw a line from *here* to *here*,” to “render *this* string of text, using *this* font, in the top-left corner of the screen,” or to “move *this* window to *this* position on-screen.”

However, as X’s design requires the server and its clients to operate separately, some overhead is always incurred. Most of this overhead comes from network round-trip delay time between the server and its clients (latency), as opposed to from the protocol itself<sup>[1]</sup>. As such, a common criticism of X’s design is that its network features result in excessive complexity and decreased performance when run locally.

### 3.3 System model

Several clients can have connections open to a server, and a client can have connections open to multiple servers<sup>[1]</sup>. The server is to multiplex requests from clients to the display, and demultiplex keyboard and mouse input to the appropriate client<sup>[1,11]</sup>.

The server is typically implemented as a single sequential process<sup>[1]</sup>. It uses round-robin scheduling among clients, with which many synchronization problems are trivially solved, though a multiprocess server has also been implemented<sup>[1]</sup>. Instead of being implemented as part of the kernel of an operating system in order to improve performance, a user-level server process is much easier to debug and maintain<sup>[1,11]</sup>.

The server incorporates the base window system, and it provides the fundamental resources, mechanisms, and hooks needed to implement the various user interfaces<sup>[1]</sup>. Device dependencies are handled by the server<sup>[1]</sup>. As a consequence, the communication protocol between clients and server forms an abstraction layer above the various devices<sup>[1]</sup>. Because the device dependencies are placed at one end of the network connection, all client applications are completely device independent<sup>[1,11]</sup>.

### 3.4 Core protocol

The core *X protocol* is the true definition of the X Window System<sup>[10]</sup>. Any code in any language that implements it is a true implementation of X<sup>[10]</sup>. It is designed to communicate the information required to operate a windowing system over a single asynchronous bidirectional stream<sup>[10]</sup>.

When client and server operate on the same machine, the connection relies on local inter-process

communication channels, shared memory, or domain sockets<sup>[10]</sup>. All connections, whether local or remote, use the X protocol<sup>[10]</sup>.

Clients typically implement the X protocol using a programming library that interfaces to a single underlying network protocol, such as TCP/IP<sup>[10]</sup>. Two of such programming libraries will be discussed in Section 3.5.

The X protocol was developed with the intention to provide *mechanism*, not *policy*<sup>[1,10]</sup>. Mechanism describes concrete implementation details, covering algorithms, data structures, and so on. Policy is defined at a higher level, and uses mechanism to establish rules and standards for interaction between X clients<sup>[2,13]</sup>.

Because the developers of the protocol found it likely the need would arise for changing one without affecting the other, they chose to separate mechanism from policy<sup>[1,10,13]</sup>. In fact, the X protocol says nothing of policy<sup>[1]</sup>. Instead, it is left to X client application developers to incorporate these rules and standards<sup>[2,10,13]</sup>.

This design decision proved to have both advantages and disadvantages. On the one hand, X has a highly modular architecture that has withstood numerous changes over the years<sup>[13]</sup>. On the other hand, many believe policy decisions were forced to be made in the wrong place<sup>[13]</sup>. Rather than being something end users, system integrators, or system administrators were given control over, X client program developers are to integrate policy themselves<sup>[13]</sup>. This is problematic not only to end users, who might notice discrepancies in the way different programs interpret a standard, if they even implement policy at all, but also to developers, who are burdened not only with effectuating the program’s desired behavior, but now also with making sure the program plays nicely with other clients running under the same server—such as the window manager<sup>[2,10,13]</sup>.

To provide end users with consistency amongst different programs, and to relieve developers from the burden of having to define their own policy, several conventions, rules, and standards were captured in standard protocols<sup>[13]</sup>. Two main protocols used widely today are the Inter-Client Communication Conventions Manual (ICCCM), and Extended Window Manager Hints (EWMH), which we will both discuss in Chapter 4<sup>[2,3]</sup>.

### 3.5 Client-side libraries

There exist two official C libraries for low-level client-side X programming: *Xlib* and *XCB*. They are essentially helper libraries—technically sets of libraries—

that provide an API for talking to the X server. Xlib, released in 1985, was the original X protocol client library<sup>[11]</sup>. In fact, it was the only official X protocol client library until the introduction of XCB in 2001<sup>[14]</sup>. The two libraries have significantly different design goals: while Xlib tries to hide the X protocol behind an API that serves as an abstraction layer, XCB directly exposes the plumbing beneath<sup>[11,14,29]</sup>.

This difference manifests itself most directly in how the two libraries handle the asynchronous nature of X's client-server architecture<sup>[11,14,29]</sup>. Xlib hides the asynchronous X protocol behind an API that behaves both synchronously as asynchronously, whereas XCB fully preserves X's asynchronous behavior in its API<sup>[11,14,29]</sup>.

For instance, to retrieve the attributes (size, position, etc.) of a window, one would write the following using Xlib's C API.

---

**Listing 3.1:** Xlib synchronous attribute lookup

```
XWindowAttributes attrs;
XGetWindowAttributes(dpy, win, &attrs);
// do something with attrs
```

---

Here, `dpy` is a pointer to the display structure that serves as the connection to the X server; it contains all information about that X server. `win` is the window we wish to receive attributes from, and `attrs` is the structure representing the window's attributes. Under the hood, `XGetWindowAttributes` sends a request to the X server, and subsequently blocks until it receives a response. Hence, it is *synchronous*<sup>[11]</sup>.

Doing the same using XCB's C API results in the following.

---

**Listing 3.2:** XCB asynchronous attribute lookup

```
xcb_get_window_attributes_cookie_t cookie =
    xcb_get_window_attributes(con, win);
// do something while waiting for a response
xcb_get_window_attributes_reply_t* reply =
    xcb_get_window_attributes_reply(con, cookie, nullptr);
// do something with reply
free(reply);
```

---

As in our Xlib example, we have identifiers representing the connection with the X server, `con`, and the target window, `win`. The former hides a queue of messages coming from the server, and a queue of pending requests that our client intends to send to the server<sup>[29]</sup>. XCB returns a cookie for each request that is sent. This cookie is an XCB-specific data structure that contains the sequence number with which the request was sent to the X11 server<sup>[29]</sup>. To receive any response, that cookie has to be provided, such that XCB knows which of the waiting replies is targeted<sup>[29]</sup>.

Because XCB is *asynchronous*, the function `xcb_get_window_attributes` does not block after sending a request to the X server; it returns immediately without waiting for a response. To subsequently block on receiving a response, `xcb_get_window_attributes_reply` is called<sup>[29]</sup>.

The advantage of the asynchronous approach is obvious if we consider a situation in which we need to retrieve the attributes of multiple windows at once<sup>[14]</sup>. Using XCB, we can immediately send all requests to the X server, do something that does not require a response, and then wait for all of them to return. With Xlib, we have to send one request at a time and wait for its response before we can send the next request. As such, we would expect to only block for the duration of a single round-trip to the X server using XCB, compared to multiple with Xlib.

The disadvantage of the asynchronous approach is XCB's verbose nature, and its much more involved interface<sup>[14]</sup>. As can be gleaned from our examples, the Xlib code, as by design, looks much more like an average C library call, hiding the fact that function calls result in protocol requests to a server, whereas the XCB code is a lot messier<sup>[11,14,29]</sup>.

A confusing part of Xlib, and consequently the main reason behind the creation of XCB, is its mixture between synchronous and asynchronous APIs<sup>[29]</sup>. Function calls that do not require a response from the X server are queued in a buffer to be sent as a batch of requests to the server<sup>[11]</sup>. Those that do require a response flush all the buffered requests, and then block until the response is received<sup>[11]</sup>. Effectively, this means that functions that return values (such as `XGetWindowAttributes`) are synchronous, while functions that do not (e.g. `XMoveWindow`) are asynchronous<sup>[11]</sup>.

Most higher-level applications, such as regular programs incorporating a graphical interface, should want to call Xlib and XCB sparingly<sup>[7]</sup>. Higher-level toolkits (such as GTK+) offer more efficient programming models, and often support features expected in modern applications that are not directly available when using Xlib or XCB<sup>[8]</sup>. Examples of such features are internationalized input and output, and integration with desktop environments<sup>[30]</sup>. Nevertheless, sometimes applications will find that they require a more direct interaction with the X server, as we shall see later on in our discussion on X window managers (Section 3.14).

## 3.6 Graphical environment

What is traditionally known as a window in most graphical user interfaces, is known to the X Window System as a *top-level window*<sup>[1,12]</sup>. A window

that is part of—that is, a child of—another window, is called a *subwindow* to its parent window<sup>[1,12]</sup>. Graphical elements, such as those mentioned in Section 2.1.1, can be realized using subwindows. The generic term *window* may refer to both top-level windows and subwindows. A top-level window is often referred to as a *client* to the X server. As such, a client is a single top-level window that may or may not consist of subwindows<sup>[10]</sup>.

A client may request the creation of a new window. Technically, this is a request to create a subwindow of an existing window. The reason only top-level windows are considered clients, is because of the tree arrangement of X11's window hierarchy<sup>[11]</sup>. At the root of the tree, fittingly, we have what is called the *root window*. The root window is a special window that is created automatically by the server at startup<sup>[11]</sup>. It is always as large as the screen, and lies behind all other windows<sup>[12]</sup>. In this arrangement, top-level windows are windows of which the parent is the root window. Analogously, all of the root window's direct children are the X server's clients.

When a window is moved, resized, covered by another window, or in general made partly or entirely non-visible, the window's contents may be destroyed, depending on whether or not the server is maintaining a backing store<sup>[10,12]</sup>. A client may request backing store memoization of window content, but there is no guarantee the server will obey such a request<sup>[24]</sup>. As such, a client may not assume a backing store is being maintained. Clients are sent special messages called events—which we will discuss in Section 3.8—that notify them that their content must be redrawn. Subwindows may exist beyond the boundaries of their parent. Parts of a subwindow that are not within its parent's drawing area are simply not displayed on the screen<sup>[11,12]</sup>.

All windows have an associated set of *window attributes*, comprising a window's geometry (size and position), border width and color, whether backing store has been requested for it, and much more<sup>[11]</sup>. There are also special messages in place to examine or change a window's attributes. Windows may be assigned a *window class* at the time of creation: *input-only*, or *input-output*<sup>[24]</sup>. Input-only windows cannot be used as a drawable—that is to say, they cannot be used as a source or destination for graphics requests<sup>[11]</sup>. Effectively, creating a window with this class results in a window that can never be shown on screen; it can only be used to receive input<sup>[11]</sup>. Input-output windows can be shown on screen<sup>[11,24]</sup>. They have a body, and may or may not consist of a border<sup>[24]</sup>. If set, the border spans all four edges of the body. The body has a background color,

may contain subwindows, and is otherwise used for drawing<sup>[11,24]</sup>. Every subwindow has a *depth* relative to its siblings. The depth defines which windows are rendered above or below which other windows<sup>[10,11,12]</sup>.

At any given time, a window may be visible or not, regardless of whether the window is within a drawable area<sup>[11]</sup>. *Unmapping* a window changes the state of a window to *hidden*, and prevents the X server from rendering the window and all of its contents—including subwindows<sup>[11,12]</sup>. *Mapping* a window registers it for rendering, along with all of its subwindows, unless a subwindow, in its turn, is hidden<sup>[11,12]</sup>.

In X, *drawables* collectively refer to windows and pixmaps<sup>[10,12]</sup>. A *pixmap* is an off-screen resource used for various operations, one of which is drawing to the screen<sup>[1]</sup>. The difference between a pixmap and a window is subtle. The main distinction lies in the way they are displayed. In windows, graphical output immediately becomes visible on screen; for pixmaps, a drawing must first be copied to the screen<sup>[1]</sup>. The content of a pixmap can be transferred to a window, and vice versa. A bitmap is a single bit-plane pixmap<sup>[1]</sup>. Pixmaps are an important resource in X, as they are used to define icons, cursors, background and border patterns for windows, and fill patterns<sup>[10,11,12]</sup>.

### 3.7 Resources and identifiers

All X resources—such as windows, fonts, and cursors—are stored on the server<sup>[1]</sup>. Clients use *identifiers* and *messages* (described in Section 3.8) to inform about and modify these resources<sup>[11,12]</sup>. Identifiers are nothing more than integers a client uses as a name for a resource on the server<sup>[10,11]</sup>. The *win* variables in Listings 3.1 and 3.2 are such identifiers. The most important server-side resources that are referenced client-side through a numerical identifier are the following<sup>[10,11,12,31]</sup>.

- Window
  - Pixmap
  - Font
  - Colormap
  - Graphics context
- } Drawable

When a client requests the creation of one such resource, an identifier must also be specified for it<sup>[10,11]</sup>. To create a new window, for instance, the client specifies several attributes of the window (its parent, position, width, height, etc.) and an identifier to associate with the window for further reference<sup>[10,11,12,31]</sup>.



At the protocol level, identifiers are 32-bit integers with the three most significant bits equal to zero<sup>[10]</sup>. The server assigns a set of unique identifiers to each client in an acceptance packet that a client receives upon successfully connecting with the server<sup>[1]</sup>. These identifiers may subsequently be used by the client to create resources<sup>[10,11,12,31]</sup>.

After a client has created a resource, its identifier is used to operate on it by sending requests to the server<sup>[10,11]</sup>. There are operations that modify a resource (a request to move a window, for example), and there are operations that merely ask for information about a resource (such as our examples in Listings 3.1 and 3.2)<sup>[10,11,12,31]</sup>.

No two resources ever have the same identifier; identifiers for all resources on the same server must be unique, even if those resources were created by different clients<sup>[11,12]</sup>. This is because a client is able to reference another client's resource, if it so desires, regardless of whether that resource's identifier is within the client's own assigned set of unique identifiers<sup>[11,12,31]</sup>. Concretely, this means that all clients that are connected to the same server use the same identifiers to refer to the same resources<sup>[10,11]</sup>.

Say a client creates a window with identifier 0x3e00061, and transfers this identifier to another application (using domain sockets, for example). This other application is now able to request for operations to be performed on the window associated with that identifier. This is an important feature used by many programs that require access to another client's windows (such as applications used for taking screenshots)<sup>[10,12,31]</sup>.

Resources are destroyed at the end of the life of the client that created them, unless the client explicitly instructs the server to retain them<sup>[12,31]</sup>.

## 3.8 Messages

The X protocol defines four types of messages. There are *requests*, which are sent from the client to the server, and *replies*, *events* and *errors*, all sent from the server to the client<sup>[10]</sup>.

- **Requests** are generated by the client and sent to the server. A request can carry a wide variety of information, such as the instruction to create or move a window, or an inquiry about the current position of a window<sup>[10]</sup>.
- **Replies** are sent from the server to the client. They are brought about by requests that ask for information. Requests that instruct the drawing of a line, for example, do not instigate replies, whereas requests that ask about

the current position of a window, do<sup>[10]</sup>.

- **Events** are also sent from the server to the client, though not necessarily in response to a request. Events convey information about a device action, or about a side effect of an earlier request<sup>[10]</sup>. Data carried by events is largely varied, because events are the principal method by which clients retrieve information<sup>[10]</sup>.
- **Errors** are similar to events, but are processed differently by clients. Errors are to be handed to error-handling routines by client-side programming libraries<sup>[10]</sup>. These routines may be defined by the programmer.

Requests that instigate a reply are called *round-trip requests*<sup>[10]</sup>. The use of such requests is ideally minimized in client programs as they lower performance in case of network delays<sup>[10]</sup>. A typical example of a round-trip request is the call for a window's attributes given in Listing 3.1.

### 3.8.1 Events

Events are the most important type of X message<sup>[10]</sup>; they are the principal means by which the window manager communicates with the environment. At the protocol level, events are packets sent from the server to a client<sup>[10]</sup>. They communicate that something the client may be interested in has happened<sup>[10]</sup>. For instance, an event is sent when the user presses a key on the keyboard, or when a mouse button has been clicked. Events not only notify of input, they may also be sent when a window requests to be mapped, or to indicate the creation of a new subwindow<sup>[10,11,12]</sup>.

Every event is connected to some window<sup>[10]</sup>. If the user clicks a mouse button when the mouse cursor is hovering over a window, the event will be relative to that window. The events a client receives need not be requests triggered by one of its own windows, however<sup>[10,11]</sup>. A client can request the server to redirect an event to another client; this is a means for clients to communicate within the X environment<sup>[11]</sup>. Requesting text that is currently selected in another client is an example of such communication. The resulting event is then sent to the client that holds the selection<sup>[11,12]</sup>.

Clients may only want to be notified of certain types of events. As such, events relative to a particular client are only reported when the client has previously *selected for* them<sup>[11,12]</sup>. For example, a client may want to be notified of keyboard input events, but not of mouse input events. Clients specify which types of future events they wish to receive

by setting a specific window attribute<sup>[10]</sup>. In order to be able to redraw a window when its contents have been affected by an external factor, a client must receive `Expose` events that inform it that the window needs to be redrawn<sup>[11]</sup>. A client will receive `Expose` events from the server, only if the client has previously instructed the server to notify it of such events. This is done by accordingly setting the event mask attribute of the window in question<sup>[10,11,12]</sup>. Consider the following example.

---

**Listing 3.3:** Setting a window's event mask

```
long mask = ExposureMask | KeyPressMask |
           KeyReleaseMask;
XSelectInput(dpy, win, mask);
// win will now receive exposure and keyboard
// input events
```

---

In this Xlib example, `dpy` again represents the required connection with the X server. `win` is the window for which we wish to receive events. The `XSelectInput` function requests that the X server report the events associated with the specified event mask, `mask`. Initially, X will not report any of these events<sup>[11]</sup>. Setting the event mask attribute of a window overrides any previous call for the same window, but not for other windows<sup>[11]</sup>.

If a window is not interested in a particular event, it usually propagates to the closest ancestor that is interested, unless the `do_not_propagate` mask prohibits it to<sup>[10,11]</sup>. This way, a client is able to receive and act on events for any of its subwindows<sup>[31]</sup>.

It is possible for different clients to request events on the same window. They may even set different event masks on the same window, because the server maintains a separate event mask for each client and window pair<sup>[10,31]</sup>. For example, a client may request only mouse input events on a window, while another client requests only keyboard input events on the same window. There are certain types of events that can only be selected for on a window by one client at once, such as event types that report about mouse button clicks and window management changes<sup>[10,11,31]</sup>.

## 3.9 Color

The programmer of a typical X application can specify color for the background and border of each of its windows, they can define colors for the cursor, and they can set foreground and background colors of so-called graphics contexts used for drawing graphics and text.<sup>[11]</sup> For a regular program, one that uses color merely to create a pleasing aesthetic, color correctness may not be of all too much importance. However, more complex applications—

such as Computer Aided Design (CAD) programs—might use color to distinguish physical or logical layers<sup>[11]</sup>. Other programs—those used in imaging, for instance—might require precise gradations of color to depict real-world data<sup>[11]</sup>. Therefore it is essential for an accurate color system to be in place; one that is to run successfully on the wide variety of screen hardware available in the X ecosystem<sup>[11]</sup>.

X's color system is implemented in such a way that programmers need not port their application multiple times to cover the spectrum of targeted display hardware<sup>[1]</sup>. X also provides for multiple applications to share a single colormap<sup>[1]</sup>. This is important, for applications should always show true color on the screen<sup>[1]</sup>. To achieve this, and to simultaneously have applications remain device independent, pixel values are not programmed explicitly into applications<sup>[1]</sup>. The X server instead manages the colormap, and the colormap is always expressed in hardware-independent terms<sup>[1]</sup>.

---

**Listing 3.4:** Changing colors

```
const unsigned long RED = 0xFF0000;
const unsigned long GREEN = 0x00FF00;
XSetWindowBackground(dpy, win, RED);
XSetWindowBorder(dpy, win, GREEN);
XClearWindow(dpy, win);
// win's background color is now red
// and its border color is now green
```

---

Without the need to tailor to hardware specifics, we are able to specify a hexadecimal (or *hex*) triplet to define a color<sup>[11]</sup>. A hex triplet is a six-digit, three-byte hexadecimal number. These bytes represent the red, green and blue components of the color. One byte represents a number in the range 00 to FF in hexadecimal notation, or 0 to 255 in decimal notation. This represents the least (0) to the most (255) intensity of each of the color components<sup>[11,12]</sup>. These values specify colors in the True Color (24-bit RGB) color scheme. The hex triplet is formed by concatenating three bytes in hexadecimal notation, as in our example in Listing 3.4, with `RED` and `GREEN`<sup>[11,31]</sup>.

In the most rudimentary of requests, the client simply supplies red, green, and blue color values. The server subsequently allocates an arbitrary pixel value and sets the colormap so that the pixel value represents the most accurate color that the display hardware can provide<sup>[1]</sup>. To avoid variations in color representation amongst displays, the server administers a color database that clients can use to translate names of colors into red, green, and blue values appropriate for the targeted display<sup>[1]</sup>.

## 3.10 Graphics operations

Generally, graphics operations are the most complex part of any windowing system, due to the fact that many different effects and variations are required to tailor to a wide range of applications<sup>[1]</sup>. The most important of graphics operations in X pertain to images, text, and exposures. We will go over each of them in turn.

### 3.10.1 Images

As alluded to in Section 3.6, X supports two types of off-screen images: pixmaps and bitmaps. A pixmap is an  $N$ -plane rectangle.  $N$  represents the number of bits per pixel used by the targeted display<sup>[1]</sup>. A bitmap is simply a pixmap with  $N = 1$ <sup>[1]</sup>. Pixmaps can be created by moving all bits to the server, or by copying a rectangular region of a window<sup>[1]</sup>.

Bitmaps are mainly used as masks, though several graphics requests use bitmaps as a clipping region<sup>[1]</sup>. A clipping region is merely a designated area used to selectively enable or disable the rendering of graphics<sup>[10]</sup>. Bitmaps may also be used to create cursors<sup>[1]</sup>. Pixmaps are used for storing frequently rendered images, and as a temporary backing store for pop-up menus<sup>[1]</sup>. Pixmaps are most often used as tiles<sup>[1]</sup>. Tiles are patterns that are replicated in two dimensions to cover a region<sup>[1]</sup>.

Graphics and text requests in X specify a logic function and a plane-select mask to alter the operation<sup>[1]</sup>. The function is applied bitwise on corresponding bits of source and destination pixels, but only on bits specified in the plane-select mask<sup>[1]</sup>. The most common logic function is the *copy* operation, replacing the destination pixels with source pixels in all planes<sup>[1]</sup>.

Another common graphics operation is the *bit block transfer*. With it, one region of a window is composited with another region of the same window<sup>[1]</sup>. The source and destination are given as equally sized rectangular regions of the window<sup>[1]</sup>. Overlap in the regions passed to this operation will not affect the result<sup>[1]</sup>.

X supports line drawing with a complex built-in primitive, which provides for arbitrary combinations of straight and curved line segments, and allows for the creation of both open and closed (filled) shapes<sup>[1]</sup>. Solid lines can be drawn by rendering with a single source pixel value, *dashed* lines are made by alternately drawing and not drawing, and *patterned* lines are created by alternately drawing with two source pixel values<sup>[1]</sup>. *Closed shapes* can be filled with either a pixel value or a tile<sup>[1]</sup>.

### 3.10.2 Text

For graphical text output, X incorporates bitmap font support<sup>[1]</sup>. Bitmap fonts consist of a matrix of pixels that represent the image of each glyph in each face and size. A font comprises up to 256 bitmaps<sup>[1]</sup>. The height of every bitmap in a font is the same, while their width may vary<sup>[1]</sup>. To facilitate the creation of server-bound font representations, clients can simply supply the name of a font, instead of having to copy bitmap images to the server<sup>[1]</sup>. A font may then be used—either as a mask or as a source—to draw strings of text<sup>[1]</sup>.

The server contains numerous *buffers* that clients may use to read and write arbitrary strings of bytes<sup>[1]</sup>. This functionality is in place to offer *cut-and-paste* operations between applications<sup>[1]</sup>. The server does not impose interpretation on the data, though they are mostly used for *clipboard* purposes to temporarily store text<sup>[1]</sup>. In general, cooperating applications can use these buffers to exchange resource identifiers and images<sup>[1]</sup>.

Numerous resources are used when performing graphics operations in X<sup>[11]</sup>. Most information about producing graphics—such as the foreground and background color, the font of text, and line style—is stored in a special type of X resource called a *graphics context*<sup>[11,24]</sup>. Most requests for graphics operations include a graphics context. In theory, the X protocol allows the sharing of graphics contexts between applications, though it is generally expected that applications use their own graphics contexts when carrying out operations<sup>[11,24]</sup>. Sharing of graphics contexts is not recommended, as client-side programming libraries may cache their states<sup>[11]</sup>. Graphics operations can be performed on all drawables<sup>[11,31]</sup>.

#### Listing 3.5: Displaying text

```
char *text = "TEXT";
XDrawString(dpy, win, gc, x, y, text, n);
// TEXT is displayed within win at x, y
// pixels from win's origin (top left)
```

In this Xlib example, we draw a string—fittingly containing the text “TEXT”—to the supplied window. In the call to `XDrawString`, we supply a graphics context, `gc`, that contains information about the font used, letter spacing, and much more. The designated position of the text is specified in terms relative to the origin of the window it will be drawn on, and thus not necessarily relative to the screen’s origin. As this is a C interface, Xlib requires the size of the string to be given, `n` here.

The most important aspect of any graphics system is performance.<sup>[24]</sup> To display an application’s graphical user interface, many different types of op-

erations are required, and, depending on rendering speed, techniques such as dragging images or the editing of complex graphics may or may not be possible<sup>[24]</sup>. X applies what are known as *poly routines*. Poly routines are hooks used for drawing<sup>[11]</sup>. They operate on an array of objects, which saves both network protocol overhead and set-up costs<sup>[24]</sup>. This technique surmises the observation that if an application draws a line or a rectangle, it tends to draw another with high probability. Besides routines for rendering text and clearing and copying regions, X has routines for rendering points, lines, rectangles, arcs, and polygons<sup>[24]</sup>.

### 3.10.3 Exposures

When a window is (partially) obscured, and then becomes visible again, the client is responsible for restoring the contents of the window<sup>[1]</sup>. The X server sends an event to a client when it has detected that one of its windows is in need of redrawing<sup>[1]</sup>. Naively, a client might respond to such an event by redrawing the entire window<sup>[1]</sup>. More sophisticatedly, one will redraw only the exposed region<sup>[1]</sup>. Of course, when a client has requested the server to administer a backing store for it, the server itself is capable of redrawing the window<sup>[1]</sup>. Even so, relying on client-controlled refresh derives from widely accepted window management philosophy<sup>[1,24]</sup>.

Applications cannot be written with predetermined top-level window sizes built in. They must generally work properly with virtually any size, and must continue to do so as the windows are resized<sup>[1]</sup>. This is required if applications are to run on a wide variety of displays under a wide variety of window management policies<sup>[1]</sup>. It is reasonable to expect an application to require a minimum size to function properly; X allows the client to attach resize hints to each of its windows, to inform window managers about size preferences<sup>[1]</sup>. More of such hints exist, and are further discussed in Chapter 4.

An Expose event is sent when an area of a window, of which the content is (partially) destroyed, is made visible<sup>[10,11]</sup>. There are numerous situations that lead to the destruction of window content<sup>[11]</sup>. One of them is the (partial) covering of a window by another window. The server generates an Expose event to notify the client that (part of) the window has to be redrawn<sup>[10,11,12]</sup>.

## 3.11 Fonts

The core protocol provides for the use of *server-side* fonts<sup>[10]</sup>. These fonts are accessed directly through use of the underlying filesystem, or otherwise over

the network from a so-called *font server*<sup>[10]</sup>. Fonts can be *loaded* and *unloaded*, meaning that they are respectively moved into and removed from the font server.<sup>[11]</sup> A font may only be unloaded if it is not in use by any other clients<sup>[11]</sup>. A client may send a request to inquire about font information, such as its inherent spacing or descent<sup>[11,12]</sup>.

Each font has a set of *metrics* that describe its inherent properties, such as spacing and which characters are defined<sup>[24]</sup>. Every glyph in the font has left and right bearing information, character width, and ascent and descent information<sup>[24]</sup>. Each font also has a set of properties that together convey other information, such as whether to superscript or subscript the font, its spacing values, underlining and strikeout information, weight, and more<sup>[24]</sup>. Font properties are extensible, and may additionally be defined by clients if they so please<sup>[24]</sup>.

The most basic form of graphical text output requests enables the client to change the font and to add spacing between characters<sup>[24]</sup>. A line of text can be printed with a single request<sup>[24]</sup>. Usually, glyphs of a font are interpreted as a mask. That is, only bits in the glyph that are set are rendered<sup>[24]</sup>. There are also text requests that additionally render the background of a character, as opposed to just the foreground<sup>[24]</sup>.

Font naming in X follows the conventions described in the X Logical Font Description Conventions (XLFD)<sup>[24]</sup>. The XLFD specifies how to name fonts; a font's name suggests point size, font face, italic, foundry, and the many other characteristics<sup>[24]</sup>.

Extensions to the X core protocol exist that move font access and glyph image generation from the X server to the X client. There are several advantages linked to client-side glyph management: access the details of the font file, application-specific fonts, rasterization, and the ability to share known fonts with the environment, like printers<sup>[32]</sup>. A popular such extension is the X Render Extension (or XRender). XRender allows for image compositing in the X server, which facilitates efficient display of transparent images and antialiasing<sup>[32]</sup>.

## 3.12 Input

In X, there are several translations that take place between the pressing of a key and its interpretation within a program<sup>[10,11]</sup>. Every physical key on the keyboard is associated with a *keycode*, which is nothing more than a number in the range 8–255<sup>[11]</sup>. The mapping between physical keys and keycodes is entirely server-dependent, and cannot be modified by a client. These keycodes only identify the



key itself, not what the key in turn maps to, like the character “k” or the term “Return”. Instead, these characters or terms are associated with what are called *keysyms*<sup>[11]</sup>. Keycodes represent physical keys that are pressed or released, whereas the value of a keysym depends on more than a single key press<sup>[11]</sup>. Lastly, there is a mapping from keysyms to strings. This mapping is local to the client. With it, a client can, for example, allow the user to map the function keys to strings, having what the user types appear as text inside the application<sup>[11]</sup>.

At the protocol level, whenever a physical key is pressed or released, the server respectively sends an X event of type `KeyPress` or `KeyRelease` to the client that has input focus<sup>[10,11]</sup>. Such events contain the following information<sup>[11]</sup>.

- The keycode of the key being pressed
- The state of the modifier keys (Shift, Control, Alt, etc.) and mouse buttons

It is up to the client to translate from keycode and modifier information to a keysym<sup>[11]</sup>. A client may, for instance, receive an event informing it that a specific key with keycode `0x61` has been pressed down, along with the Shift modifier. The client can convert the keycode into a character or term—in this case, keycode `0x61` stands for the lowercase character “a”—and hence associates this event with the uppercase character “A”<sup>[11]</sup>. Keysyms are often referenced in client-side programming libraries by named symbols, such as `XK_A` in `Xlib`<sup>[11]</sup>.

Even though the translation from keycodes and modifiers into keysyms is done client-side, the table that holds this association is maintained by the server<sup>[11]</sup>. As such, the table only needs to be stored in a single, centralized location, and all clients can access it. Usually, a client only requests this mapping to use it for decoding keycodes and modifiers into keysyms, though a client is also able to change it, if it wants to<sup>[11]</sup>. Since this mapping is server-wide, however, changes to the table will affect all clients<sup>[11]</sup>. Normal applications should therefore not want to make changes to this table.

A modifier is a key that, when pressed, changes the interpretation of other keys<sup>[11]</sup>. The Shift and Alt keys are such modifiers. Each modifier may be associated with more than one key on the keyboard. It is quite common for keyboards to have more than one key for a single modifier. Many keyboards have two Shift keys, for example. Although the two keys produce different keycodes when pressed, they both map to the same modifier within the X environment<sup>[11]</sup>.

The X server also reserves modifier mappings for the mouse buttons<sup>[11]</sup>. Mouse buttons cannot be

mapped or unmapped, only permuted<sup>[11]</sup>. This is mostly useful for interchanging left and right mouse buttons for left-handed users.

Within X, there are two modes of keyboard management: *real-estate* and *listener*<sup>[1]</sup>. In real-estate mode, the keyboard always follows the mouse cursor. That is, keyboard input is always directed to whatever window the mouse cursor is hovering over<sup>[1]</sup>. In listener mode, all keyboard input is directed to a specific window, regardless of the position of the mouse cursor<sup>[1]</sup>.

There are times when an application might wish to bypass normal keyboard or pointer event propagation, in order to receive input independent of the position of the pointer, and independent of keyboard management mode. This is the purpose of *grabbing* the keyboard and mouse<sup>[11]</sup>. A grab is a state within the X environment, in which all keyboard or mouse events are directed to a single client<sup>[11]</sup>. It is possible for a client to request all keyboard and mouse events to be grabbed, or for it to arrange that only certain keyboard and mouse events are directed to it<sup>[11]</sup>. Until the grab is subsequently released, all specified keyboard and mouse events are sent to the client that has initiated the grab<sup>[11]</sup>. Only the grabbing client will receive these events; other clients will not<sup>[11]</sup>. One reason for grabbing is to handle a series of events contiguously, without the intervention of unwanted events.

A client initiating a grab must specify which window is to be grabbed. The specified events are sent to the initiating client, as if they were relative to the window being grabbed<sup>[11]</sup>. Other clients do not receive these events, even if they have been set in the grabbed window's event mask<sup>[11]</sup>.

X implements *active* and *passive* grabs. An active grab takes effect immediately, whereas a passive grab becomes active only after a certain combination of keys or buttons is pressed<sup>[10,11]</sup>.

### Listing 3.6: Grabbing the mouse

```
int button = 1; // button 1 is the left mouse button
int modifiers = ShiftMask | ControlMask;
unsigned mask = ButtonPressMask | ButtonReleaseMask;
XGrabButton(dpy, button, modifiers, win, true, mask,
             GrabModeAsync, GrabModeAsync, None, None);
// win will now receive events informing it when
// a button has been pressed or released, but
// only if at the time of pressing or releasing,
// the Shift and Control keys are pressed down
```

The `XGrabButton` function establishes a passive grab. If in the future, the mouse is not already grabbed, the specified button (here, the left mouse button) is logically pressed down when the specified modifier keys are logically down, and no other buttons or modifier keys are logically down, then the

pointer becomes actively grabbed<sup>[11,31]</sup>. Once these conditions are met, a `ButtonPress` event is reported to the grabbing client<sup>[11]</sup>. Since, in our example, the grab's event mask consists of both `ButtonPress` and `ButtonRelease` events, the same will occur when the button is logically released, provided that analogous conditions are met<sup>[11,31]</sup>.

Grabbing the keyboard is similar to setting the keyboard focus window<sup>[11]</sup>. At the protocol level, giving a window keyboard focus generates a `FocusIn` event for the window gaining focus, and a `FocusOut` event for the one losing it<sup>[10,11]</sup>. Clients usually respond to such events by updating visual cues. A terminal emulator may do this by changing the command-line cursor, for example. A window manager will usually govern the transition of focus between clients, and will add its own visual cues, such as title bars that indicate which client is currently focused by, for instance, changing its background color. If mouse grabs and ungrabs cause the mouse to move in or out of a window, they generate `EnterNotify` and `LeaveNotify` events, respectively<sup>[10,11]</sup>.

A request for grabbing can include a request for *freezing* the keyboard or the mouse cursor<sup>[11]</sup>. The difference between the two, is that grabbing changes the which client receives events, whereas freezing ceases their delivery altogether<sup>[11,12]</sup>. The events a frozen window would normally instigate are instead queued up, such that they can be delivered once the freeze is released.<sup>[11,12]</sup>

It is also possible for a client to perform a grab on the server itself. Doing so will cause no requests to be processed by the server, except for those coming from the client that initiated the grab<sup>[11,12]</sup>.

### 3.13 Properties and atoms

In essence, a *property* in X is nothing more than a means by which clients communicate arbitrary data with each other<sup>[11]</sup>. At the protocol level, a property is a packet of named, typed data that is associated with a window<sup>[11]</sup>. A client's properties are always public, in the sense that they are available to all other clients running under the same server<sup>[10,31]</sup>. Properties are an essential part of the X environment, for without them, there would be no standardized way for clients to impart their preferences and requirements<sup>[11,31]</sup>. In fact, without them, X window managers would not nearly be as effective as they are with them (as discussed further in Chapter 4).

Properties have a string name and a unique numerical identifier, called an *atom*<sup>[11]</sup>. Atoms for properties are analogous to identifiers used to refer to server resources, except that both an atom and a window are needed to uniquely identify a prop-

erty<sup>[11]</sup>. The same atom may be used to identify a property on one window as on another—only the window is different in the calls to set or read this property on the two windows. In client code, properties are never explicitly referred to; they are underlying data managed by the server—just as with a window and its identifier<sup>[11]</sup>.

Property name strings are typically all upper case, with words separated with underscores, such as `WM_TRANSIENT_FOR`<sup>[11]</sup>. The protocol specification suggests that names of atoms used for private vendor-specific reasons should begin with an underscore<sup>[31]</sup>. To prevent conflicts among software libraries and organizations, additional prefixes should be chosen (e.g. `_KRAEWM_SIDEAR_WIDTH`)<sup>[2,31]</sup>.

A property also has a type, such as string, or integer—it can even be an atom<sup>[11,31]</sup>. Because these types are encoded using atoms, arbitrary new types can be defined by the client programmer. Data of only a single type may be associated with a property name<sup>[11,31]</sup>.

---

#### Listing 3.7: Retrieving a window property

```
Atom state = XInternAtom(dpy, "_NET_WM_STATE");
Atom _a, prop;
int _i;
unsigned long _ul;
unsigned char *prop_raw = NULL;

XGetWindowProperty(dpy, win, state, 0L, sizeof prop,
    false, XA_ATOM, &_a, &_i, &_ul, &_ul, &prop_raw);
prop = *(Atom *)prop_raw; // untyped data to Atom
XFree(prop_raw); // free X-allocated memory
// use prop, which in this case is the state of
// win, as defined in the EWMH (retrieval may
// fail; error handling redacted)
```

---

Because properties can hold data of any type, comprehensive client-side administration is required. In this example, we want to retrieve the state of the supplied window, as defined in the EWMH (Section 4.2). We pass the `_NET_WM_STATE` property name to the `XInternAtom` function, which queries the server for the atom associated with this property name<sup>[11]</sup>. Using the atom, we can ask the server for the data belonging to the supplied window for this property, by calling the `XGetWindowProperty` function<sup>[11]</sup>. In this case, the data returned is an atom itself, though it is initially untyped, so we must convert it to the appropriate type before using it<sup>[3,31]</sup>.

The core protocol has a set of *predefined properties* and associated atoms, which are always available for use by the client programmer<sup>[11]</sup>. It is essentially an implementation trick to avoid the cost of repeatedly *interning* (i.e. retrieving an atom from the server) atoms required upon startup of all graphical applications. Interning requires a handshake with the server, and can be a costly process if done (un-

necessarily) often<sup>[10,11]</sup>. An example of a predefined atom is `XA_WM_NAME`; it represents and is used to retrieve the title of a window<sup>[11]</sup>.

Client programmers are themselves free to define arbitrary information, store it using properties, and associate it with windows within the environment<sup>[31]</sup>. This may be useful for a group of related clients, amongst which several properties and atoms can be defined, that will have a meaning known to all clients in the group<sup>[31]</sup>.

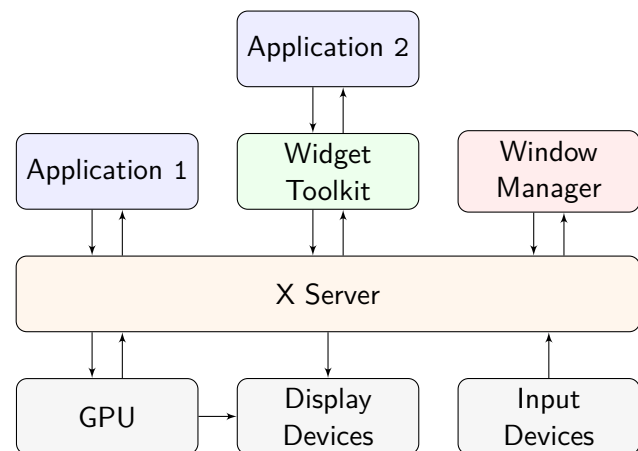
At server start-up, properties do not have any contents until a client or the window manager sets them<sup>[11]</sup>.

One of the most important uses of properties is to communicate information from applications to the window manager and vice versa<sup>[11]</sup>. A client may, for instance, set a property on its top-level window, to specify the range of sizes it prefers. A good window manager will take such client preferences, called *hints*, into account, and process them accordingly. Many such properties, collectively known as *standard properties*, are defined in standardized protocols, which we will discuss further in Chapter 4<sup>[11]</sup>.

### 3.14 The X window manager

X does not mandate any particular type of window manager; it does not even require a window manager to be running at all<sup>[10]</sup>. We saw that X operates in a client-server model, in which the server controls one or more physical display devices and input devices, and the clients are the applications that wish to interact with these devices and display their application interfaces to the user (Section 3.2). Window managers in X only concern themselves with the management interface, while leaving the management of each application interface to its respective client. In this schema, there is no separate role for the programs that form the desktop environment. This means that all programs that are not the X server are clients, no matter the responsibility of the program<sup>[10]</sup>. In particular, this means that the window manager is a client like any other application, even though it intuitively has more rights and responsibilities than those other applications. In fact, the window manager must be in control of the other applications, and may decide whether to accept or reject requests it receives from them. Despite this observation, the window manager operates entirely in user space; it does not require any superuser privileges to function<sup>[10,11]</sup>. To achieve this, it establishes a connection with the X server and makes use of a special set of APIs that only it has access to<sup>[11]</sup>. The X server makes sure that only a single window manager is running at a time, by only allowing one

client access to these APIs at a time<sup>[10]</sup>. As we shall discuss in Chapter 4, the window manager controls and communicates with other applications through the X server by using two mechanisms: properties and messages. The following diagram illustrates the interactions between entities in the X environment.



Interactions in the X environment

An important concept for X window managers is *substructure redirection*. When no window manager is running, an application that wants to do something with a window—like move it or resize it—sends a request that will be directly processed by the X server<sup>[10]</sup>. A window manager of course needs to intercept these requests, to decide whether or not to allow them. Reparenting window managers, for example, need to know when new windows enter the environment, such that they can decorate them by adding a frame. If that window is resized, then the frame around it also needs to be resized accordingly. Substructure redirection is the mechanism that allows window managers to not only be notified of these requests, but also to accept, deny or alter them<sup>[11]</sup>.

When a client registers for substructure redirection on a window, the registering client will receive all matching requests on any direct child window of the targeted window<sup>[11]</sup>. None of these requests will be executed by the X server unless they are accepted by the window manager. The X server therefore *redirects* any requests on the targeted window to the registering client<sup>[11]</sup>. The *structure* refers to the attributes of a window, that is, its position, size, border width, stacking order and mapping status. The *substructure* is any of these statistics about any child of a particular window<sup>[11]</sup>. Only direct children are targeted by substructure redirection; a request on a child of a child will not be received by the registering client<sup>[11]</sup>.

Registering for substructure redirection on the root window will allow the window manager to intercept any change to the configuration of any child

of the root window. But as we have seen, all top-level windows in the X environment are children of the root window, and so rerouting client window requests to non-reparenting window managers is quite straightforward. If a window manager reparents its windows, however, it is harder to intercept client window requests, because the frames will be the direct children of the root window. A reparenting window manager will therefore need to do some administration to make sure it stays in the know of every client window's behavior.

The X server allows only one client to register for substructure redirection on any given window at a time. The request for registering substructure redirection will fail if another client has already successfully done so on the same window, and has not since unregistered, disconnected from the X server, or crashed<sup>[10,11]</sup>. Since every window manager must register for substructure redirection on the root, in X, it is not possible for two or more window managers to run on the same display simultaneously<sup>[11]</sup>.

### 3.15 Wayland

*Wayland* is developed by a group of volunteers as a free and open source community-driven project, with the intention of replacing the X Window System with a modern, simpler windowing system in Linux and other Unix-like operating systems<sup>[33]</sup>.

Like the X Protocol, Wayland is but a protocol that specifies the communication between a windowing system and its clients<sup>[33]</sup>. A windowing system implementing the Wayland protocol is called a Wayland *compositor*<sup>[33]</sup>.

The developers of the protocol also provide a reference implementation of the protocol, written in C, named *Weston*<sup>[33]</sup>. Weston can run as an X client or under Linux kernel mode setting (KMS), and ships with a few demo clients<sup>[33]</sup>. The Weston compositor is a minimal and fast compositor that is suitable for many embedded and mobile use cases<sup>[33]</sup>.

The compositor can be implemented in several ways. It may be a standalone windowing system, running on Linux KMS and evdev input devices, an X application, or a Wayland client itself<sup>[33]</sup>. The clients can be traditional applications, X servers, or other windowing systems<sup>[33]</sup>.

#### 3.15.1 Architecture

To discuss the Wayland architecture, and to show how it differs from that of X, we will follow an event from an input device to the point where the change it propagates appears on screen. We assume a compositing window manager is running on top of X,

as this best demonstrates the problem with X, and Wayland's approach to alleviating it.

In X, the following sequence of events takes place<sup>[33]</sup>.

1. The kernel receives an event from an input device, and sends it to the X server through the input driver.
2. The X server determines for which window the event is meant, and sends it to the clients that have stated interest in it (Section 3.8.1).
3. The interested clients decide what to do with the event. Most of the time, the application interface of the window instigating the event will need to change, if, for example, the mouse is hovering over a link that must change color, or if a checkbox was clicked. After processing the event, the client sends a rendering request to the server.
4. When the X server receives the rendering request, it sends it to a driver to have the hardware perform the rendering. Because the compositor may need to apply effects in response to the event, it requests the X server to calculate the clipping region of the rendering, and to send it a damage event for that region.
5. The damage event tells the compositor that something has changed in the window, and that it has to recomposite the part of the screen defined by the supplied clipping mask. Usually, the compositor now sends the composited image back to the X server for rendering.

In step 2 of the above sequence, because a compositor (Section 2.2.4) may perform transformations on a window (such as scaling and animations), and thereby controls the window's final location and shape on screen, the X server cannot with certainty say that it knows exactly which part of the window is affected by the event. This is a direct consequence of the fact that compositing features are not built into the server. A compositor in X is—just as any window manager in X—a client of the server, and therefore interacts with the server in much the same way as any client does. This means that only *after* the server has rendered a window and its content, the compositor applies its effects.

The flaw here is obvious. The compositor is responsible for rendering everything on the screen, yet it first has to go through the X server to achieve this. Essentially, the X server has now become an intermediary between applications and the compositor, and between the compositor and the hardware.

In Wayland, the compositor sends input events directly to the clients, and lets clients send damage events directly to the compositor. A change in the environment follows the following sequence of events in Wayland<sup>[33]</sup>.

1. The kernel receives an event from an input device, and sends it to the Wayland compositor through the input driver.
2. The compositor determines for which window the event is meant, and knows the exact region affected, because it is aware of what is on screen, and it understands which effects and transformations are applied to which elements.
3. The client is responsible for rendering its application interface in response to the event. It subsequently sends a damage request to the compositor to indicate which region has changed.
4. The compositor collects damage requests from all its clients, and then recomposites the screen.

As the windowing system incorporates compositing abilities, the aforementioned problems with X are circumvented<sup>[33]</sup>.



# Chapter 4

## Policy in X

X was built with the intention to specify only mechanism, and not policy<sup>[10,13]</sup>. A programmer looking to write an application for X is handed a programming interface and a lot of freedom in deciding how the application is to behave. Thanks to this, an application will almost always operate correctly in isolation, given there are no flaws in its basic functioning. However, a user seldomly uses a graphical application without also concurrently running other programs (like a window manager). What should happen when two programs wish to share information about the graphical environment? Is there a shared clipboard that allows the user to copy information from one program, and paste it in another? How does a tiling window manager know whether a program's window is a dialog box or not, such that it can decide whether or not it should tile it along with the other windows, or have it operate in the floating state? Different programmers may choose to define policy in vastly different ways. Not only the communication medium may be different, but also data representation and semantics. Immediately it becomes clear that even though it is beneficial for a windowing system to be as simple as possible—by only offering the most rudimentary of operations for constructing applications—a graphical environment without policy only leads to unexpected (faulty) program behavior and ultimately to user exasperation.

To specify policy within the X environment, an inter-client communication protocol is defined *on top of* the X protocol<sup>[2,13]</sup>. There are several standard protocols widely established, of which we will discuss the most important two here.

### 4.1 ICCCM

The *Inter-Client Communication Conventions Manual* (most often referred to as the *ICCCM*) is a standard communication protocol for the X Window System. It defines policy for communication between clients within the X environment through use of the X server. Most of the standards defined in it pertain to communication between regular X clients and the window manager, though it also proposes protocols on selection management, cut buffers, session management, manipulation of shared resources, and device color characterization. In short, when it comes to creating a cohesive environment in which graphical applications can communicate with each other seamlessly, it tries to cover everything that the X

Window System lacks. The ICCCM was designed at the X Consortium (which has since been absorbed by The Open Group) in 1988, just a few years after the initial release of the X Window System.

Throughout the rest of this section, the portions of the ICCCM relevant to our case study (the development of an ICCCM and EWMH compliant top-level tiling window manager, detailed in Chapter 7) will be discussed. In doing so, we will be continually referencing the manual<sup>[2]</sup>, only explicitly referring to other sources whenever they are used.

The ICCCM proposes conventions its authors deemed suitable, without attempting to enforce any particular user interface. To also not impose any particular programming language or system architecture, the conventions in it are expressed only in terms of protocol operations, not in any specific programming interface related denotation.

To impose semantics on property names, the protocol defines six name spaces amongst which a particular atom can have some client interpretation. An atom can belong to multiple name spaces, and the property associated with it depends on the name space (context) from which it is called. The name spaces are as follows<sup>[11]</sup>.

- Property name
- Property type
- Selection name
- Selection target
- Font property
- ClientMessage type

The first two deal with generic properties (as described in Section 3.13) that do not belong to any of the other name spaces. The third and fourth govern properties related to *selections*. Selections in X are the most powerful method of general communication between clients. They establish a dialogue between two applications, juxtaposed with *cut buffers* which only set up a one-way communication channel<sup>[11]</sup>. Selections are also used to realize a shared clipboard in X<sup>[11]</sup>. Font properties evidently specify anything to do with fonts, and ClientMessage type properties pertain to X events manually generated by a client (as opposed to automatically by the server) using the `XSendEvent` function<sup>[11]</sup>.

As we are primarily concerned with window management in this paper, we will only cover that part of the ICCCM that deals with communication between clients and the window manager. An interested reader is referred to the manual itself for a

rigorous description of and discussion on peer-to-peer selection and cut-buffer communication, session management, manipulation of shared resources, and device color characterization.

To enable window managers to assume their expected role of mediating clients' competing demands for screen space and other resources, several conventions must be in place that dictate on one hand how clients should communicate their desires, and on the other how the window manager is to retrieve from clients the information it requires to do its job well. These conventions should be defined in such a way that a normal client should neither know nor care which or whether a window manager is running. Of course, when no window manager is running, some client functions will not be supported. For example, without a window manager active, the concept of being iconified does not apply to a client, as iconification is a feature implemented by the window manager, and not by the client.

#### 4.1.1 Client properties

Clients should always act as they would in the absence of a window manager, with the following three exceptions.

- The client should expose *hints* to the window manager about resources it would like to obtain;
- the client should cooperate with the window manager by accepting resources allocated to it, even if they are not explicitly requested; and
- the client should be prepared for resource allocations to change at any time.

A window manager should always only be concerned with top-level windows—that is, direct children of the root. If the window manager reparents the clients it manages, it will control the placement and size of the frame window, but read properties from the original window (the child of the frame). A client should place certain properties on its top-level windows to inform the window manager of behavior and resources it desires. For any properties not defined by a client, the window manager will have defaults it assumes apply. Clients that have specific requirements must therefore always supply associated properties, and should not rely on the window manager's assumptions. The window manager will examine properties set by the client and monitor them for changes. The properties a client should always want to set are the following.

- WM\_NAME

- WM\_ICON\_NAME
- WM\_CLASS
- WM\_TRANSIENT\_FOR
- WM\_NORMAL\_HINTS
- WM\_HINTS
- WM\_PROTOCOLS

We will briefly cover each of them in turn.

##### WM\_NAME

The WM\_NAME property has type TEXT, and is an uninterpreted string that the client wants associated with the window. The value set in this property will usually be the title or name of a window. A window manager may, for example, use this string in the title bar of the frame around the window, or in the status bar to indicate which window currently has input focus.

Window managers are expected to display this information; ignoring WM\_NAME is behavior not in accordance with the ICCCM. Clients may assume that at least part of this string will be visible to the user. Of course, even if a window manager displays the string in question, it is not guaranteed it will be visible to the user at all times. The window may be partially off-screen, or otherwise covered. The client programmer must therefore never encode WM\_NAME with application-critical information, or use it to announce changes of the application's state such as to elicit a timely user response.

##### WM\_ICON\_NAME

The WM\_ICON\_NAME property (of type TEXT) is similar to WM\_NAME, in that it is an uninterpreted string that the client wants to be interpreted by the window manager as being the title or name of the window. As its name suggests, this string will be displayed when the window is iconified. Clients should not themselves attempt to display this information in their icon pixmap or window, and should instead rely on the window manager to do so.

##### WM\_CLASS

The WM\_CLASS property is of type TEXT and comprises a string without control characters that contains two consecutive null-terminating strings. It specifies the instance and class names to be used by the client and the window manager to retrieve resources for the application and identify the client. This property must be set when a window leaves the `WithdrawnState` state, and can only be changed when in the `WithdrawnState` state.

**WM\_TRANSIENT\_FOR**

The WM\_TRANSIENT\_FOR property is of type WINDOW, and contains the ID of a different top-level window. A window that sets this property should be a pop-up or dialog window of the window represented by the ID set in the property. Window managers will often decide not to decorate (reparent) such windows, or otherwise treat them differently.

**WM\_NORMAL\_HINTS**

The WM\_NORMAL\_HINTS property is an important property used by a client to indicate size and position preferences to the window manager. A window manager that complies with the ICCCM will take these values in account, and respect them when possible. The type of this property is WM\_SIZE\_HINTS, and its contents are the following.

field	type	default
flags	CARD32	
pad	4 × CARD32	
min_width	INT32	base_width
min_height	INT32	base_height
max_width	INT32	
max_height	INT32	
width_inc	INT32	
height_inc	INT32	
min_aspect	(INT32, INT32)	
max_aspect	(INT32, INT32)	
base_width	INT32	min_width
base_height	INT32	min_height
win_gravity	INT32	NorthWest

The min\_width, min\_height, max\_width and max\_height members respectively specify the minimum and maximum window sizes that still allow the application to be useful. The width\_inc and height\_inc members define an arithmetic progression of sizes (minimum through maximum) into which the window prefers to be resized. The min\_aspect and max\_aspect fields represent ratios of x and y; they allow an application to specify the range of aspect ratios it prefers. The base\_width and base\_height members are set by an application to specify the exact size it wishes to be given upon startup, and it is the base size to which size increments are added.

The win\_gravity member is a point within the body of the window (for example, East or Center), including any frame and border, general to which the window manager should interpret the rest of the data.

The pad field is added padding for backwards compatibility, and the flags bit definitions are as follows.

name	value	field
USPosition	1	User x, y
USize	2	User width, height
PPosition	4	Program x, y
PSize	8	Program width, height
PMinSize	16	min_width, min_height
PMaxSize	32	max_width, max_height
PResizeInc	64	width_inc, height_inc
PAspect	128	min_aspect, max_aspect
PBaseSize	256	base_height, base_width
PWinGravity	512	win_gravity

The first two bits indicate to the window manager that the client's request for changing the position or size of the window was generated by the user. The rest of the bits represent requests generated by the program without direct user involvement. A distinction is made because a user's wishes should generally have precedence over a program's suggested position and size hints. User-specified hints allow a window manager to know that the user specifically instructed a window to be moved or resized, and that further interaction is not required. An example of user-specified position and size constraints would be xterm, when ran with the -geometry flag, allowing the user to determine the size and position of the window on startup<sup>[34]</sup>.

**WM\_HINTS**

The WM\_HINTS property is used to communicate to the window manager anything not covered by the WM\_NORMAL\_HINTS property and several strings that require separate properties, such as WM\_ICON\_NAME. Its type is WM\_HINTS, and the contents of the property are as follows.

field	type
flags	CARD32
input	CARD32
initial_state	CARD32
icon_pixmap	PIXMAP
icon_window	WINDOW
icon_x	INT32
icon_y	INT32
icon_mask	PIXMAP
window_group	WINDOW



The `input` member communicates to the window manager the input focus model used by the application. Applications that expect input but never explicitly set focus to any of their subwindows (push model focus management) should set this member<sup>[11]</sup>. Applications that set input focus to their subwindows only when the window manager assigns focus to their top-level window should also set this member<sup>[11]</sup>. Applications that independently manage their own input focus by explicitly setting focus to one of their subwindows whenever they want keyboard input (pull model focus management) should not set this member, and neither should windows that never expect keyboard input focus<sup>[11]</sup>.

Pull model window managers should enable push model applications to get input by setting input focus to the top-level windows of applications that have its input member set<sup>[11]</sup>. Push model window managers should assert that pull model applications do not break them by resetting input focus to `PointerRoot` when appropriate (for instance, when an application with its input member set sets input focus to one of its subwindows)<sup>[11]</sup>.

The `icon_pixmap` member represents a pixmap that may be used as an icon, and the `icon_window` member, if set, should contain the ID of the window that the client wants to use as its icon. The `icon_x` and `icon_y` fields indicate to the window manager the preferred position of the icon. Of course, the window manager ultimately chooses the position of the icon, and the client should therefore not assume the acknowledgement of this hint. The `icon_mask` defines which pixels of the icon pixmap should be used in the icon, allowing for nonrectangular icons.

At any time, a client can be in one of three states: `NormalState`, `IconicState` or `WithdrawnState`. The `NormalState` state suggests that a client's top-level window is viewable as normal. When a client is in the `IconicState` state, its top-level window is not viewable, but iconified, whatever that means for the window manager, because, as mentioned earlier, iconification is implemented by the window manager and not by the client. If it is set in `icon_window`, the icon window will be viewable; otherwise the icon pixmap (`icon_pixmap`) or the string in `WM_ICON_NAME` will be displayed. A client in the `WithdrawnState` state neither has its top-level window nor icon visible. A window manager is allowed to implement states with different (self-defined) semantics if it so desires. A client's initial state is captured in the `initial_state` field. Newly created top-level windows always start in the `WithdrawnState` state, and only the client can transition itself out of the `WithdrawnState` state, before the window manager takes control and the initial

state is effectuated.

The `window_group` member allows a client to specify that the window belongs to a group of windows; window managers may provide facilities for manipulating a group as a whole.

The flags bit definitions are the following.

name	value	field
<code>InputHint</code>	1	<code>input</code>
<code>StateHint</code>	2	<code>initial_state</code>
<code>IconPixmapHint</code>	4	<code>icon_pixmap</code>
<code>IconWindowHint</code>	8	<code>icon_window</code>
<code>IconPositionHint</code>	16	<code>icon_x</code> , <code>icon_y</code>
<code>IconMaskHint</code>	32	<code>icon_mask</code>
<code>WindowGroupHint</code>	64	<code>window_group</code>
<code>MessageHint</code>	128	<i>obsolete</i>
<code>UrgencyHint</code>	256	<code>urgency</code>

The `MessageHint` bit, if set in the flags field, indicates that the client is using an obsolete window manager communication protocol.

The `UrgencyHint` flag, if set in the flags field, is used by a client to communicate that the contents of a window are urgent, in the sense that a prompt user response is required. Window managers must do their best to induce the user to focus on this window as long as the flag in question is set.

A window manager is free to assume values it deems convenient for all fields of the `WM_HINTS` property that have not been defined by a client.

## WM\_PROTOCOLS

The `WM_PROTOCOLS` protocol has type `ATOM[]`, and defines a list of atoms. Each atom in the list specifies a communication protocol the client is willing to use to talk with the window manager. A client is not obligated to set this property; if no value is set, then the client does not wish to participate in any window manager protocol.

### 4.1.2 Window manager properties

The properties discussed in Section 4.1.1 are those defined by the client to communicate information to the window manager. There are also properties that the window manager sets on clients' top-level windows and on the root window, which we will discuss next. The most important properties that the window manager is concerned with are the following two.

- `WM_STATE`
- `WM_ICON_SIZE`

Furthermore, the ICCCM stipulates that a window manager should interfere in the following situations.

- A window requires a change in state;
- a window wishes to be *configured*, that is, resized or repositioned;
- a change has occurred or is to occur in a window's attributes;
- a window wishes to acquire input focus;
- a window is to be iconified;
- a client wants to *pop up* a window; or
- a window has been added to or removed from a window group.

We shall now briefly elaborate on each of these points.

### WM\_STATE

The WM\_STATE property has type WM\_STATE, and should be set by the window manager on all top-level client windows that are not in the WithdrawnState state.

There exist programs that require information on top-level client windows in the X environment. When a reparenting window manager is running, such a program does not know whether the top-level window it targets is a frame or a client window. An example of such a program is xprop, which is part of the X.Org Foundation Distribution (Appendix C). xprop will ask the user to click a top-level window on which it is to operate. To convey to the environment which window is the top-level window of a client, the window manager must set the WM\_STATE property on this window. A client should recursively search the window hierarchy until it finds a window with this property set.

### WM\_ICON\_SIZE

The WM\_ICON\_SIZE property is set on the root window by the window manager, and is used to place constraints on sizes of icon windows and icon pixmaps. The contents of this property are the following.

field	type
min_width	CARD32
min_height	CARD32
max_width	CARD32
max_height	CARD32
width_inc	CARD32
height_inc	CARD32

The fields in this property are very similar to those in WM\_NORMAL\_HINTS, which a program uses to encode window size hints that are to be taken into account by the window manager.

Clients that wish to customize the appearance of their icon must take this property into account.

### Changing the state of a window

As we have seen, a client window can be in one of three states (WithdrawnState, NormalState or IconicState) or in a state with semantics the window manager implements itself. State changes can be imposed by the window manager or precipitated by the client to which the window belongs. New windows start out in the WithdrawnState state, and possible subsequent transitions are the following.

from		to
WithdrawnState	→	NormalState
WithdrawnState	→	IconicState
NormalState	→	IconicState
NormalState	→	WithdrawnState
IconicState	→	NormalState
IconicState	→	WithdrawnState

The first two specify transitions out of the WithdrawnState state; the *initial\_state* field of the WM\_HINTS property must be set to NormalState and IconicState respectively at the time the client maps the window. The rest of the transitions take place when the window is actively in use, and occur when the window is iconified, deiconified and unmapped.

Reparenting window managers must make sure to unmap windows that transition into the IconicState state; merely unmapping the frame window is not sufficient.

A window manager must update or remove the WM\_STATE property when a client withdraws a window. Clients that wish to re-use a window after issuing a withdrawal must wait for the withdrawal to complete, by waiting until the window manager has removed or updated the aforementioned property.

On transitions from the NormalState state to IconicState, a client must send a ClientMessage event to the root window. This event must contain the window to be iconified, the WM\_CHANGE\_STATE atom, and the IconicState value.

### Configuring a window

To configure a window, clients can send a ConfigureWindow request to the server. Using this

request, several attributes can be changed: the window's position on screen, its size, its border width, and its place in the stack (which is used by the windowing system to determine which windows are to be drawn in front of or behind which others on screen). The position is determined relative to the root window's origin, irrespective of any frames that may be around the window if reparenting has occurred. In determining the position, the most recently set value for `win_gravity` is taken into account. For example, if the last known value for `win_gravity` for a window was set to `East`, and a request to reposition that window has been received, the center of the right edge of the window will be exactly positioned on the supplied coordinates. `Configure` requests are similar to initial geometry preferences that are set in the `WM_NORMAL_HINTS` property (read when a window transitions from the `WithdrawnState` state) and are interpreted in the same way by the window manager. Clients are not guaranteed that the window manager will adhere to these requests, and must therefore be prepared to take on any size and position; a tiling window manager, for example, has its own arrangement policy, and will ignore `ConfigureWindow` requests that state the wish to alter the window's dimensions.

A window manager generally has three ways to respond to a `ConfigureWindow` request:

- The window manager does not allow anything the client requests, and the window the request was issued for remains unchanged;
- the window manager moves or restacks the window, without resizing it or changing its border width; or
- the window manager resizes the window or changes its border width (regardless of whether it was also moved or restacked).

In the first situation, the client will receive a synthetic `ConfigureNotify` event informing it of the unchanged dimensions of the window. The position supplied is in the root coordinate system, that is, relative to the root window's origin. The client will not receive a real `ConfigureNotify` event, as no change has really taken place. In the second situation, the client will also receive a synthetic event of the same type, this time containing the new dimensions of the window. The client may also receive a real `ConfigureNotify` event, only if that client's top-level window has not been reparented by the window manager. In that case, the synthetic event will follow the real one. In the third situation, if the client has selected for `StructureNotify` events to be received, a real `ConfigureNotify` event will be sent containing the new dimensions of the window.

All of these synthetic events contain coordinates that are relative to the root window's origin, while the coordinates in the real `ConfigureNotify` events are in the window's parent's space (i.e., relative to the parent's origin). Both real and synthetic events are sent because the client may want to use them for different purposes.

A client must note that its windows' borders may not be visible, as window managers are free to remove them, and often do when applying reparenting techniques. A client's attempts to alter the border width will then likely be overridden, and set to zero. Clients must therefore never depend on changes to a window's border width, and must never assume that the border will be visible.

Clients that issue requests to change a window's place in the stack may fail when that window has been reparented by the window manager. Window managers are always free to choose a window's place in the stack as they see fit. As such, clients must never rely on the stacking order it suggests to a window manager.

### Altering a window's attributes

A window in the X environment has several attributes that are supplied upon window creation. These attributes can be changed using the `ChangeWindowAttributes` request, although most of them are private to the client:

attribute	accessibility
background pixmap	private
background pixel	private
border pixmap	private
border pixel	private
bit gravity	private
<b>window gravity</b>	<b>public</b>
backing-store hint	private
<b>save-under hint</b>	<b>public</b>
<b>event mask</b>	<b>public</b>
do-not-propagate mask	private
<b>override-redirect flag</b>	<b>public</b>
colormap	private
cursor	private

The attributes that are private to the client will never be altered by a window manager. The four public ones may, and often will change to fit the window manager's needs. The window gravity of a client's top-level window will sometimes be changed by a reparenting window manager, depending on its decoration policy. A client can set the `save-under`

hint on its top-level window (which asks the X server to save the contents of any windows that are rendered below it and have been obscured), but must take into account that it can be overridden by the window manager. Every window has per-client event masks, as we have seen in Section 3.8.1. As a result, clients are allowed to select for any events deemed convenient, irrespective of any events the window manager may be selecting for. Some events can only be selected for by a single client; the ICCCM states that a window manager must never select for any of these events. Clients are discouraged from setting the `override-redirect` attribute on top-level windows, except when popping up windows or redirecting requests. The `override-redirect` flag is used to specify whether `map` and `configure` requests on a window should override a `SubstructureRedirectMask` mask on the parent. Window managers use the `SubstructureRedirectMask` mask to intercept (redirect) `map` and `configure` requests such as to control window placement or add decoration. If a client sets the `override-redirect` flag on its top-level window, the window manager will not be notified of the necessary events, and will consequently not be able to manage it in a way the user expects. Pop-up windows, on the other hand, do need to be mapped without window manager interference.

### Acquiring input focus

The ICCCM identifies four models of input handling:

- **No input.** Keyboard input is never handed to the client, and the client never expects as much. Such clients are essentially output-only. An example is `xload`, a program that displays system load averages.
- **Passive input.** The client expects to be handed keyboard input, but it never explicitly sets input focus. An example of such a client is one that has no subwindows and only accepts input in `PointerRoot` mode (in which the focus window is dynamically taken to be the root window of whatever screen the pointer is on at each keyboard event) or when the window manager sets input focus to its top-level window.
- **Locally active input.** The client expects to be handed keyboard input, and it explicitly sets input focus when one of its windows already has focus. An example of such a client is one with subwindows that can be cycled through using the `Next` and `Prev` keys, moving input focus between the subwindows. This only works if the client's top-level window has acquired focus in `PointerRoot`

mode or when the window manager sets input focus to its top-level window.

- **Globally active input.** The client expects to be handed keyboard input, and it explicitly sets input focus, even if focus is currently in a window that it does not own. An example of such a client is one that allows the user to scroll the window without disturbing input focus, even if it is in another client's window. It wants to acquire input focus only when the user clicks in the scrolled region, but not when they click the scroll bar itself.

A client can realize any of these four models by appropriately setting and unsetting the `input` field in `WM_HINTS` and the `WM_TAKE_FOCUS` atom in the `WM_PROTOCOLS` property.

input model	input	WM_TAKE_FOCUS
no input	unset	unset
passive input	set	unset
locally active input	set	set
globally active input	unset	set

Clients that implement the passive or locally active input models both must set the `input` field, indicating to the window manager that they require assistance in acquiring input focus. Clients with an active input model (be it local or global) and other windows that set the `WM_TAKE_FOCUS` atom in their `WM_PROTOCOLS` property may receive a `ClientMessage` event from the window manager; if the client receiving such an event wants focus at that time, it should respond with a `SetInputFocus` request with the window field set to that client's window that last had input focus, or to their default input window. A client may receive `WM_TAKE_FOCUS` when it transitions from the `IconicState` state, or when an area outside of its top-level window (such as its frame) is clicked, given the window manager uses this to assign focus. Ultimately, the goal is to support window managers that want to assign input focus to a client's top-level window in such a way that the client can decide to pass it on to one of its subwindows, or decline it altogether. The ICCCM suggests that a client should never give up input focus on its own volition; it should ignore input it receives instead.

The method by which the user commands the window manager to set the focus to a window is up to the window manager. For instance, clients are not guaranteed to detect a click that transfers focus to it.

When the window that currently has input focus becomes not viewable, the focus needs to revert to

another window. To indicate to the window manager which window to fall back on in such an event, the `revert-to` field of the `SetInputFocus` request is set to one of three values: `Parent`, `PointerRoot` or `None`. The first is generally employed by clients that assign focus to subwindows. If the subwindow is unmapped, the focus will remain in the client's possession. The second transfers input focus to the root window of the screen currently containing the mouse pointer. The third completely releases input focus. Neither `PointerRoot` nor `None` are entirely safe to use, because if the window manager crashes while input is lost, there is no way to reobtain it. As such, clients that invoke a `SetInputFocus` request should set its `revert-to` field to `Parent`.

### Iconification

To know what to display in a window's icon, the window manager reads the `WM_ICON_NAME` property associated with that window. The pixmap a client specifies to be used in the icon (for example, a logo) should be of one of the sizes found in the `WM_ICON_SIZE` property. If the property is not found, the window manager will likely not display the pixmap, show only part of it if it is too large, or tile it if it is too small. A client may set the `icon_window` field of the `WM_HINTS` property for a window; when that window is iconified, the client expects the window manager to map that window. The icon window should ideally be of one of the sizes set in the `WM_ICON_SIZE` property on the root window, but window managers are reserved the right to resize these windows as they see fit.

When a window is iconified, a window manager should generally make sure that the following conditions are met.

- If the `icon_window` field is set in the `WM_HINTS` property of the iconified window, the window it refers to should be visible;
- if the `icon_window` field is not set in the `WM_HINTS` property of the iconified window, but the `icon_pixmap` field is set in that property, the pixmap it refers to should be visible; and
- otherwise, the string in the `WM_ICON_NAME` property for that window should be visible.

A window manager may choose to implement more than just one of the above. For example, some window managers display both the string in `WM_ICON_NAME` and the icon pixmap in a client's icon.

The ICCCM suggests clients to stick to the following conventions pertaining to iconification.

- The icon window should be an `InputOutput` window;
- the icon window should be a child of the root window;
- the icon window should be one of the sizes in the `WM_ICON_SIZE` property of the root window;
- a client should leave mapping of the icon window to the window manager;
- a client should leave unmapping of the icon window to the window manager;
- a client should leave configuration of the icon window to the window manager;
- a client should not set the `override-redirect` flag on the icon window;
- a client should not select for `ResizeRedirect` events on the icon window;
- a client should not assume that it will receive input events through its icon window; and
- a client should select for `Exposure` events on its icon window, and repaint its contents when the X server requests it.

Most window managers will support input events on a client's icon window, such as mouse or keyboard events. For example, a window manager may deiconify a window if its icon has been clicked.

### Pop-up windows

To create a pop-up window, there are three things a client can do:

- Create a new top-level window as normal; it will get managed and possibly reparented by the window manager.
- Create a new top-level window with the `WM_TRANSIENT_FOR` property set to the ID of the window creating the pop-up window; the window manager knows it belongs to another window, and will often add less or no decoration. Clients should create a window of this type if it expects the window to be visible for a relatively short period of time. An example of such a window would be a dialog box.
- Create a window with the `override-redirect` flag set. Because the window manager will never manage these windows, this should only be done if the pointer is grabbed as long as the window is mapped. Clients should create a window of this type if it expects the window to be visible for a very short period of time. A pop-up menu is an example of such a window.

Because a user is not able to configure or transfer input focus to a window that has the override-redirect flag set (due to the fact the window manager will not be managing that window), a client will have to grab the keyboard or have another top-level window with an active focus model to receive keystrokes on that window.

Window managers are free in deciding whether or not a window that has the `WM_TRANSIENT_FOR` flag set should be iconified when the window it is transient for is.

### Window groups

As we have seen when we discussed the properties a client should set to communicate preferences to the window manager, the `window_group` field of the `WM_HINTS` property is used by a client to state a relation between that window and other top-level windows. Each window in the group should set the `window_group` to the same window ID, representing the group leader. Window managers may treat the group leader differently from the other windows in the group. An example of this would be that the group leader gets a decorated frame around it, while the others do not. The group leader need not necessarily be a window that is in use by the user; it may just as well be an invisible window that exists merely as a placeholder. The window manager is entirely free in deciding how it manages a group.

The ICCCM additionally enumerates conventions that stipulate how a client is to respond to window manager actions. We will not further discuss them here, as they do not directly affect or concern the window manager or its implementation. An interested reader is again referred to the manual itself for information on client compatibility.

## 4.2 EWMH

The *Extended Window Manager Hints* specification (referred to as the *EWMH*, also known as *NetWM*) is a standard communication protocol for the X Window System that was developed by the X Desktop Group (XDG). It is an extension to the ICCCM, defining conventions for interaction between window managers, regular clients, and utilities that define the desktop environment such as docks and status bars. The ICCCM specifies window manager interactions at a lower level, and does not provide much ways to implement features expected in modern desktop environments. The Gnome and KDE desktop projects had originally developed their own extensions to the ICCCM to support such features; the EWMH re-

places those custom extensions, and offers a set of standardized ICCCM additions that any desktop environment can adopt.

Again, only the parts of the specification<sup>[3]</sup> relevant to our case study (Chapter 7) will be discussed and continually referenced. Other sources will be explicitly referred to.

Like the ICCCM, the EWMH specification defines only protocol operations. It specifies properties to be used by window managers and clients alike to communicate their needs and preferences.

### 4.2.1 Non-ICCCM features

The EWMH recognizes window management features and behaviors that are not specified in the ICCCM, but are common in modern desktop environments, most important of which are the following.

- additional states
- modality
- large desktops
- sticky windows
- virtual desktops
- pagers
- taskbars
- activation
- animated iconification
- window-in-window MDI
- layered stacking order

We will briefly discuss each of them in turn.

#### Additional states

As per the ICCCM, the window manager is allowed to define its own state semantics. Custom states will usually be substates of the `NormalState` and `IconicState` states.

Two examples of common self-defined states are `MaximizedState` and `ShadedState`. A maximized window should fill the screen as much as is possible (this does not necessarily have to be the entire screen, as the window manager may use parts of the screen for own use, to display a status bar, for instance). Modern window managers sometimes offer horizontal and vertical maximization in addition to full maximization. Window managers should remember a window's position and size such that upon de-maximization, the window's geometry is restored. Shaded windows are windows that are *rolled up*, often implemented as an alternative to iconification. Rolled up windows show up on the screen as only a title bar (top of the frame); the window content is hidden.



## Modality

Modality is a property of window management that describes the behavior of transient windows. The `WM_TRANSIENT_FOR` property defined in the ICCCM allows a client to link a top-level window to another top-level window. An archetypal example would be a dialog box. A dialog box always belongs to another window, in the sense that it is brought into existence with the sole purpose to aid in the functioning of that other window. Some dialog boxes can be open for a long period of time, while the user continues working in the main window. Others are closed moments after being opened. Some even have to be closed before input focus is allowed to be transitioned back to the main window.

The ICCCM allows clients to implement modal windows using a globally active input model, though window managers sometimes also offer support for modality themselves.

## Large desktops

Window managers may choose to implement desktops (or workspaces) that are larger than the screen (equivalently, the root window). The screen then functions as a viewport on this large desktop. Two common policies that govern the positioning of the viewport within the desktop are *paging* and *scrolling*. Paging allows the user to only change the viewport position in increments of the screen size, while scrolling allows arbitrary repositioning.

The window manager should always interpret user-defined and client-provided size and position values relative to the current viewport, and not to the desktop as a whole. This is necessary to comply with the ICCCM requirement that all clients should behave the same, whether a window manager is running or not.

## Sticky windows

Sticky windows are windows that retain their absolute position on the screen (i.e. relative to the viewport). For large desktops, this means that the window remains in the same position, irrespective of any viewport movement.

## Virtual desktops

A user may want more than a single desktop (or workspace) to organize their windows, even if they employ only a single screen. Virtual desktops effectuate this functionality. Window managers may implement virtual desktops in different ways. Desktops may be fixed in allocation, or new ones may be created dynamically. The number of desktops may

be fixed on startup, or change dynamically as well, as the user adds or removes them from the environment. Only a single desktop will be visible at a time, and the window manager should offer the user the ability to switch between virtual desktops and move windows between them as they see fit.

If the virtual desktops are *large*, a window manager may choose to have the viewport at an independent position per desktop, or all fixed at the same position.

## Activation

Window activation in X stands for transferring input focus. When virtual desktops are employed, and a window is in a withdrawn state because it is on a desktop that is not currently active, activation will involve more than just moving input focus to that window. The window manager will first have to switch desktops (by first unmapping windows that belong to the currently active desktop, and mapping those belonging to the to-be-activated desktop), and must subsequently possibly deiconify or raise the window.

## Pagers

A pager implements a separate user interface that gives the user a *zoomed out* view of the desktop(s). Windows are represented by small rectangles, and the user is able to perform various window management actions by manipulating these representations. Such actions often include activation, moving, resizing, restacking, maximization and shading, iconification, and closing.

## Taskbars

Just as a pager, a taskbar implements a separate user interface for window management tasks. Taskbars display windows as a row of icons filled with icon windows, icon pixmaps, the title of the window, a user-defined image, or some combination of each. When pressing an icon, an action is initiated depending on the state of the window it represents. Typical actions include activation, iconification, and deiconification.

## Animated iconification

As we have seen in our discussion on compositing (Section 2.2.4), a window manager may want to add effects and animations to window management activities, such as iconification. To achieve certain animated effects, a compositor is not necessarily required. For example, a window manager might draw lines that connect the corners of the window with the

corners of the icon, or the window may be moved and resized along a trajectory, joining the window and the icon.

### Window-in-window MDI

Window-in-window MDI (for multiple document interface) is a window management technique originally developed by Microsoft for the Windows environment. A program is allowed to define a workspace within its top-level window; this workspace may contain subwindows for documents that may be open in it. These subwindows may be decorated with window manager frames as if they were children of the root window and reparented as normal. They can also be manipulated by the user within the parent window as if they were ordinary top-level windows.

### Layered stacking order

The X server manages the stacking order of windows in the environment itself. That is, which windows are to be drawn in front of or behind which other windows is ultimately determined by the X server. Of course, the window manager will want to communicate its stacking preferences to the server, and manipulate an individual window's position within the stack. Luckily, the X server offers the functionality to do so. Some window managers choose to create different *layers* within the stack. Within each layer, windows can be raised or lowered arbitrarily, but, depending on whether or not the stacking order is *strict*, a window in a lower layer can never be raised above a window in a higher one, and vice versa.

The EWMH specification aims to provide conventions to achieve the following.

- Enable clients to hint their startup preferences to the window manager, with respect to iconification, maximization, shading, stickiness, startup desktop, stacking order.
- Allow clients to hint the type of one of its windows, such that the window manager can treat it differently from regular client windows. An example of a special window type is a dock; docks should have a fixed position on screen, and should be visible on all desktops.
- Make it possible for pagers and taskbars to be implemented as regular clients and allow them to work with any compliant window manager.

These conventions allow special programs that are part of the desktop environment—such as docks,

paggers, status bars and taskbars—to be implemented as regular clients. The specification also adds external control of window management actions. Thanks to these external control elements, clients are able to more directly interact with the window manager; upon startup of a program, for example, a user may be able to specify a `--desktop=n` command line argument that tells the window manager the program should start on virtual desktop *n*. As we have mentioned, these conventions are defined in protocol operations, just as those of the ICCCM. The EWMH specifies two types of protocol definitions: properties and messages on client windows, and properties and messages on the root window.

### 4.2.2 Root window protocol definitions

The EWMH specifies that a protocol definition on the root window is a property that is initially set by the window manager, and is subsequently read by clients that require information about the environment and window management policy. Some clients are allowed to change properties by sending events to the root window. It is up to the window manager to decide whether or not to accept the change.

To send a message to the root window, the specification stipulates that the client should create a `ClientMessage` event and send it using the `SendEvent` request with the following arguments.

field	value
destination	the root window
propagate	False
event mask	SubstructureNotify   SubstructureRedirect
event	<b>the <code>ClientMessage</code> event</b>

As per the EWMH, the most important properties on the root window relating to window management are the following.

- `_NET_SUPPORTED`
- `_NET_CLIENT_LIST`
- `_NET_NUMBER_OF_DESKTOPS`
- `_NET_DESKTOP_GEOMETRY`
- `_NET_DESKTOP_VIEWPORT`
- `_NET_CURRENT_DESKTOP`
- `_NET_DESKTOP_NAMES`
- `_NET_ACTIVE_WINDOW`
- `_NET_WORKAREA`
- `_NET_SUPPORTING_WM_CHECK`
- `_NET_VIRTUAL_ROOTS`
- `_NET_SHOWING_DESKTOP`
- `_NET_CLOSE_WINDOW`
- `_NET_MOVERESIZE_WINDOW`



- `_NET_WM_MOVERESIZE`
- `_NET_RESTACK_WINDOW`

We will briefly discuss each of them in turn.

### `_NET_SUPPORTED`

The `_NET_SUPPORTED` property is of type `ATOM[]` and defines a list of atoms. It holds the full set of hints and features the window manager supports. For instance, if a window manager supports a subset of the states defined in the EWMH specification, say `HiddenState`, `ShadedState` and `StickyState`, both the `_NET_WM_STATE` atom and the atoms representing the states (`_NET_WM_STATE_HIDDEN`, `_NET_WM_STATE_SHADED`, `_NET_WM_STATE_STICKY`) should be in the list.

### `_NET_CLIENT_LIST`

The `_NET_CLIENT_LIST` property represents a list of all clients under control of the window manager. That is, each top-level client window that is being managed by the window manager must be in this list. Its type is therefore `WINDOW[]`. The list should be in initial mapping order, starting with the oldest (first mapped) window.

Another property of the same nature, `_NET_CLIENT_STACKING_LIST`, contains all windows in bottom-to-top stacking order. Both properties should be set and updated by the window manager.

### `_NET_NUMBER_OF_DESKTOPS`

The `_NET_NUMBER_OF_DESKTOPS` property is of type `CARD32` and is to be set and updated by the window manager to denote the number of virtual desktops.

A pager may request a change in the number of virtual desktops by sending a `ClientMessage` event to the root window containing the `_NET_NUMBER_OF_DESKTOPS` message type and the new number of desktops. A window manager may choose to accept or deny this request. If it is accepted, the `_NET_NUMBER_OF_DESKTOPS` and `_NET_VIRTUAL_ROOTS` properties must change accordingly, and so must `_NET_DESKTOP_VIEWPORT` and `_NET_WORKAREA`. If the number of desktops has decreased and `_NET_CURRENT_DESKTOP` contains a value not in the range of desktop numbers, then this property must be set to the last value in the range and `_NET_WM_DESKTOP` must be updated.

### `_NET_DESKTOP_GEOMETRY`

The `_NET_DESKTOP_GEOMETRY` property has type `CARD32[2]` and represents an array of two values

that together define the common size of all desktops. This can be the size of the large virtual desktop or the screen size if large desktops are not implemented.

A pager may request a desktop resize by sending a `ClientMessage` event to the root window containing the `_NET_DESKTOP_GEOMETRY` message type and new width and height values. A window manager may choose to accept or deny this request.

### `_NET_DESKTOP_VIEWPORT`

The `_NET_DESKTOP_VIEWPORT` property holds an array of pairs, and is therefore of type `CARD32[] [2]`. These pairs represent  $x$  and  $y$  values that define the top-left corner of each desktop's viewport. Of course, if large desktops are not implemented, this array will contain  $n$  pairs that are set to 0, 0, where  $n$  is the number of desktops.

A pager may request to move the current desktop's viewport by sending a `ClientMessage` event to the root window containing the `_NET_DESKTOP_VIEWPORT` message type and new  $x$  and  $y$  values.

### `_NET_CURRENT_DESKTOP`

The `_NET_CURRENT_DESKTOP` property is of type `CARD32` and contains the index of the currently active desktop. An index is a value between 0 and `_NET_NUMBER_OF_DESKTOPS - 1`. A window manager must set and update this value.

A pager may request to switch to another virtual desktop by sending a `ClientMessage` event to the root window containing the `_NET_CURRENT_DESKTOP` message type and the new index value. A window manager may choose to accept or deny this request.

### `_NET_DESKTOP_NAMES`

The `_NET_DESKTOP_NAMES` property (of type `TEXT[]`) is a list of null-terminated strings containing the name of each virtual desktop. The list may be larger or smaller than the amount of desktops defined in `_NET_NUMBER_OF_DESKTOPS`. If it is larger, the strings defining names for the desktops at indices in excess of the total number of desktops will be reserved in case the number of desktops increases. If it is smaller, the desktops with no associated string will be unnamed. A window manager or pager may change these values at any time.

### `_NET_ACTIVE_WINDOW`

The `_NET_ACTIVE_WINDOW` property has type `WINDOW` and contains the window ID of the currently active (focused) window, or `None` if no window is in

focus. In principle, this is a read-only property that is set and updated by the window manager.

A client may request to transfer focus to another window by sending a `ClientMessage` event containing the `_NET_ACTIVE_WINDOW` window type, a source indication value (1 if the request is from a regular application, and 2 if it is from a pager), and the window ID of the requesting client's active top-level window. The window manager may choose to completely ignore the request, or set the `_NET_WM_STATE_DEMANDS_ATTENTION` property.

### `_NET_WORKAREA`

The `_NET_WORKAREA` property is of type `CARD32[] [4]`, meaning it holds a list of four-tuples. Each tuple represents the geometry (position and size) relative to the viewport of an area within the viewport. This workarea must be used by desktop applications to place their desktop icons. A tuple must be defined for each virtual desktop.

### `_NET_SUPPORTING_WM_CHECK`

The `_NET_SUPPORTING_WM_CHECK` property (of type `WINDOW`) is used by the window manager to communicate to the environment that an EWMH-compliant window manager is running. To do this, the window manager creates a new child window of the root window upon startup, and stores that window's ID in this property on the root window. The child window does not necessarily need to be visible on screen, and must store its own window ID in its `_NET_SUPPORTING_WM_CHECK` property. The window manager must also set the `_NET_WM_NAME` on this child window to set the name of the window manager.

### `_NET_VIRTUAL_ROOTS`

The `_NET_VIRTUAL_ROOTS` property holds IDs of windows that act as root windows for each of the virtual desktops. Its type is therefore `WINDOW[]`.

This property allows programs that change the the desktop background to function with virtual root windows, and allows clients to find out which frame belongs to their top-level window (if a reparenting window manager is running).

### `_NET_SHOWING_DESKTOP`

The `_NET_SHOWING_DESKTOP` property is of type `CARD32` and is used by the window manager to communicate to the environment whether or not the *showing desktop* mode is activated, in which all windows are hidden, and the (virtual) root is the only visible window. A window manager only has to set

this property if it supports this feature. It then sets the value in this property to 1 if this mode is activated, and 0 if it is not.

A pager may request to enter or leave showing desktop mode by sending a `ClientMessage` event to the root window containing the `_NET_SHOWING_DESKTOP` message type and the new boolean value. A window manager may choose to accept or deny this request.

### `_NET_CLOSE_WINDOW`

The `_NET_CLOSE_WINDOW` request is used by the pager or normal applications to request for a certain window to be closed. A window manager must then attempt to close the targeted window.

Normally, to request a window to close, the client sends a `WM_DELETE_WINDOW` message to the application if that protocol is supported, and otherwise an `XKillClient` message should be sent. Because the window manager may have more information about the application that is to be closed, the requesting client should always send the `_NET_CLOSE_WINDOW` request before trying anything else.

### `_NET_MOVERESIZE_WINDOW`

The `_NET_MOVERESIZE_WINDOW` request is used by the pager or normal applications to request a window to be moved, resized or both.

In the request, the requesting client specifies the source indication (regular application or pager), the change flags (whether to only change the window's position, only its size, or both), window gravity, and position and size values. The values that can be supplied for the window gravity are: `NorthWest`, `North`, `NorthEast`, `West`, `Center`, `East`, `SouthWest`, `South`, `SouthEast` and `Static`. When a value of 0 is supplied, the window manager should use the gravity supplied in the `WM_NORMAL_HINTS` property.

This request may be used in place of the `ConfigureRequest` request. The only reason a client would choose this request over the `ConfigureRequest` is if it has a reason to specify the window gravity. Pagers especially benefit from using this request, as with it, a window can be exactly positioned and resized regardless of any framing (window decorations), by specifying `Static` window gravity.

### `_NET_WM_MOVERESIZE`

The `_NET_WM_MOVERESIZE` request is used by clients to ask the window manager to initiate window movement or resizing. With this request, programs can

have elements in their own application interfaces that the user can use to move or resize the window.

A client will benefit more from using this request to hand off movement and resizing to the window manager, rather than using the `_NET_MOVERESIZE_WINDOW` or the `ConfigureRequest` requests to move or resize the window by itself. To the user, this also makes movement and resizing more consistent.

In the request, the client must supply which button was pressed to initiate the movement or resizing, the position of the button press with respect to the virtual root, whether it is a move or resize event, and to which edges of the window the event applies to (only relevant if it is a resize event).

### `_NET_RESTACK_WINDOW`

The `_NET_RESTACK_WINDOW` request is used by a pager to indicate to the window manager that a window should be restacked. This request is similar to a `ConfigureRequest` request with the `CWSibling` and `CWStackMode` flags set. It should only be used by pagers, and not by regular clients; regular clients should use the `ConfigureRequest` request instead.

## 4.2.3 Client window protocol definitions

As we have seen, the window manager requires information about applications to manage them correctly. A dock, for instance, should not be reparented, nor should it be able to be moved manually by a user. Applications should set certain properties on their top-level windows to communicate information about them to the window manager, such as its type (dock, dialog box, ...), the desktop it wishes to start on, its state, and so on.

The most important properties that the EWMH stipulates applications should set on their top-level windows are the following.

- `_NET_WM_NAME`
- `_NET_WM_VISIBLE_NAME`
- `_NET_WM_ICON_NAME`
- `_NET_WM_VISIBLE_ICON_NAME`
- `_NET_WM_DESKTOP`
- `_NET_WM_WINDOW_TYPE`
- `_NET_WM_STATE`
- `_NET_WM_ALLOWED_ACTIONS`
- `_NET_WM_STRUT`
- `_NET_WM_STRUT_PARTIAL`
- `_NET_WM_ICON_GEOMETRY`
- `_NET_WM_ICON`
- `_NET_WM_PID`
- `_NET_WM_HANDLED_ICONS`
- `_NET_FRAME_EXTENTS`

We will briefly discuss each of them in turn.

### `_NET_WM_NAME`

The `_NET_WM_NAME` property is of type `TEXT` and contains the title of the window. The EWMH states the window manager should use this property in preference to the `WM_NAME` property.

### `_NET_WM_VISIBLE_NAME`

The `_NET_WM_VISIBLE_NAME` property has type `TEXT`, and should contain the title of the window as displayed by the window manager. If the window manager does not change the title of the window in any way, the string in this property should be the same as in `_NET_WM_NAME`. If the window manager displays a custom title somewhere on the screen, this property should contain that string. This property may be useful to pagers that want to display the same title for each window as the window manager does.

### `_NET_WM_ICON_NAME`

The `_NET_WM_ICON_NAME` property is similar to the `_NET_WM_NAME` property. It has type `TEXT` and should contain the title of an application. The EWMH states the window manager should use this property in preference to the `WM_ICON_NAME` property.

### `_NET_WM_VISIBLE_ICON_NAME`

The `_NET_WM_VISIBLE_ICON_NAME` property is similar to the `_NET_WM_VISIBLE_NAME` property. It is of type `TEXT` and should contain the title of the window as displayed by the window manager.

### `_NET_WM_DESKTOP`

The `_NET_WM_DESKTOP` property has type `CARD32` and should contain a value that represents the virtual desktop the client wishes to start in. The value zero represents the first desktop, one is the second desktop, and so on. A client may of course choose not to set this property; the window manager then decides where to place it itself. The window manager should always honor this request, if possible.

### `_NET_WM_WINDOW_TYPE`

The `_NET_WM_WINDOW_TYPE` property (of type `ATOM[]`) is an important application window property that holds the *type* of a window. The EWMH recognizes eight basic window types:

window type	atom
normal	<code>_NET_WM_WINDOW_TYPE_NORMAL</code>
desktop	<code>_NET_WM_WINDOW_TYPE_DESKTOP</code>
dialog	<code>_NET_WM_WINDOW_TYPE_DIALOG</code>
dock	<code>_NET_WM_WINDOW_TYPE_DOCK</code>
menu	<code>_NET_WM_WINDOW_TYPE_MENU</code>
toolbar	<code>_NET_WM_WINDOW_TYPE_TOOLBAR</code>
splash	<code>_NET_WM_WINDOW_TYPE_SPLASH</code>
utility	<code>_NET_WM_WINDOW_TYPE_UTILITY</code>

This property should always be used by the window manager to determine where to place the window, which decorations to give it (if reparenting happens), how users are able to interact with the window, its stacking position, and whether or not the window is visible on all virtual desktops, on a select few of them, or perhaps on none of them at all.

A client is allowed to specify more than a single type of window, some of which may be extensions to the basic types named above. The types it lists must be in order of preference, the first being the most preferable. Because a window manager may not support any extensions to the basic types, a client must always specify at least one basic window type.

*Normal* windows are regular, top-level windows that do not have a different (acknowledged) type. Windows that do not set the `_NET_WM_WINDOW_TYPE` property nor `WM_TRANSIENT_FOR` must always be taken as this type by the window manager.

*Desktop* windows are windows that implement some type of desktop feature. This may be a single window with the same dimensions as the root window that contains desktop icons. This would allow the desktop environment to be in full control of the desktop, without the need to proxy root window events.

*Dialog* windows are windows that act as dialog boxes. If a dialog box does not set this property, then the `WM_TRANSIENT_FOR` property must be set by the client to achieve the desired effect.

*Dock* windows are windows that implement a dock or panel feature. A window manager would mostly want to keep these windows on all virtual desktops and have them rendered above all other windows (i.e. keep them at the top of the stack).

*Menu* and *toolbar* windows are windows seemingly popped out from the main application window. These windows may also set the `WM_TRANSIENT_FOR` property to indicate the main application window.

*Splash* windows are splash screens that are briefly displayed as an application is starting up. Such windows should usually be centered on the screen.

*Utility* windows are small persistent windows such as palettes or toolboxes. A utility window is different from a toolbar because it is not popped out of the main application window. It is also different from a dialog window because it is not a transient dialog box; the user will usually keep it open for the entire lifetime of the application. These windows may also set the `WM_TRANSIENT_FOR` property to indicate the main application window.

## `_NET_WM_STATE`

The `_NET_WM_STATE` property (of type `ATOM[]`) lists an array of hints describing the window state. All atoms that are in the array must be considered set by the window manager, and those not in the array are unset. That is, the atoms in the array represent states that accumulatively apply to the client's window. The window manager must always honor the state of a window and treat it accordingly, if possible.

A client may wish to change its state during operation. It can notify the window manager of its state change by sending a `ClientMessage` event containing the `_NET_WM_STATE` message type and the new state to the root window. The window manager must always keep the state of the windows it manages updated. The window manager should remove the property upon withdrawal of the window. The EWMH recognizes twelve basic window states:

window state	atom
modal	<code>_NET_WM_STATE_MODAL</code>
sticky	<code>_NET_WM_STATE_STICKY</code>
max. (V)	<code>_NET_WM_STATE_MAXIMIZED_VERT</code>
max. (H)	<code>_NET_WM_STATE_MAXIMIZED_HORZ</code>
shaded	<code>_NET_WM_STATE_SHADED</code>
skip taskbar	<code>_NET_WM_STATE_SKIP_TASKBAR</code>
skip pager	<code>_NET_WM_STATE_SKIP_PAGER</code>
hidden	<code>_NET_WM_STATE_HIDDEN</code>
fullscreen	<code>_NET_WM_STATE_FULLSCREEN</code>
above	<code>_NET_WM_STATE_ABOVE</code>
below	<code>_NET_WM_STATE_BELOW</code>
dem. attn.	<code>_NET_WM_STATE_DEMANDS_ATTENTION</code>

Just as with window types, the window manager may recognize extensions to these basic window states. A window manager must ignore atoms it does not recognize, effectively removing it from the array of window states.

A window in the *modal* state indicates that that window is a modal dialog box. If the `WM_TRANSIENT_FOR` property is set, the value in it is the ID of the window that it is a dialog box for. If

it is not set or set to the root window, the window is a dialog box for its window group.

A window in the *sticky* state means that that window is sticky, as described in Section 4.2.1.

A window in the *maximized* state, either vertical or horizontal, indicates that that window is stretched along the entire height or width of the screen respectively.

A window in the *shaded* state is rolled up, as described in Section 4.2.1.

A window in the *skip taskbar* state indicates that it should not be included in the taskbar. This hint should be set by the application, and not by the window manager. It should only be set if the `_NET_WM_WINDOW_TYPE` property does not already communicate the nature of the window.

A window in the *skip pager* state indicates that it should not be included on a pager. It should only be set if the `_NET_WM_WINDOW_TYPE` property does not already communicate the nature of the window.

A window in the *hidden* state is not currently visible on screen, even if it were on the currently active desktop and within the viewport. This state is set by the window manager when it unmaps the window. Minimized windows, for instance, must be in this state.

A window in the *fullscreen* state should fill the entire screen, except parts that are reserved by the window manager. Upon leaving this state, the window manager is expected to restore the geometry the window had before it entered the fullscreen state.

A window in the *above* or *below* state means that that window should be rendered before or after most other windows, respectively. That is, it should respectively be at a low or high position in the stack. These two states are meant to be set by the user when stating their preferences to the window manager. A program must not, for instance, use these states to draw attention to dialog windows.

A window in the *demands attention* state indicates that some action in or with the window has happened, and it requires user attention. For instance, the window manager will put a window in this state if that window requested activation, but the window manager refused it. Both the client and the window manager are allowed to set this state. This state should be removed from the window manager when it got the required attention.

#### `_NET_WM_ALLOWED_ACTIONS`

The `_NET_WM_ALLOWED_ACTIONS` property is a list of atoms (hence of type `ATOM[]`) that each represent a window management action that the window wants allowed. That is, the list enumerates user operations that should be allowed on the window; op-

erations not in the list should not be allowed by the window manager. The window manager must keep this list updated, as taskbars, pagers and other such tools use this property to decide which actions are allowed on a window.

The EWMH recognizes the following user operations:

user operation	atom
move	<code>_NET_WM_ACTION_MOVE</code>
resize	<code>_NET_WM_ACTION_RESIZE</code>
minimize	<code>_NET_WM_ACTION_MINIMIZE</code>
shade	<code>_NET_WM_ACTION_SHADE</code>
stick	<code>_NET_WM_ACTION_STICK</code>
maximize (H)	<code>_NET_WM_ACTION_MAXIMIZE_HORZ</code>
maximize (V)	<code>_NET_WM_ACTION_MAXIMIZE_VERT</code>
fullscreen	<code>_NET_WM_ACTION_FULLSCREEN</code>
change desktop	<code>_NET_WM_ACTION_CHANGE_DEKSTOP</code>
close	<code>_NET_WM_ACTION_CLOSE</code>

The operations listed are those that the window manager will honor for this window. A client must request the operation to be performed through the normal mechanisms discussed earlier. For instance, if a window supports the close action (`_NET_WM_ACTION_CLOSE`), this does not imply that clients can send a `WM_DELETE_WINDOW` message directly to remove the window from the environment; supporting this action means clients can make a request to close it through the window manager by sending a `_NET_CLOSE_WINDOW` message. The window manager subsequently decides if and how to honor the request, perhaps by sending a `WM_DELETE_WINDOW` request to the window first, and if it does not respond, kill it using the `XKillClient` message.

Window managers are free to recognize extensions to these window manager actions. A window manager must ignore atoms it does not recognize, effectively removing it from the array of window states.

The *move* action indicates that the window may be moved around the screen.

The *resize* action indicates that the window may be resized. Of course, if its minimum and maximum size are identical (as specified in the `WM_NORMAL_HINTS` property), the window manager knows this window is non-resizable.

The *minimize* action indicates that the window may be iconified (Section 4.2.1).

The *shade* action indicates that the window may be shaded (Section 4.2.1).

The *stick* action indicates that the window may be put into the sticky state (Section 4.2.1).

The *maximize* (horizontal or vertical) action indicates that the window may be maximized horizontally or vertically, respectively.

The *fullscreen* action indicates that the window may be put into the fullscreen state.

The *change desktop* action indicates that the window may be moved between virtual desktops.

The *close* action indicates that the window may be closed, that is, a `WM_DELETE_WINDOW` message may be sent.

#### **\_NET\_WM\_STRUT and \_NET\_WM\_STRUT\_PARTIAL**

The `_NET_WM_STRUT_PARTIAL` property is an important property used by desktop environment programs such as docks and status bars to indicate which region along the edge of the screen should be reserved for it to function properly. The information in this property is crucial to window managers, as it needs to know, for example, where to stop tiling windows, or which parts of the screen not to cover when maximizing a window. The property is of type `CARD32[12]` and contains four values that specify the width of the reserved area at each border of the screen, and an additional eight values that specify the beginning and end corresponding to each of the four struts. The start and end values of each strut allow the client to specify areas that do not span the entire width or height of the screen. All coordinates specified are relative to the (actual) root window. A client may change this property at any time, and so the window manager must watch for events that notify it of property changes.

The `_NET_WM_STRUT` property is equivalent to the `_NET_WM_STRUT_PARTIAL` property where all start values are 0, and all end values are the height or width of the screen. If a client has set both the `_NET_WM_STRUT_PARTIAL` and `_NET_WM_STRUT` properties, the window manager must acknowledge the former and ignore the latter.

#### **\_NET\_WM\_ICON\_GEOMETRY**

The `_NET_WM_ICON_GEOMETRY` property is of type `CARD32[4]` and contains the geometry (position and size values) of a possible icon for the window, if it is ever iconified. Desktop environment tools like taskbars or icon boxes are allowed to set this value, though it is optional. Window managers may use this property to perform animations such as the stretching of the window into its icon.

#### **\_NET\_WM\_ICON**

The `_NET_WM_ICON` property (of type `CARD32[] [2+n]`) lists an array of  $n$  possible icons

for the window. The window manager is allowed to scale these icons as it pleases.

#### **\_NET\_WM\_PID**

The `_NET_WM_PID` property has type `CARD32` and holds the process ID of the application that owns the window. This ID is sometimes used by the window manager to kill an application if it does not respond to the `_NET_WM_PING` protocol.

#### **\_NET\_WM\_HANDLED\_ICONS**

The `_NET_WM_HANDLED_ICONS` property may be used by pagers or taskbars on one of its own top-level windows to indicate to the window manager that it does not need to provide icons for iconified windows.

#### **\_NET\_FRAME\_EXTENTS**

The `_NET_FRAME_EXTENTS` property (of type `CARD32[4]`) contains a four-tuple of values that determine the frame protrusions along the four edges of the window. That is, it defines the size of the window decoration (if reparenting occurs) at each edge of the window. This property must be set by the window manager.

The EWMH additionally suggests implementation conventions on some of the subjects discussed in Section 4.2.1 and more. An interested reader is referred to the specification itself for more information on how to implement window manager features such as pagers and taskbars, how to kill hung processes and how to maintain the stacking order.

# Chapter 5

## Agile Software Development

Software development approaches have significantly changed throughout the years. It used to be common to work along so-called *heavyweight* methodologies, like the *Waterfall model* or the *Spiral model*<sup>[35]</sup>. These traditional methodologies focus on detailed documentation, inclusive planning, and highly structured design<sup>[35]</sup>. They have a relatively linear, sequential design regime with pre-organized phases of the development lifecycle<sup>[35]</sup>. The flow of development in such methodologies is said to be unidirectional, as the developers progress in order through requirements, design, development, testing, and maintenance<sup>[36]</sup>. In many of these traditional approaches, each phase has a fixed deliverable and detailed documentation, both of which are subjected to rigorous review<sup>[35]</sup>. They tailor well to products and environments in which requirements are precisely defined and largely immutable (e.g. the construction industry). The IT industry, in stark contrast, is continually changing; it is often near impossible to define every requirement well ahead of time. Employing a heavyweight methodology means that the solution is to be detailed in advance, without the ability to change the requirements once development has begun<sup>[35]</sup>.

To tackle the aforementioned shortcomings, *lightweight* or *Agile* methodologies came into existence<sup>[35,36]</sup>. Unlike their heavyweight counterparts, Agile approaches revolve around the product owner, are precise and unambiguous, and above all, allow for modifications to be made to the product throughout the phases of development<sup>[35]</sup>. As a result, these methodologies tend to propose an incremental and iterative line of action, in which the product or solution is repeatedly refined<sup>[35]</sup>.

### 5.1 Agile Manifesto

The *Manifesto for Agile Software Development* is a proclamation backed by IT consultants and other industry professionals (who collectively called themselves the Agile Alliance) that articulates several *values* and *principles* that should guide software developers in their work<sup>[37]</sup>. The manifesto is in actuality an objection to the heavyweight development approach, which they perceived to be cumbersome, unresponsive, and too focused on documentation requirements<sup>[37]</sup>.

The values enumerated in the manifesto are said to promote a software development process that focuses on quality by creating a product that meets

the consumer's needs and expectations. The principles are there to facilitate a working environment that focuses on the customer, adheres to business objectives, and is sufficiently resilient to changing user needs and market forces, for change is the only constant<sup>[37]</sup>.

The Agile approach to software development commits to creating software in an incremental manner, in the sense that the development lifecycle is divided into relatively brief periods of time, often called *sprints*<sup>[38]</sup>. Splitting up product development work into increments in this manner is meant to minimize up-front planning and design<sup>[38]</sup>. Each sprint typically lasts one to four weeks, and involves cross-functional teams working in all of the functions relevant to the development process (e.g. planning, design, programming, and testing)<sup>[38]</sup>.

Agile is but a philosophy, a reasoning about software development relationships and priorities. It does not specify a particular implementation such as to achieve its goals; it does not identify means to the desired end<sup>[37]</sup>. Several methodologies and frameworks have since been designed to formalize many or all of the values and principles expounded in the Agile Manifesto. One of the most popular such implementations is *Extreme Programming*<sup>[39]</sup>, which we will briefly discuss in Section 5.2.

We will furthermore discuss the *Personal Software Process* development process<sup>[40]</sup> in Section 5.3, as it has been used to apply *Personal Extreme Programming*<sup>[41]</sup>, a modified version of the Extreme Programming methodology (discussed in Section 5.4) throughout Chapters 6 and 7.

### 5.2 Extreme Programming

Extreme Programming (or *XP*) is an Agile software development methodology that was initially developed in response to problem domains with changing requirements<sup>[39]</sup>. Because it implements the Agile philosophy, Extreme Programming accepts that requirements will change, and creates opportunities for improvement and competitive advantage<sup>[39]</sup>.

Each feature the customer wants is represented as a *user story*<sup>[42]</sup>. User stories are written by the customer as things that the system needs to do for them<sup>[42]</sup>. They are similar to usage scenarios, except that they are not limited to describing a user interface<sup>[42]</sup>. Each story is in the format of a few sentences ideally written by the customer, in the cus-



tomer's terminology, without technical jargon<sup>[42]</sup>.

One of the most important key values in Agile software development methodologies is communication about the software<sup>[37]</sup>. The use of a *system metaphor* is a powerful tool to facilitate this<sup>[43]</sup>. The system metaphor is a type of user story that is ultimately used to create the overall product design<sup>[39]</sup>. It helps each member of the team and the customer understand and contribute to this design. As such, the metaphor need not necessarily relate directly to the product being developed<sup>[39]</sup>. A customer, for example, may very well find it easier to talk about a "chameleon" than about a window that should reactively change its transparency<sup>[43]</sup>. It is there to help the team and the customer establish a common vocabulary and reach agreement about the requirements<sup>[43]</sup>. The metaphor is consequently often used in conversations amongst developers and with the customer<sup>[43]</sup>.

XP additionally uses test driven development and iterative refactoring to produce a working and well-defined demonstration at the end of each sprint<sup>[39]</sup>. High code quality is essential on an XP project. Rules that enhance quality include pair programming, refactoring, a sustainable pace, and full test coverage at two different levels: unit tests and acceptance tests<sup>[39]</sup>.

Developers constantly receive feedback by working in pairs and testing code as it is written, managers get feedback at daily stand-up meetings, which are nothing more than short meetings in which attendees make commitments to team members and otherwise participate while standing, and customers get feedback with test scores and demos at the end of each sprint<sup>[39]</sup>.

The values and rules of Extreme Programming are summarized into twelve *core practices* that every XP programmer should follow<sup>[39]</sup>.

1. **The planning game.** There is constantly a back and forth negotiation of user stories between the customer and the developers.
2. **Small releases.** At the end of each iteration, a working system containing a supplementary set of features is put into production.
3. **Metaphor.** Each project has a system metaphor, which helps guide the development process and communication between all parties.
4. **Simple design.** The most simple design possible is used to build the product at all times, provided that all business requirements are met.
5. **Testing.** Always apply a test-driven development approach; every snapshot of the product must be validated in its entirety before starting work on the next feature.
6. **Refactoring.** If necessary to keep the codebase as simple as possible, apply a series of behavior-preserving transformations to improve the structure of the code.
7. **Pair programming.** Programmers are paired up and perform all development work together at the same computer.
8. **Collective ownership.** All code is shared amongst every member of the development team; anyone can make changes to the codebase at any time.
9. **Continuous integration.** Changes are integrated into the main codebase as frequently as possible. After an integration, tests are run and must pass.
10. **40-hour week.** Programmers adhere to a 40-hour work week in order to maintain productivity and to avoid burn out.
11. **On-site customer.** The customer is at all times available to the development team, and must help guide the development process.
12. **Coding standard.** Every programmer uses the same coding standards.

These practices incarnate the values and principles enumerated by the Agile manifesto, and thereby help developers prioritize the customer, communicate about the software and the requirements, increasingly refine the product by adding features and keeping the codebase minimal (by preventing code bloat), and in general keep defects and loss of productivity at a minimum<sup>[39]</sup>.

The Extreme Programming process is geared towards groups in that it specifies interactions between the different functions within a team. As we shall see in Section 5.4, an adaptation of XP exists that defines rules for applying the methodology in a one-person team.

### 5.3 Personal Software Process

The *Personal Software Process* (or *PSP*) is a structured software development process that is meant to provide engineers with a disciplined personal framework for doing software work<sup>[40]</sup>. It aims to improve the quality and productivity of the personal work of the software engineer as an individual. To this end, it consists of methods, forms, and scripts that show



software engineers how to plan, measure, and manage their work<sup>[40]</sup>. The PSP uses a disciplined, data-driven procedure to enable engineers to improve their estimating and planning skills, to make realistic commitments, to manage the quality of a project, and to reduce the number of defects in their work<sup>[40]</sup>. It was introduced in a paper by Watts Humphrey at Carnegie Mellon University, which we will be referencing throughout the rest of this section<sup>[40]</sup>.

Engineers are to follow a process of three phases: *planning*, *development*, and *postmortem*. This process is applied to all development *tasks*. A task is an iteration through this process, and most software will comprise a large number of them. Input to each task is the *problem description*. If the problem description for one task is too small or trivial, multiple problems can together form a single task. The three phases are organized as follows.

#### ■ Planning

1. Create a *requirements statement* based on the problem description. A requirements statement is a simple list of requirements that the task must satisfy.
2. Begin setting up a *project plan summary*, which is a snapshot of the progress of a task. At the end of this phase, it should contain information about the *planned* changes to the codebase; this information is mostly revolved around lines of code (such as the number of additions, deletions, and modifications), though it also includes defects found and resolved.
3. Make a time estimation of each of the phases (including the current phase) in a *time recording log*, based on previous iterations.

#### ■ Development

1. Produce a design that resolves the problem, and document it.
2. Implement the design.
3. Test the implementation.
4. Supplement the time recording log with information gathered from this phase.

#### ■ Postmortem

1. Compare estimated duration with actual duration.
2. Record and review new duration and defect rates.
3. Complete the project plan summary.

A developer should first apply the rules that are part of the planning phase. This involves identifying

the requirements and properly documenting them, logging task progress, and making time estimations for the current iteration. The second phase pertains to the development process. In it, the developer is to work on the product by creating or refining the design, by consequently implementing any changes, and by making sure tests covering the implementation succeed and are complete. The final phase serves as a reflection on the progress made throughout the current iteration and on the development process as a whole.

Throughout the iterations of development, at every phase apart from the first, any defects found must be recorded in a *defect log* and subsequently fixed.

## 5.4 Personal XP

As we have discussed in Section 5.2, Extreme Programming is geared towards groups. It lays out rules that stipulate group processes like stand-up meetings and user story negotiation, and many of the practices involved require teamwork, as production code is written, refactored, and tested in teams of two programmers who constantly review each other's work. To be able to apply, as an individual, the Agile software development philosophy and specifically the rules and practices defined by the Extreme Programming methodology, the *Personal Extreme Programming* (or *PXP*) methodology was developed. The rest of this section summarizes the ideas established in the paper that introduced the methodology<sup>[41]</sup>.

In PXP, all core practices in XP that are trivially translatable to a single-person situation are left unchanged. The following practices are not directly applicable by an individual and hence differ from XP.

1. **The planning game.** If the developer is working for themselves or otherwise has enough information to stand in for the customer, then the developer only needs to alternately switch roles during the planning phase. Even if the customer is not involved in this phase, the developer should write user stories in the usual format all the same.
7. **Pair programming.** The benefits of pair programming are obviously lost when working by oneself. A programmer could apply the so-called *rubber duck debugging* technique<sup>[44]</sup>, ask a colleague or loved one for reminders, and should otherwise apply informal walkthroughs and a test first philosophy.

11. **On-site customer.** When working alone, projects are often not large enough to justify

the customer being available at all times, and communication through email or phone will suffice. Demos should still be prepared and given at the end of each iteration, even if the developer is working for themselves.

With alterations to three of the guiding practices of Extreme Programming—namely those that consist of the ulterior requirement that the developer applying it is working with others in a single team—a viable foundation is established with which a developer can apply the Agile paradigm when working by themselves.

# Chapter 6

## Implementation Process

In this chapter, we will discuss the process gone through to realize the implementation of the window manager detailed in Chapter 7. We have mainly applied the software development approaches described in Chapter 5. Specifically, the Personal Extreme Programming methodology was at all times adhered to during the development process.

Specialized versions of the twelve practices set forth in the PXP paradigm were applied throughout. For each of these practices, we will describe the specifics of their application to the development of our window manager.

### 6.1 The planning game

*User stories are written in user terms and contain minimal technical jargon.*

While they do not contain technical details of the window manager's internal architecture per se, our user stories do contain terminology common to the X ecosystem, such as references to specifics in keyboard and mouse handling (Section 3.12). There is no way around these references, as they directly pertain to the manner in which a user wishes to be in control of the different window management features. Unlike most window managers, ours is targeted at developers and hobbyists (collectively referred to as *power users*) who additionally want to be able to describe how to *control* the software.

Consider the following user story. It requires the window manager to allow the user to move windows around by using a key on the keyboard in conjunction with the mouse.

*As a user, I can press down the main modifier key on the keyboard, left click the window I want to move, and subsequently move that window by moving the mouse. Releasing the left mouse button will stop moving the window in question.*

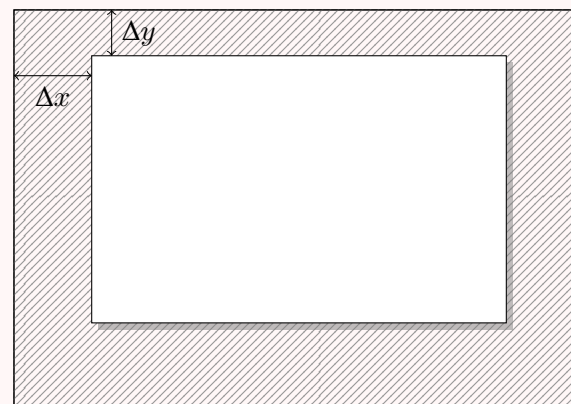
Surely, without such details, the user would not be able to precisely define the usage parameters that they require to reach that certain level of control. The need for a higher degree of granularity is especially prevalent amongst projects aimed at power users, as they often need to be given an extra layer of complexity in order to achieve a more efficient working situation than the average user.

A list of user stories was compiled up front, and

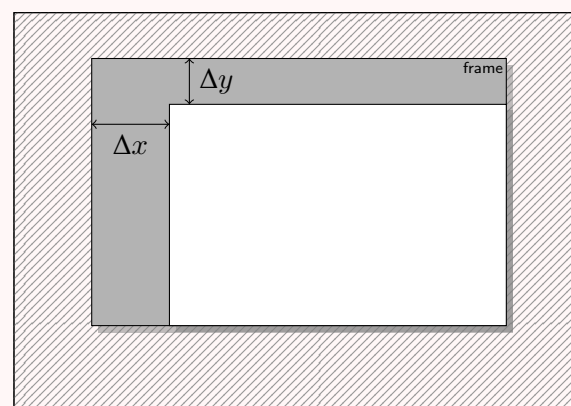
more were added later on as the need for them arose. Features that allow for the direct manipulation of window management concepts such as windows, workspaces, and key and mouse bindings are all narrated in user stories.

Bugs encountered during the development process, and issues otherwise related to convenience and consistency were not documented in user stories, as they require a more technical, elaborate, and precise wording. To this end, so-called *issues* were created, similar to those found in common version control solutions such as GitHub<sup>†</sup>. When applicable, they will often contain a sketch of the problem. An example of such an issue is the following.

*Upon the reparenting of a window spawned by a Java-based application, subwindow positioning is incorrectly offset relative to the frame. Coordinates of the subwindow are translated relative to the origin of the frame by the distance between its requested position and the origin of the root window.*



The window's requested position



The window's frame offset

User stories and issues were logged both digitally and on paper. Paper logging was incorporated when illustrations were required to better delineate a problem. This approach did not cause any notable hindrance, as logs did not need to be reported or otherwise shared.

## 6.2 Small releases

*At the end of each iteration, a working system containing a supplementary set of features is put into production.*

Our sprints have varied in length, as changes in requirements (Section 6.1) and schedule (Section 6.10) were vetted preliminary to each iteration. Initially, two-week sprints were employed; this time frame proved suitable to the inaugural requirements gathering and project set-up processes. After the test framework was set up, a properly functioning base system was in place, and an inceptive set of important requirements was lined up, sprint durations ranged from a week to up to four weeks.

At the end of each iteration, a working snapshot of the product was put into production. Each snapshot was required to pass unit and acceptance tests that covered the entire codebase, and not just the subset that had changed throughout the sprint. Each release was accordingly *tagged*. Tags serve as markers for specific points in history. To effectuate release tagging, *Git*, our version control system of choice, was used<sup>[45]</sup>. Each tag contains the iteration number and a short message that summarizes the changes since last release. Consider the following tag. It introduces the concepts implemented in the third iteration.

*it3*

*Implements workspaces: virtual desktops that remember window arrangement, tiling layout, and several other parameters that pertain to groupings of windows. A window belongs to exactly one workspace, and exactly one workspace is active at any time. Windows can now be moved between workspaces, and the user is now able to cycle between workspaces.*

Putting a release into production involves merging any changes made with the main development branch. At all times, the main development branch must reflect the most recent stable version of the product. This implies that for the latest release, the codebase is as simple as possible (Section 6.4) and all-encompassing tests pass (Section 6.5).

## 6.3 Metaphor

*The project has a system metaphor, a high-level abstract (non-technical) description of the product's functionality.*

The following user story serves as our system metaphor.

*As a user, I want a snappy, convenient, and reliable tiling window manager. It should behave like any other modern window manager in its interaction with other applications. I want it to contain predefined tiling layouts, virtual desktops, and convenient keyboard and mouse bindings that collectively give me the means to work effectively and efficiently.*

The product does not lend itself to abstract or obscure metaphors, for its users (together the customer) are power users who are familiar with common window management concepts, and therein have specific needs and preferences.

A *snappy, convenient, and reliable* window manager is one that has a low memory footprint, does not cause impediment in its use, and does not crash or lag during conventional operation. Proper planning (Section 6.1), small releases (Section 6.2), a simple design (Section 6.4), extensive testing (Section 6.5), and a stream of small integrations (Section 6.9) were all hewed to to accommodate this.

For it to *behave like any other modern window manager in its interaction with other applications*, the window manager should comply with standardized protocols that define policy for inter-client communication. To this end, our window manager implements both the ICCCM and EWMH standard protocols (Chapter 4).

Because our product is a list-based tiling window manager, it offers its users predefined layouts along which groupings of windows are arranged. The manipulation of each such grouping is administered by a workspace. Workspaces are virtual desktops that record important information related to operations performed on the windows it encompasses. The window manager has mouse and key bindings that allow the user to effortlessly perform certain window management actions. As such, it *contains predefined tiling layouts, virtual desktops, and convenient keyboard and mouse bindings that collectively give the user the means to work effectively and efficiently*, as required.

## 6.4 Simple design

*At all times, the codebase is as simple as possible while still meeting the business requirements.*

Simple in this context does not mean short or small. It does not mean *not complicated*. Nor does it mean that everyone who is able to understand the syntax must also be able to understand the semantics. The meaning of a simple design depends on the product. A project that aims to implement airplane autopilot software, for instance, brings with it an inherent complexity that cannot be factored out by writing *less complicated* code.

To constantly keep the codebase at a most simple state, the developer must take a “simple is best” approach. Whenever a new feature is added, the author should ask themselves whether or not there is a simpler way to introduce the same functionality. In fact, this holds retroactive to the codebase as a whole. That is, the author should assess whether any part of the codebase can be made simpler, even if that part does not directly relate to the new feature. Perhaps the new feature and another part of the program use the same underlying techniques, and the problem can be generalized. Maintaining a simple design helps rule out scope creep, and empowers the developer in understanding which parts of the codebase are affected by the introduction of a new feature.

The simplification of code should not be mistaken with its optimization. A simpler design only aims to better the code, not the product. Of course, simplifying code sometimes instigates changes in running time or memory usage. If speed is an important factor, and a simplification of code significantly slows down the product, then that simplification is not deemed viable, as the business requirements are no longer all met.

As development progressed, the window manager’s codebase was continually kept as simple as possible. Certainly at the end of each sprint, and often multiple times throughout the iteration, refactoring (Section 6.6) was done to generalize shared functionality, to introduce encapsulation and polymorphism, to organize code into different compilation units, or to otherwise move functionality from one class to another.

## 6.5 Testing

*A test-driven development approach is applied; tests cover the entire codebase and must pass at the end of each iteration.*

A test-first philosophy was applied all through the development process. Unit tests for a feature were written before the code that implements it was, and acceptance tests made sure that high-level requirements were at all times met.

The Catch2 (short for *C++ Automated Test Cases in a Header*)<sup>[46]</sup> test framework was used to construct and run our tests. Catch2 is C++-native and header-only. It facilitates simple unit testing as well as test-driven and behavior-driven development. Consider the following example.

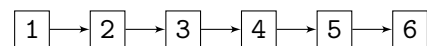
**Listing 6.1:** Focus cycle forward rotation test

```
TEST_CASE("test focus cycle", "[focus-cycle]")
{
    SECTION("full group rotation forward")
    {
        auto before = fc.get_all();
        for (size_t i = 0; i < before.size(); ++i) {
            auto pre = fc.get_all();
            fc.rotate_group_forward(0, before.size());
            auto post = fc.get_all();

            for (size_t j = 0; j < before.size(); ++j)
                REQUIRE(pre[(j+1)%before.size()] == post[j]);
        }

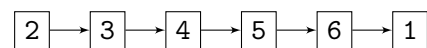
        auto after = fc.get_all();
        REQUIRE(before.size() == after.size());
        for (size_t i = 0; i < before.size(); ++i)
            REQUIRE(before[i] == after[i]);
    }
}
```

Here, tests are defined for the `focus-cycle` compilation unit. Specifically, group rotation functionality is tested on the entire window list. A rotation is a single-step counterclockwise moving of all windows in the list. Where, for instance, the window list initially looks like this:



Ordered list containing 6 windows

After the rotation, the windows in the list are ordered as follows.



Result of a single forward rotation

This test case only tests forward rotation functionality in one specific situation: all windows in the list are non-floating. Other test cases handle different situations, and together they assure that the compilation unit as a whole functions as expected. It is worth noting that test cases like the one given above are rather complex, and should never serve as the only test that covers some higher-level functionality. In fact, our test case uses several *atomic* opera-



tions on the data structure that must be tested separately, such as the `rotate_group_forward` member function.

Test cases are defined for each of the compilation units that together comprise the window manager. They are all made sure to pass whenever new functionality is added and when an iteration is brought to a close.

## 6.6 Refactoring

*If the codebase is not as simple as possible, it is refactored before moving on to the next iteration.*

Before the end of each iteration, and oftentimes throughout the lifetime of a sprint, refactoring was done to keep the codebase at a scalable and maintainable level. With scalable we mean that the requirements that are still planned are able to make its way into the codebase in a *natural* manner, that is, without requiring a change in the code responsible for other functionality, and without breaking any of the patterns or standards (Section 6.12) adhered to thus far. Maintainable code can be reliably covered by test cases, is *readable*, and logically separates concerns.

Of course, due to the Agile nature of our approach, not all requirements were known up front. Whenever a new requirement was introduced somewhere down the line, given the state of the codebase, there may not have been a scalable and maintainable way to implement it. As a result, an entire overhaul of the codebase may have been in order. Such a refactoring is a large undertaking, and may take up half if not all of the sprint to realize. It is, however, an essential part of the design process—as the Waterfall generation learned the hard way—and it cannot be replaced by any amount of upfront planning or experience<sup>[35]</sup>. *Technical debt*, a metaphor developed by Ward Cunningham, helps explain that doing things the quick and dirty way incurs interest payments (like financial debt) in the form of extra effort that needs to be done in future development because of inferior design choices<sup>[47]</sup>.

Aside from several small refactorings, we only found the need to do a large overhaul of the codebase once. This was to accommodate the drawing of window manager specific graphics (especially text) to the screen. Specifically, the refactoring involved the splitting up of a compilation unit responsible for handling data related to the X Server (such as the variable that represents the connection with the X Server) into a subsystem, separating it from the functionality specific to window management behavior (independent of any windowing system). This

change brought with it a significant change in overall code structure.

## 6.7 Pair programming

*Informal walkthroughs and rubber duck debugging are applied throughout.*

Whenever faulty program behavior was encountered, the so-called rubber duck debugging technique was applied. It involves the explaining of the problem line-by-line to oneself (not necessarily aloud) as if they were speaking with someone else<sup>[44]</sup>. This technique helps the programmer to evaluate the problem from a different perspective, and can sometimes provide deeper understanding. Most of all, it helps break the habit of forcing oneself into tunnel vision.

In addition, informal walkthroughs were given to interested friends and family members. These walkthroughs are helpful in looking at the product from an outsider's perspective, and are mainly useful for determining new requirements.

## 6.8 Collective ownership

*Multiple features may be concurrently worked on in separate branches.*

The window manager comprises many different feature categories. Window movement, user control, window decorations (framing), sidebar rendering, workspaces, and so on. While it may be a good idea to work on a single category during an iteration, sometimes bugs arise in others, or a change must be made elsewhere to facilitate the development of a particular feature. It is best to not work on these features at once—at least not in the same *version* of the code. For this, we use *branches*. Branches allow the programmer to develop features completely separate of each other, and to later merge them into the main codebase. As the development progressed, many features were implemented concurrently, as a sprint rarely saw the implementation of just a single feature.

## 6.9 Continuous integration

*Changes are integrated into the main codebase as frequently as possible. After an integration, tests are run and must pass.*

Strongly related to our single-team interpretation of collective ownership (Section 6.8), continuous integration pertains to the development of features as an isolated set of code alterations. These alterations should be relatively small, and *committed* once implemented. When a feature is completed, its dedicated branch is finalized and merged with the main

branch. Moving the development of each feature to a dedicated branch is tremendously helpful in localizing bugs, and it allows the programmer to go back to the state of the codebase before the program started exhibiting faulty behavior to inspect just what went wrong. To then pinpoint the culprit, only a relatively small part of the code has to be scrutinized. Each sprint saw tens of commits and multiple merges.

## 6.10 40-hour week

*An attempt is made to continually adhere to a 40-hour project work week.*

A good attempt was made to adhere to 40-hour work weeks (which also comprised weekends) dedicated to the development of the window manager. This includes time spent on research, design, implementation, debugging, and testing.

## 6.11 On-site customer

*Customer requirements are garnered and refined by assuming the role of an intermediate user of common tiling window managers.*

As mentioned in Sections 6.1 and 6.3, our targeted users (the customer) are power users. These users are experienced with window managers, especially with more advanced tiling window managers such as those described in Appendices A and B. As an avid user of such window managers myself, assuming the role of the customer did not pose any difficulty.

## 6.12 Coding standard

*The same coding standards are used throughout.*

Working alone means that it is relatively simple to adhere to the same coding standards throughout the development process. Even so, it is crucial to remain wary of contradicting practices that might diminish the readability of code or worse, introduce faulty program behavior. An example of the former is the use of more than one code formatting standard. If arbitrary naming conventions are chosen as the programmer sees fit, or if there is no consistency in indentation or block delimiter usage, then other developers that may want to collaborate (or future you) will have a harder time understanding the code. The latter may occur if, for instance, incorrect use of programming constructs leads to undefined behavior.

Many of the coding standards adhered to throughout the development of the window manager are those stipulated in *C++ Coding Standards*,

a seminal book written by Herb Sutter and Andrei Alexandrescu<sup>[48]</sup>. In this book, the authors list rules, guidelines, and best practices to improve software quality, to reduce time-to-market, to eliminate the wasting of time, and to simplify maintenance.

Additionally, the tried-and-true principles enumerated in the *C++ Core Guidelines*<sup>[49]</sup> were followed throughout development. These core guidelines are a collaborative effort led by Bjarne Stroustrup, creator and developer of the C++ programming language, and came into existence as a result of many years of discussion and design across a number of organizations. Their design encourages general applicability and broad adoption.

# Chapter 7

## Implementation Product

This chapter details a proof of concept for the matter discussed throughout the first four chapters. We will discuss a complete C++ implementation of an ICCM (Section 4.1) and EWMH (Section 4.2) compliant top-level reparenting (Section 2.2.5), tiling (Section 2.2.3) window manager (Section 2.2.1), built on top of X11 (Chapter 3), using Xlib (Section 3.5) as client-side programming library. We will do so by means of demonstration: for every feature it boasts, a brief explanation is provided, including illustrations where necessary.

The name I had decided to give the window manager is *kranewm*, for no other reason than that cranes move objects around in the same way a user might operate a window manager to move windows around, spelled with a *k* for distinguishing purposes, and ending in *wm* to indicate that it is indeed a window manager, as is tradition. It is released as free and open source software under the BSD 3-Clause license and is available on my GitHub and GitLab pages<sup>†</sup>.

### 7.1 Design

At a high level, *kranewm* does not present anything particularly different from other, contemporary window managers. This is of course by design, as we have sought to bring together and thoroughly understand common window management techniques and features by, in essence, implementing them ourselves. At a lower level, personal preferences and experience has sometimes led to changes in the specifics of a feature. Whenever this is the case, the consequences of the choice and the rationale behind it will be provided.

The design for *kranewm* was inspired by a whole host of window managers in popular use today, including but not limited to those discussed in Appendices A and B. Influences go far beyond those of window managers alone, however, as several key *bindings* (Section 7.5) and other design choices are directly based on *Vim*, the text editor authored by Bram Moolenaar. This influence goes even deeper, as most other window managers already inherit notions from *Vim*. This is mostly due to the editor's modal nature, which, originally, was a necessity in the early days of Unix computing, before the wide adoption of the mouse, as it allowed for complete keyboard driven editing. Similarly, most modern tiling window managers aim to provide the ability to “mouselessly” control windows, as does ours.

*kranewm* is ICCCM and EWMH (Chapter 4) compliant and as such acts and feels like any other modern tiling window manager.

### 7.2 Reparenting

Not all tiling window managers choose to reparent the windows under their reign, as some will simply resort to changing the color of the window's border to help the user distinguish which window currently has input focus. A popular such window manager is *dwm* (Section A.2). Choosing to not reparent windows greatly reduces the window manager's inner complexity, as this cuts down on bookkeeping and circumvents the necessity to deal with several corner cases in handling specific types of applications, such as Java-based applications (Section 6.1). Or, as the author of *katriawm* describes their experience with implementing the reparenting feature<sup>[50]</sup>:

*“Turned out, reparenting is not as easy as you might think. Reparenting adds an additional layer of complexity or maybe even more than one layer.*

*Plus, reparenting does not magically fix all your problems. For example, Java expects to run under a reparenting window manager by default. If it doesn't, then you might only get a grey window. Surely, when you write a reparenting WM, even a simple one, this must be fixed, right? No, it won't be fixed. I ended up with either half of the window being grey or with misplaced menus.”*

Despite these difficulties, reparenting is a feature in many modern window managers, and we have therefore decided to implement it in full. Windows in *kranewm* are embedded in a frame that protrudes two pixels from the top, as can be seen in the figure below.

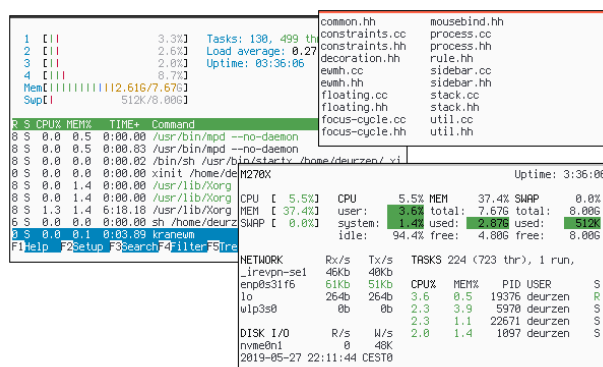


Figure 7.1: Active, inactive, and urgent windows



The window that currently has input focus (Figure 7.1, at the bottom) is given a frame with a light protrusion from the top (by design very subtle, though in practice sufficiently distinguishable), whereas windows that do not will have a dark protrusion (top left). *kranewm* works with all types of *correct* (that is, mostly adhering to and not violating the standards in the ICCCM and EWMH) programs, including Java-based applications.

The frame also acts as an indicator for application urgency. Whenever the *demands attention* bit is set in the window state property (as defined in the EWMH, Section 4.1.2), the user should direct their attention to that window, for one reason or another. To facilitate this, the frame of an application window that sets this bit will be rendered in red (Figure 7.1, top right). Whenever the window that demanded attention is given input focus, or the urgency bit is unset by the application, the frame is colored normally depending on the conditions given above.

7.3 Workspaces

Workspaces (sometimes called *virtual desktops* or just *desktops*, see Section 4.2.1) are essentially groupings of windows that are simultaneously active or inactive. Active windows are shown on screen and can be manipulated by the user, whereas inactive windows are out of sight and therefore out of control. A user is able to switch workspaces (usually by means of a convenient key binding, Section 7.5), which will render the windows on the previous workspace inactive, and those on the target workspace active.

Different window managers implement workspaces in differing ways. *awesome* (Section A.3), for instance, enables the user to have more than a single workspace active at a time. Some window managers, such as *dwm*, do not see workspaces as anything other than groupings of windows, and will not have other information saved amongst them, such as tiling layout or the number of master clients.

*kranewm* does couple some kind of state with each workspace. Each workspace remembers, specific to it, its current and previous tiling layout, the number of master clients, master-to-stack factor, gap size, and whether or not the orientation is mirrored. We will briefly go over each of these variables and the way they are implemented in *kranewm*, in turn.

Layout

Layouts are predefined patterns that define window arrangements and together form a workable ensem-

ble. One and only one layout is enabled at a time, and the user decides which layout to activate. As a result, the user is at all times expected to anticipate which tiling layout best fits their latest needs. For this reason, tiling window managers are considered not beginner friendly, and mostly target power users. To alleviate the user from always having to think in patterns, or to otherwise cater to applications that require specifically structured workflows, most window managers provide a special kind of layout, often called *floating mode*, that can be activated by the user like any other layout. The floating mode layout provides the user with a similar experience to that found in traditional, stacking window managers (as discussed in Section 2.2.2), where windows can be moved around and resized freely.

*kranewm* implements a floating mode layout as well as several tiling layouts (Section 7.10), some of which are based on layouts found in other list-based tiling window managers.

NMaster

Tiling layouts usually divide the entire tileable area up into two partitions, one for master clients and one for stack clients (Section 2.2.3). *kranewm* does so too in its layouts. For the window manager to determine which of the clients on the workspace are to be part of the master area, and which of the stack area, the *nmaster* variable is used. The user can increment or decrement this variable in real time as they see fit, and doing so will automatically rearrange the windows. The figure below illustrates five tiled clients on a workspace in tiled mode, with *nmaster* = 2.

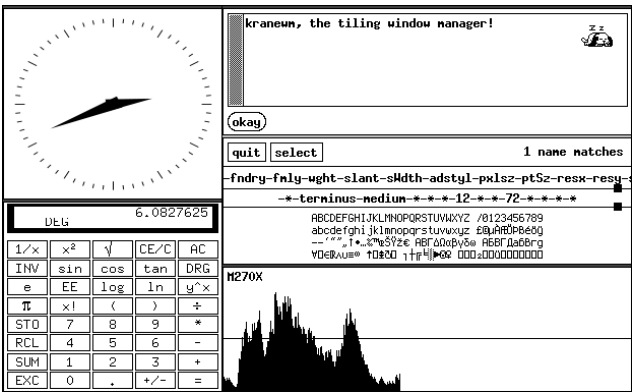


Figure 7.2: Two master clients, three stack

MFactor

The *mfactor* variable determines the size of the master area relative to the entire tileable area. So, with *mfactor* = 0.60, the master partition will have a width of 60% of this area, while the stack partition will take up 40%. *kranewm* allows the user

to dynamically increase or decrease the value of this variable dynamically. Again, doing so will automatically rearrange the windows. The workspace shown in Figure 7.2 has `mfactor = 0.30`.

Window gap

*kranewm* uses this variable (`gap_size`) in tiling procedures to render a gap between the edges of the tileable area and the outer edges of the master and stack areas, and between windows amongst each other. The following figure shows the same workspace as in Figure 7.2, but now with `gap_size = 10` (as opposed to 0).

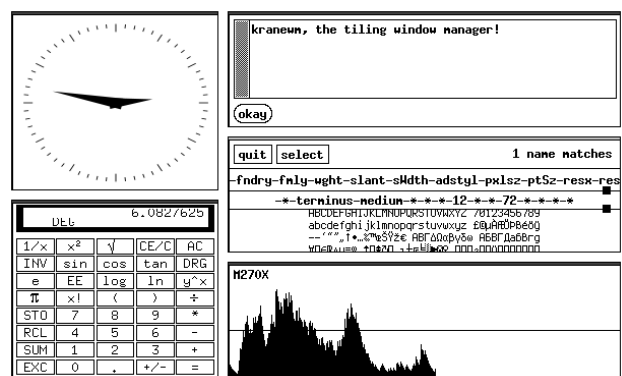


Figure 7.3: Gap size of 10

A *gap* in this regard is nothing more than unused, void space. Rendering empty space around and between windows can effectuate a clearer sense of overview. It helps the user better distinguish between application interfaces, and it may provide for a pleasing aesthetic as well.

Mirroring

Mirroring is a feature that was not (intentionally) imitated from other window managers. It was added to *kranewm* after the need for it arose from extended usage.

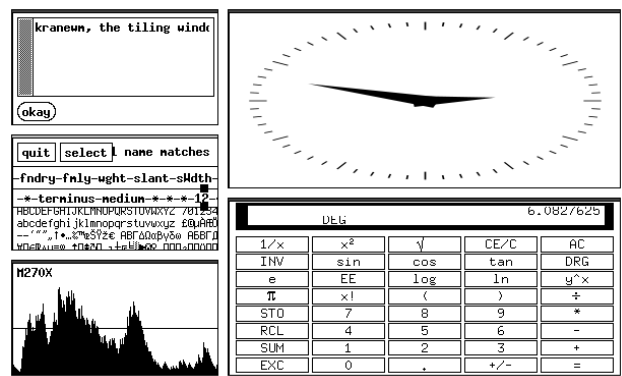


Figure 7.4: Mirrored workspace

The user may toggle mirroring on a workspace, which will swap the position and size of the master and stack partitions, dynamically. If, before toggling, the workspace looks like the one in Figure 7.3, then after it will like the one in Figure 7.4.

Toggling it again will of course reverse the result. Mirroring a workspace will not change any other variables. That is, `nmaster`, `mfactor`, and `gap_size` will remain the same, and so will input focus. Clients in the fullscreen or floating state (Sections 7.8.1 and 7.8.2) will not be affected. Mirroring a workspace in floating, grid, or center mode (Section 7.10) has no visible effect, and mirroring a workspace in pillar mode (Section 7.10) swaps the position and size of the left and right stack panes—the master area is unaffected.

7.4 Sidebar and struts

Where traditionally window managers did not deal with such things as status bars, taskbars, system trays or other such subsidiary information conveying components (Section 4.2.1), as they were each a separate part of the desktop environment—therein properly following the Unix philosophy—modern window managers often do choose to incorporate them. Usually, users will be allowed to disable these components if they so please. Any window manager complying with the EWMH will support external programs that provide similar functionality.

*kranewm* has a built-in sidebar that provides information about the current workspace. As with most other window managers, the user may choose to disable it and opt for an external tool, or decide to not use one at all. The following figure shows the window manager in use, with the built-in sidebar enabled.

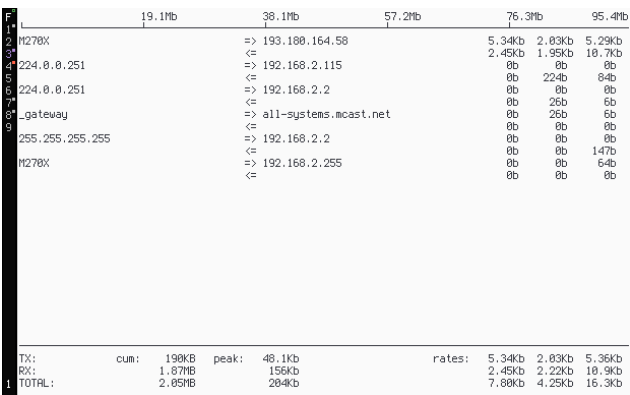


Figure 7.5: Enabled sidebar

The sidebar displays, from top to bottom:

- A *state indicator* at the top right of the sidebar;

- The current workspace's *layout symbol*;
- *Workspace numbers* and corresponding *status indicators*;
- *Workspace client count*.

A state indicator only appears when the currently focused window is in the fullscreen state (colored indicator, green in Figure 7.5) or in the floating state (white indicator). The layout symbol indicates which layout is active on the current workspace: floating ("F", Figure 7.5), tiled ("T"), stick ("S"), deck ("D"), doubledeck ("\$"), grid ("#"), pillar ("P"), column ("C"), monocle ("M"), or center ("^") mode. Each of these tiling layouts is discussed in Section 7.10. The workspace numbers are drawn in order from top to bottom. The currently active workspace's number is colored. A square to the top right of a workspace number, if present, indicates one of three things. A square next to the currently active workspace's number will also be colored and indicates that that workspace has at least one client on it. Inactive workspaces' numbers provide more information. If a window on a different workspace has demanded attention, the indicator next to its workspace's number is colored red (like workspace number four in Figure 7.5). Otherwise, if it contains windows, a white square will be rendered next to it (as with workspace numbers one, seven, and eight in Figure 7.5). If a workspace contains no windows, no status indicator is drawn. The client count at the bottom of the sidebar shows how many clients are on the currently activated workspace if there are less than ten, and the ">" symbol otherwise. The colors used by the sidebar are configurable by the user.

Notice that the window that is in the fullscreen state is nicely aligned with the sidebar, as well as with the other three edges of the screen. It is reasonable for the user to expect that upon disabling the sidebar, or when enabling an external such program (i.e. one that is given an absolute, static position along an edge of the screen), the tileable screen area is to change along with those adjustments. That is, if we were to swap out the built-in sidebar with, say, *lemonbar* [51], and configure it to be positioned at the top of the screen with a height of  $n$  pixels, all windows that are being tiled by the window manager should then together take up the entire screen width, and the screen height minus  $n$ .

*kranewm* is able to detect any changes of this nature dynamically—so, when enabling or disabling a program of this kind, the window manager will adjust its tileable area accordingly in real time, without the need to configure or restart it. It does so by watching for changes in the `_NET_WM_STRUT` and `_NET_WM_STRUT_PARTIAL` properties, specified in the EWMH (Section 4.2.3). Whenever the window

manager detects that a new window has spawned, it reads its strut properties (if present) and consequently does the bookkeeping required to precisely define the tileable screen area. It is capable of reserving space at each edge of the screen. Of course, when a window for which a strut area is reserved is destroyed, the tileable area is readjusted. Figure 7.6 demonstrates a workspace in tiled mode with two instances of *lemonbar* running at the bottom and top strut areas, and the built-in sidebar at the left strut area.

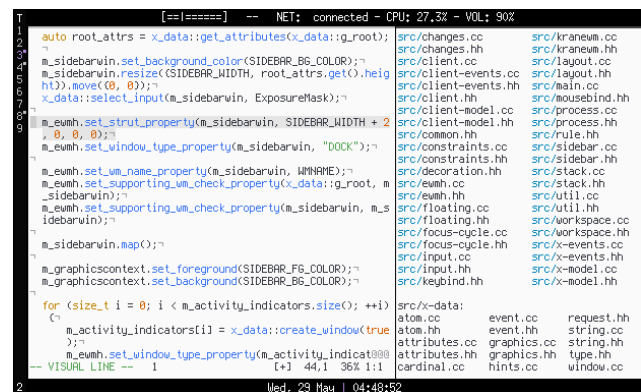


Figure 7.6: Three occupied strut areas

When a window is moved around or resized manually by the user (implying that that window is either in the floating state or floating mode is activated for its workspace), that window is of course allowed across the borders of the tileable area. Windows in the fullscreen state disregard the top and bottom strut areas, and will always take up the entire height of the screen. Reserved space by programs at the left or right strut areas are taken into account.

In the implementation notes section of the EWMH, a stacking order is suggested for the different window types (Sections 4.2.1 and 4.2.3). In particular, it is stated that *dock windows*, which are the windows that reserve a strut area at the edge of the screen, are to always be rendered over regular windows. The window managers enumerated in Appendices A and B, amongst others, do not agree with this suggestion, and choose to give regular windows precedence over dock windows, as far as stacking order is concerned. Regular windows are in active use and convey more important information than dock windows. *kranewm* therefore also prioritizes regular windows over dock windows in the stacking order.

## 7.5 Key bindings

As *kranewm* focuses on keyboard driven window management, most functionality is accessed by the user through *key bindings*. Key bindings are sets of keys that, when pressed together, instruct the win-

dow manager to perform an action.

Throughout the rest of this chapter, when we speak of key bindings, we mean that the functionality behind the key binding is implemented by *kranewm*, and that the user can access that functionality by pressing a unique set of keys.

All key bindings are customizable in *kranewm*. For that reason, we will only mention that an action is coupled with a key binding, and not which specific set of keys activates it.

## 7.6 Mouse bindings

Similar to key bindings, *mouse bindings* couple window manager functionality with the operation of the mouse. A mouse binding is a specification for the mouse button being clicked and an optional modifier key pressed. For example, holding down the alt key while left clicking the mouse on a client initiates window movement.

*kranewm* defines customizable mouse bindings that mostly map to actions over workspaces, such as moving to the next and previous workspace by pressing the forward or backwards buttons on the mouse, respectively.

## 7.7 General usage

Where historically, window manager functionality involved only the manipulation of windows, they are more and more inheriting functionality traditionally left to different parts of the desktop environment. We will briefly go over some of the additions prevalent amongst modern window managers like those listed in Appendices A and B. They are considered general usage features, and they are mostly bound by key bindings.

### 7.7.1 Terminal

The most important program to the power user will probably be the terminal emulator. As such, most modern window managers define a key binding to spawn a terminal window; *kranewm* does so too.

### 7.7.2 Program launching

While other programs could be launched from the terminal, many prefer to not clutter their workflow and would rather use a program launcher. For that reason, window managers often also define a key binding to spawn an external program used to delegate the program launching process. *kranewm* defines such a key binding as well. Its program of choice

is *dmenu*, though a user can always swap it out for a different program if they so please.

### 7.7.3 Closing clients

*kranewm* does not draw buttons on its window frames to provide users with a universal method for closing windows. Instead, closing functionality is coupled with a key binding.

## 7.8 Window states

Like most modern window managers, *kranewm* implements a *floating state* and a *fullscreen state*.

### 7.8.1 Floating state

Some windows are not meant to be tiled, such as pop-up or dialog windows. The window manager automatically detects these, as they should have the `WM_TRANSIENT_FOR` property (Section 4.1.1) set, given they properly comply with the standards in the ICCCM. To distinguish them from other, regular windows that are to be tiled according to the active layout, such windows are put in the *floating state*. The floating state can also be manually toggled on a per window basis by the user. So, if they would rather have the window manager tile a transient window, they can toggle the floating state on that window. Likewise, regular windows that are being tiled along the active layout can be put into the floating state. Tiling will be performed on all windows that are not in the floating state. Consider the following figure. In it, two windows are in the floating state. They are not tiled, and are drawn above the tiled windows (that is, they take precedence in the window stack).

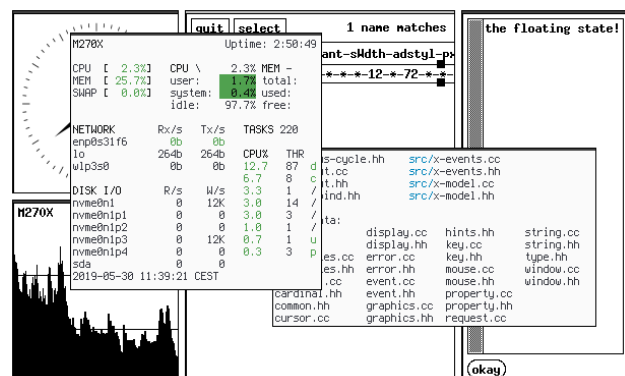


Figure 7.7: Two windows in the floating state

Essentially, all windows in the floating state function as if they were in the floating mode layout. That is, the user can move and resize them freely, keyboard and mouse operations for floating windows



work on them, and their drawing order depends on their position in the window stack, though they will always be rendered atop the tiled windows.

### 7.8.2 Fullscreen state

Fullscreen windows are an integral part of the modern desktop experience. All major desktop environments support them, and most applications rely on the window manager to implement the *fullscreen state*.

Like most other window managers, *kranewm* supports the fullscreen state. Applications can ask the window manager to put them into the state by sending a `ClientMessage` event containing the `_NET_WM_STATE_FULLSCREEN` atom (see Section 4.2.3) to the root window. The user can also toggle the fullscreen state on a window manually through the window manager. Doing so will add the `_NET_WM_STATE_FULLSCREEN` atom to the window's `_NET_WM_STATE` property, notifying the application that it is being treated as a fullscreen window. Figure 7.5 shows a window in the fullscreen state.

## 7.9 Window manipulation

Windows in the environment, called *clients* to the window manager, can be manipulated in several ways. Each window manipulation belongs to at least one of the following three categories.

- Inner-workspace focus movement;
- Inner-workspace window movement;
- Cross-workspace window movement.

Inner-workspace focus movements do not alter the position or size of a window. They merely exist to move input focus from one window to another. Focus is kept track of on a per workspace basis. That is to say, focus is part of the state of a workspace. For example, switching from workspace one to workspace three, and then back to workspace one again will have no effect as far as input focus is concerned. Inner-workspace window movements do affect the position or size of a window, and in the process they may alter the ordering of the clients on the workspace. Cross-workspace window movements handle the transition of a window from one workspace to another. A window can, for example, be moved from workspace two to workspace three. We will briefly discuss the features that belong to each of these three categories.

### 7.9.1 Inner-workspace focus movement

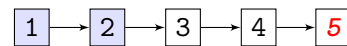
Focus can be directed by using the mouse. Clicking on a client will move input focus to it. Doing so will

also move that client to the top of its part of the window stack, thereby having it rendered above all other such windows on the screen.

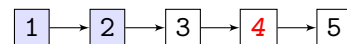
The user may also use the keyboard to transfer focus from one client to another. Key bindings are defined that allow the user to transfer focus forward a client, or backwards a client. This is what is known as *focus cycling*. Alternatively, focus can move to arbitrary clients in the list—this we call *focus jumping*.

#### Focus cycling

The user can cycle focus up or down, relative to the ordered list of clients. For instance, if the clients on a workspace are initially ordered as follows, where the colored nodes represent master clients, and the node with its index colored red is the client that has input focus.



Moving the focus forward will result in the following.

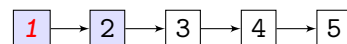


Moving focus backwards has the reversed effect. This process wraps around, so moving focus forward from the first client sets it to the last.

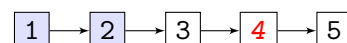
#### Focus jumping

*kranewm* enables the user to move focus directly to the first master client, the first stack client, and the last client in the ordered list with convenient key bindings.

Additionally, *kranewm* implements so-called *pane jumping*. A user can move focus from the master pane to the stack pane, and vice versa. The last set focus for each pane is remembered, such that a pane jump results in focus transference to the last focused client of the other pane. If, starting from our previous example, we were to jump panes, we would get the following.



The first master client is now focused, as no other master client has previously had input focus. Jumping panes again will move focus back to the fourth client in the list, as that was the last focused client from the stack.



Lastly, *kranewm* offers key bindings for directly jumping to any of the first nine clients in the ordered list by index.

## 7.9.2 Inner-workspace window movement

As most window managers, *kranewm* lets the user move and resize clients with the mouse. To make this process more convenient, the user can move a client by holding the alt key while pressing down the left mouse button and moving the mouse; they can resize the client by holding the alt key while pressing down the right mouse button and moving the mouse. Clients are resized from the corner closest to the mouse cursor upon initiating the resize.

*kranewm* defines several key bindings for moving and resizing windows. Depending on a window's state and the layout that is currently active, some of these bindings either directly move or resize the currently focused client, or they rearrange clients in the ordered list, thereby also changing their position and size as the layout is reapplied.

### Centering and snapping

A user can center a floating window manually (that is, the user directs the window manager to center the currently focused window), or a rule (Section 7.11) can be created to automatically have a window centered.

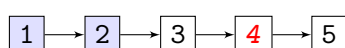
Additionally, floating windows can be manually *snapped* to any of the four edges of the screen. That is, when a user instructs the currently focused window to be snapped to, say, the top edge of the screen, only that window's vertical position will change such that its top border is at the top of the screen.

### Zoom

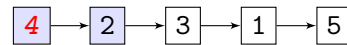
*Zooming* is a concept inherited from other window managers—*dwm* is an example of a window manager that implements zoom functionality.

Zooming a window rearranges the ordered list of clients such that the focused client gets moved to the front of the list. If `nmaster > 0`, zooming will result in the currently focused client becoming the first master client.

If the currently focused client is the first client in the ordered list, zooming will result in a reversal of the previous zoom, if the previously unzoomed master client, if any, is still present on the current workspace. That is, if we begin with the clients ordered as follows.



Zooming will result in the following ordering.

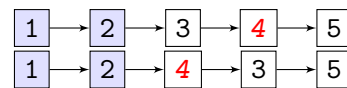


Zooming again will reverse the result. If, however, after the first zoom, client number one were to be removed from the workspace, zooming a second time with the focus on client number four would have no effect.

### Bubble

*Bubbling* a window up or down involves the single-step movement of a window forward or backwards in the ordered list of clients.

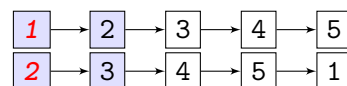
In *kranewm*, key bindings allow the user to bubble up or down the currently focused window. Bubbling down manipulates the ordered list as follows.



Bubbling up from this configuration results in the reversed effect. Focus follows the client being bubbled up or down.

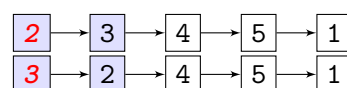
### Rotation

In all layouts except for floating mode, client rotation results in the single-step rotating of the ordered list of clients. A forward rotation of the entire client list:

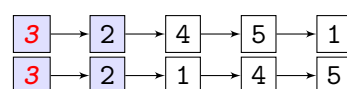


Again, rotating backwards will reverse the result. Notice that focus moves in the opposite direction of the rotation. That is, focus remains at the same index in the ordered list.

Besides full client list rotation, rotation functionality also exists on the master clients and stack clients separately. For example, starting from our previous configuration, rotating the master clients forward:



And subsequently rotating the stack clients backwards:



Rotating stack clients while focus is on a master client has no effect on focus, and neither does rotating master clients when a client in the stack has focus.

*kranewm* defines convenient key bindings for rotating the entire client list, the master clients, and the stack clients forward and backwards.

### Incremental move and resize

A floating client is a client that is in the floating state, or one that is on a workspace with the floating mode layout active. Floating clients can be incrementally moved or resized by using the keyboard. This allows the user to arrange clients on the screen themselves, without the need to use the mouse.

Key bindings are defined that allow for incremental growing and shrinking of a client at all four of its edges, and for movement up, down, left, and right.

### 7.9.3 Cross-workspace window movement

When working with groupings of windows, it is reasonable for a user to expect that it is possible to move windows from one grouping to another. All of the window managers discussed in Appendices A and B that implement workspaces (or a similar concept) have the functionality to move clients across them, and so does *kranewm*.

Key bindings are defined for moving the currently focused window to one of the workspaces by index, and for moving it up or down one workspace.

## 7.10 Layouts

As *kranewm* is a list-based tiling window manager, it implements layouts along which windows on a workspace are arranged. The user can switch between layouts with the activation of key bindings. When changing a layout, the windows on the active workspace are automatically rearranged. We will briefly go over each of the offered layouts here.

### 7.10.1 Floating mode

Floating mode can be seen as an implementation of the stacking window management paradigm (Section 2.2.2). Windows can be moved around and resized freely with both the mouse and they keyboard, and the stacking order (which windows appear above which others) is determined by the chronological order of input focus—windows focused earlier will be rendered below those focused later.

The arrangement of windows in floating mode is saved, such that changing from floating mode to a

tiling layout and back again will preserve the placement and size of the windows.

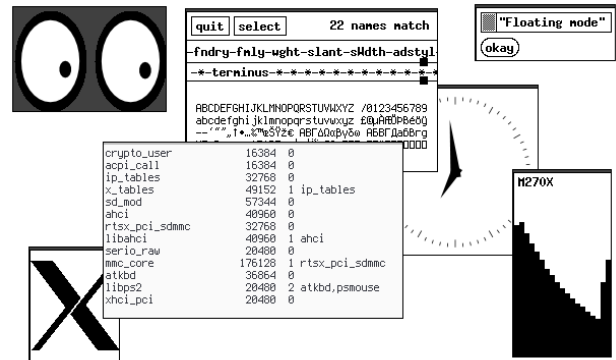


Figure 7.8: Workspace in floating mode

Figure 7.8 shows a workspace in floating mode.

### 7.10.2 Tile mode

Besides floating mode, *kranewm* allows the user to dynamically activate several tiling layouts.

Like most other list-based tiling window managers, *kranewm* implements *n-master* mode (Section 2.2.3), which we call *tile* mode (activated in Figure 7.9). It lays out the workspace's clients in two *panes* (columns): a master pane (left), and a stack pane (right).

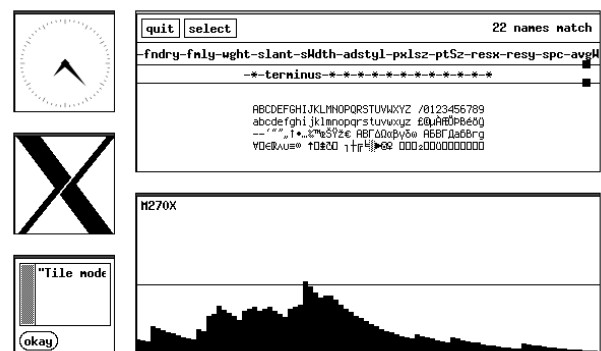


Figure 7.9: Workspace in tile mode

The amount of clients in the master pane is determined by the `nmaster` variable (Section 7.3), which can be set by the user during operation. In fact, all variables set for the workspace, except for the layout variable, of course, are applied, no matter the layout. That is, switching from tile mode to, say, deck mode would have no effect on the variables that determine the number of master clients, the gap size, master pane width, or mirror status.

### 7.10.3 Stick mode

Almost identical to the tile mode layout is *stick* mode, a layout not (intentionally) inherited from



other window managers.

Stick mode arranges windows in the exact same way as tile mode does. The only difference lies in the layouts' handling of gap size. Whereas tile mode renders space between windows amongst each other and between the panes and the screen edges equally, stick mode only does so between the outer borders of each pane and the screen.

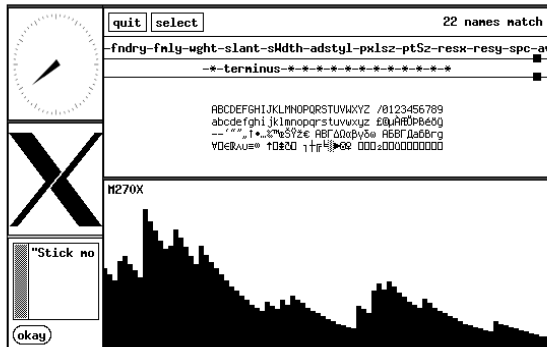


Figure 7.10: Workspace in stick mode

Figure 7.10 shows a workspace with stick mode activated. The workspaces in Figures 7.9 and 7.10 have the same gap size value.

The rationale behind the implementation of stick mode is that arranging windows with a gap between them may be less effective on large monitors, due to the fact that the windows will be pushed away towards the edges of the screen. Stick modes helps keep windows neatly organized near the center of the screen, regardless of its size.

#### 7.10.4 Deck modes

The *deck* mode and *doubleddeck* mode layouts arrange windows in panes, stacked behind each other, like a deck of cards. Whereas deck mode only stacks windows of the stack, doubleddeck mode stacks both the master and stack panes. The following figures showcase workspaces with these modes activated.

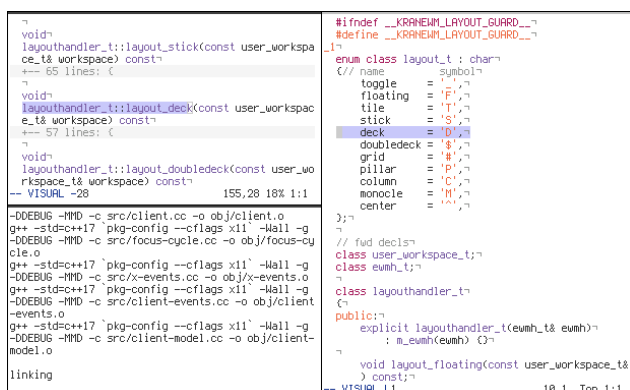


Figure 7.11: Workspace in deck mode

The workspace in Figure 7.11 contains five windows; two master windows (left), and three on the stack, of which only the last focused is visible.

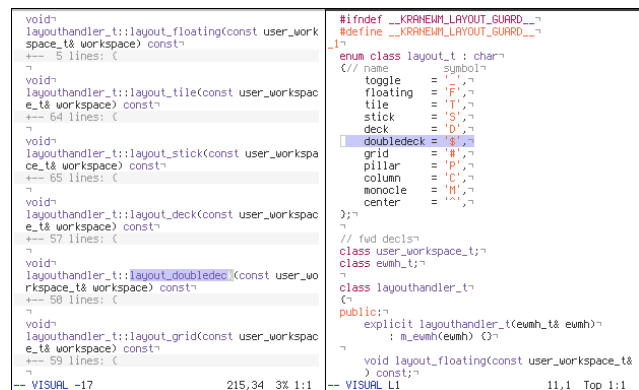


Figure 7.12: Workspace in doubleddeck mode

Figure 7.12 shows a workspace in doubleddeck mode. It contains the same windows as the one in Figure 7.11, though now only the last focused master window from the master pane is visible.

Inner-workspace focus and window movements determine which clients are visible on workspaces with one of these two layouts active. Generally, the last focused window per pane will be visible. Rotating a grouping that is not active will cause the next client from that grouping to become visible. For example, if one of the master clients currently has focus, and the user rotates the stack forward, the next stack client will become visible.

Layouts with windows stacked behind one another can prove useful when working on small screens or with programs with minimal application interfaces. Cycling through the windows in either of the panes can be done entirely through use of the keyboard. This makes it possible to quickly and intuitively access each program without the need to grab for the mouse.

#### 7.10.5 Grid mode

The only layout that ignores all workspace variables is *grid* mode (Figure 7.13). To it, there is no such thing as a master or stack pane, and it does not render a gap between or around windows. Grid mode was not (intentionally) inherited from other window managers.

Grid mode is a somewhat special layout in that it caters to situations in which the user is working with a (large) number of programs with minimal application interfaces. Whenever possible, each window will be given the same amount of space on screen, as windows are arranged in a grid pattern.

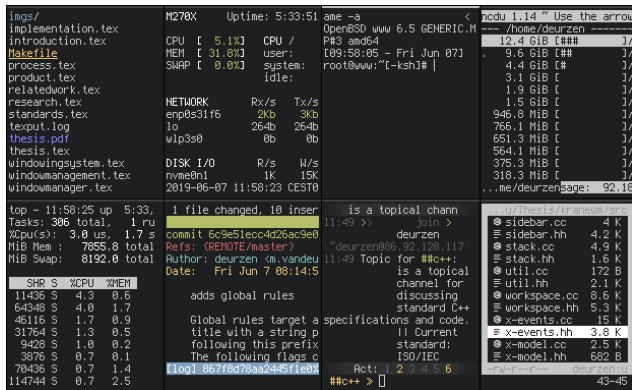


Figure 7.13: Workspace in grid mode

Workflows that could benefit from this layout may for instance arise when working with multiple terminal emulator windows, or when there is otherwise the need for a high level overview of the windows and their contents.

### 7.10.6 Pillar mode

The *pillar* mode layout, also independently added to *kranewm*, arranges the clients on a workspace into three columns, or *pillars*. The center pillar stacks the master clients on top of each other, while the left and right pillars contain the windows from the stack, equally divided amongst the two, also stacked on top of each other.

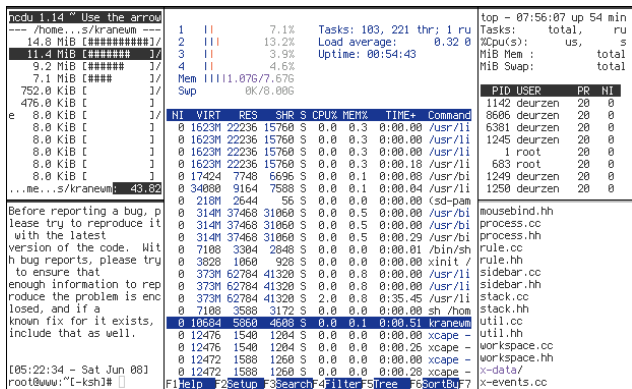


Figure 7.14: Workspace in pillar mode

This layout responds to the various workspace variables. Like the other layouts that do so, the *mfactor* variable determines the size of the master pane relative to the size of the screen. So, if *mfactor* = 0.60, the center pillar will take up 60% of the screen, and the left and right pillars will each take up 20%. Mirroring a workspace with this layout activated will swap the positions and sizes of the left and right pillars.

The pillar mode layout may for example be useful when working with one or several programs with an involved application interface (center pillar), and

multiple less demanding programs (left and right pillars).

### 7.10.7 Column mode

Another layout not (intentionally) inherited by *kranewm* from other window managers is the *column* mode layout.

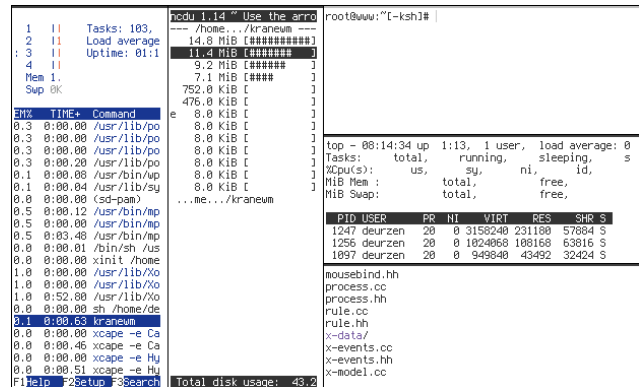


Figure 7.15: Workspace in column mode

Like tile mode, this layout divides the tileable area up into two panes, one for master clients and the other for the stack. The only difference with tile mode is that the master clients are tiled next to each other, instead of on top of each other. All workspace variables work the same on this layout as on tile mode.

Column mode may be suitable when working with programs that have (vertically) extensive application interfaces.

### 7.10.8 Monocle mode

*Monocle* mode is a widely implemented layout amongst modern list-based tiling window managers. It essentially has all clients take up the entire tileable area, stacked behind each other, like one large deck (Section 7.10.4).

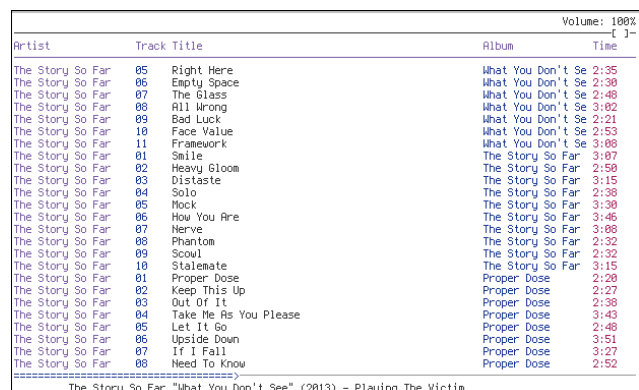


Figure 7.16: Workspace in monocle mode

This layout only responds to gap size. Gaps are

added between the edges of the clients and the edges of the screen.

Monocle mode is similar to having all clients on the workspace in fullscreen mode, though they are not actually to the clients as the `_NET_WM_STATE_FULLSCREEN` atom is not set in the `_NET_WM_STATE` property (Section 4.2).

Monocle mode may useful when working on a small screen, or with programs with involved application interfaces.

7.10.9 Center mode

The *center* mode layout was added independently to *kranewm*. It was added with the intention to serve as a distraction-free layout for programming or for reading or writing prose, without the need to manually arrange windows in such a way using the floating state or floating mode.

The layout is similar to monocle mode, in that all windows are stacked behind each other. The difference is that in center mode, a workspace’s clients have a gap rendered between their left and right edges and the edges of the screen.

```
user_workspace_t& set_gap_size(unsigned);
user_workspace_t& set_factor(float);
user_workspace_t& set_layout(layout_t);

unsigned get_master() const;
unsigned get_gap_size() const;
float get_factor() const;
layout_t get_layout() const;
const workspace_stack_t& get_stack() const;

bool is_master(client_ptr_t);
bool is_stack(client_ptr_t);

bool master_focused();
bool stack_focused();

private:
    unsigned m_number;
    std::string m_name;
    unsigned m_master;
    unsigned m_gap_size;
    float m_factor;
    bool m_mirrored;
    focus_cycle_t m_clients;
    workspace_stack_t m_stack;
    layout_t m_layout;
    layout_t m_previous_layout;
    layouthandler_t m_layouthandler;

    user_workspace_ptr_t;
    <5285C [w] 185,1 92% 8:4
```

Figure 7.17: Workspace in center mode

As with other layouts that respond to gap size, increasing it decreases the size of the clients. Doing so in this layout keeps the proportions of the clients roughly the same.

7.11 Rules

A *rule* is a concept inherited from other window managers, some of which are listed in Appendices A and B (*dwm*, for instance). Rules are mostly used as a last resort, to fine-tune the integration of a window in the environment, or to correct some faulty program behavior.

In its most basic form, a rule is a mapping from a window’s class (Section 4.1.1) to a set of parameters that alter the (initial) operating environment of that window. Most window managers that imple-

ment both rules and workspaces (or a similar concept) will allow the user to specify which workspace a program’s windows should start on. If some kind of floating state is supported, then a user will often also be able to specify whether a program’s windows should start in the floating state, or if they should be arranged normally according to the active layout.

Because an application’s class is set on all of its windows, and may therefore be too general to target with a rule, further distinction can be made by allowing the user to specify an instance name (Section 4.1.1) or title (Section 4.2.3) that must match with a window. An instance name is a class name that is set on a specific instance or window of a program. The window belonging to the *Hangouts* extension for the Chrome browser, for example, will have the same class as the main browser window, but a different instance name. In this case, the `WM_CLASS` property will be a list of two strings. Most X programs have an option to customize a window’s instance name, and often the title as well. The terminal emulator *xrvt-unicode*, for example, can be started with the `-name NAME` flag. Doing so will add *NAME* to the `WM_CLASS` property of the resulting window.

*kranewm* supports rules. Along with the previously mentioned rule-related functionality available in other window managers (initial workspace and floating state), *kranewm* offers the ability to perform *centering*, *autoclosing*, and *workspace hint blocking* on targeted windows.

In addition to the rules that target specific applications by class, *kranewm* implements *global rules* that target any application that has a specific predefined instance name or title set. A window with the instance name `kranewm:fw7c`, for example, will be put into the floating state on workspace 7, and centered upon entering the environment. Any characters after the `kranewm:` prefix represent flags that relate to rule behavior. The following flags are supported.

flag	description
f	start the window in the floating state
c	start the window centered
F	start the window in the fullscreen state
a	autoclose the window
n	do not accept workspace hints for the window
wX	set workspace hint for the window, where X is in the range [0-9]

The order of flags does not matter. With these rules, applications (and the users controlling them) can themselves instigate rule application, without

the need to configure the window manager. Global rules are a feature added to *kranewm* independent of other window manager implementations.

For each of the rule-related features unique to *kranewm*, brief argumentation is provided.

### 7.11.1 Centering

It may sometimes occur that an application completely disregards the X environment, and spawns its windows at the origin of the root window. That is, whenever the application is run, its window appears at the top-left corner of the screen. This can be an inconvenience if not an annoyance to the user, and for that reason the *center* parameter was added.

One might ask, why not automatically center all windows that wish to spawn at the root window's origin? Some programs, such as *Gimp*, may create multiple windows at once. The program's combined application interface might consist of three windows, one of which represents a left pane, one a center pane, and one a right pane. The program might intend to spawn the left pane at the top-left corner of the screen, and the right pane at the other end of the screen, while neatly positioning the center pane in the middle. The window manager is unable to predict this intended behavior, and can therefore never make assumptions about a window's requested position. A user who has experience with a program, can, and to this end the *center* parameter can be used.

### 7.11.2 Autoclosing

The *autoclose* parameter, when set, has two modes: *once* and *persistent*. Any window matching the rule will be closed immediately upon spawning, practically blocking it from the environment. Persistent autoclose rules close the targeted window any time it is encountered. Rules set to run *once* will only close the window the first time it enters the environment; subsequent encounters will not be closed.

Persistent autoclose is useful to factor out windows created by a program that may be considered a hindrance or otherwise useless to the user. An example of such a window is a browser pop-up, that can be targeted by its title, or a dialog box being created by an application to advertise a paid version of the software. Ideally, the hindrance is avoided by configuring the software itself. If nothing else seems viable, autoclose functionality can be used.

Use cases for closing an application or window only upon first encountering it are relatively niche. The need for it may arise when working with programs that minimize to the system tray, and therein behave inconsistently or cannot be properly config-

ured. Windows that close to the system tray do so when receiving the `WM_DELETE_WINDOW` atom in the `WM_PROTOCOLS` property (specified in the ICCCM, Section 4.1.1), where normally this should (if properly implementing the ICCCM) lead to the window being removed entirely from the environment. Many programs that have system tray support will offer the user the option to minimize to the system tray on startup. A user can set up an autoclose rule to do so even if such an option is not available to them.

### 7.11.3 Workspace hint blocking




The *nohint* parameter can be set in a rule to notify the window manager that any matching windows should not be listened to when requesting a workspace to start on. Applications can use the `_NET_WM_DESKTOP` (defined in the EWMH, Section 4.2.3) property to signal to the window manager that it wishes to start on a specific desktop (by index). Usually, a window manager should not make assumptions about the program making such a request, and should obey.

The need for workspace hint blocking functionality may arise when, for instance, programs that minimize to the system tray are gratuitous in workspace signalling. *Artha* is an example of such a program. The user may expect to be able to *untray* the program from any workspace they are currently on. *Artha* will, however, remember the workspace it originally started on, and persistently request to be started on that same workspace after each subsequent spawn. To prevent this from happening, the user would use workspace hint blocking.

## 7.12 Process jumping

*Process jumping* is somewhat similar to rule specification (Section 7.11), in that both features target windows by class. It was added to *kranewm* independent of other window manager implementations. Whereas rules manipulate window behavior within the environment, process jumping allows the user to move focus to any window that matches the class specified. A *process bind*, which defines a key binding while registering for processes with the class specified to be traced by the window manager, has the following form.

keysym	mask	class
XK_b	Mod1Mask ShiftMask	→ Firefox

This process bind would register the key binding   . When the user activates the key binding by pressing the corresponding keys, the last

focused client with the class specified in the process bind will be focused. If no such client exists, nothing will happen. Process jumping works across workspaces. That is, if workspace four is currently active, and the last focused instance of Firefox is on workspace one, the window manager switches to workspace one and moves focus to the targeted window.

When attempting to process jump from a window with the targeted class name (that is, the window being jumped from is the last focused window with the targeted class name), focus will instead be directed back to the client that was last jumped from.

## 7.13 Marking

*Client marking* is a concept inherited from Vim. The text editor allows marks to be set at arbitrary positions in a file, such that it can be easily returned to later. Similarly, marking a client in *kranewm* allows directing focus back to it by use of a key binding.

Jumping to a marked client works across workspaces. Like process jumping, when the currently focused client is the marked client, activating the mark jumping key binding will move focus back to the client that was previously jumped from. Say, for instance, we mark a client on workspace one, activate workspace four and subsequently jump to the marked client, workspace one will be activated and focus will be transferred to the corresponding client. If from this client, we were to again activate the mark jumping key binding, workspace four would be activated and focus would be directed to the client last jumped from. If the client that was last jumped from no longer exists, nothing will happen after attempting to jump back from the marked client.

## 7.14 Summary

*kranewm* is a list-based tiling window manager. That is, it internally represents the windows it manages as elements of an ordered list. Depending on the *layout* applied and its order within this list, a window will be given a fixed position and size on screen. Layouts are predefined arrangement schemes, and the user can switch between them dynamically, as they see fit. *kranewm* offers ten layouts, including *floating mode*, which is essentially an implementation of the stacking window management paradigm. Furthermore, there exist two *states* that are set on a per-window basis. The floating state has a window operate as if it were in floating mode, while the rest of the windows remain tiled (if applicable) behind it. The fullscreen state has the window take up the entire tileable area of the screen.

The windows being managed by *kranewm* are distributed amongst window groupings called *workspaces*. Each workspace has a state associated with it, meaning several variables are kept track of. The currently activated and previously activated layouts are stored, such that moving from one workspace to another and back again does not change the particular arrangement of the windows on a workspace, and such that the toggling between two layouts is made possible. Each workspace independently remembers the gap size, or the spacing between windows and the edge of the tileable area, and between windows amongst each other. It is also stored whether or not a workspace has been *mirrored* on a per-workspace basis. Of course, the number of master clients and the size of the master area are also part of the state.

Additionally, *kranewm* supports three categories of window management actions pertaining to window manipulation. Inner-workspace focus movements allow the user to transfer *input focus* from one window to another. Inner-workspace window movements allow windows to be rearranged in the ordered list, or otherwise for a window's size and position to be changed manually by the user. Cross-workspace window movements allow windows to move across workspaces.

*kranewm* implements both the ICCCM and EWMH almost fully. Only parts of these standards deemed irrelevant, such as pager support, have been left out.

To allow the user to have exact control over an application's windows, *kranewm* offers so-called *rules*. Rules target a window by class, instance name, or title, and automatically perform an action on a window once it has been matched. A rule may have a window spawn in floating mode, start on a specific workspace by index, centered upon startup, (persistently or only initially) autoclosed, or hint blocked.

*Process binds* are similar to rules, in that they too target a window by class, instance name, or title. A process jump moves input focus to the last focused window matching the associated process bind, possibly across workspaces.

Lastly, *kranewm* implements window *marking*. Marking a window allows the user to later move input focus back to it without having to manually look for it across workspaces.

As *kranewm* is keyboard-centric, all of the above-mentioned features are accessible through use of convenient key bindings.



# Chapter 8

## Related Work

There exist numerous open source implementations of top-level window managers. While, to date, most of them are built on top of the X Window System, more and more are being developed for the more modern alternative, Wayland. The window managers showcased in Appendices A and B are examples of X window managers. *Sway*<sup>[52]</sup>, *Way Cooler*<sup>[53]</sup>, and *Weston*<sup>[33]</sup> are all Wayland window managers (compositors).

*TinyWM*, a self-proclaimed *exercise in minimalism*, is a small, minimal window manager implementation that demonstrates the basics of creating an X window manager<sup>[54]</sup>. It only comprises around fifty lines of C code, and lets the user move windows, resize them, raise them (pertaining to stacking order), and move focus between them by moving the mouse (which X supports by itself).

A window manager that has expanded upon the basics implemented by *TinyWM* is *SmallWM*<sup>[55]</sup>. It additionally supports iconification, layering, click-to-focus, virtual desktops, and window-to-edge snapping.

Furthermore, an informal yet interesting account of the development process of an X window manager, *katriawm*, has been written by Peter Hofmann<sup>[50]</sup>. The window manager's code is also freely available.

The three aforementioned window managers do not implement the ICCCM and EWMH standard protocols, whereas our window manager does. Most window managers discussed in Appendices A and B do implement them as well.

In addition, several catalogs exist that list window managers and their features. These catalogs will often include links to pages that provide more detailed usage and installation instructions. The *Arch Wiki* provides such a catalog<sup>[56,57,58]</sup>, and so does Wikipedia<sup>[59]</sup>. More extensive surveys also exist, such as WikiBooks' *Guide to X11/Window Managers*<sup>[60]</sup>. Another such detailed survey is Giles Orr's *Comprehensive List of Window Managers for Unix*; in it, he includes accounts of his own experience with each of the listed window managers<sup>[61]</sup>.

A study similar to the one presented in this thesis has recently (2018) been done by Evan Bradley towards his graduation from The Honors College at Oakland University<sup>[62]</sup>. It describes a zoomable user interface window manager for the X Window System that aims to provide mechanisms for easily managing a large number of windows. Additionally, a brief discussion on other research interfaces is included alongside a more extensive survey on X win-

dow managers, which, by his own account, provide a substantial source for contemporary window management research. While his focus is also on the X Window System and the various window management techniques and technologies that have come into existence as a result of it, it differs significantly from this thesis in the development process and goal of the product being presented.

# Chapter 9

## Conclusions

This thesis has discussed the functioning of contemporary top-level window managers. We have shown what common features window managers may consist of, and which roles they assume in various system-level interactions. In particular, we have argued about the window manager's rights and responsibilities, and we have shown the different types of window managers that are in popular use. It was found that commercial system software, like Windows and macOS, mostly comprise a single largely non-replaceable desktop environment, in which stacking window management techniques are employed. Systems in which the window manager is a more distinct and replaceable component, like those in the open source ecosystem, see its users reaching for different types of window managers. Next to stacking window managers, we speak of tiling, compositing, and reparenting window managers. Tiling window managers are popular amongst power users—often developers and hobbyists—as they have proven to be more efficient than their stacking counterparts, though they are considered harder to learn. We recognize two often applied tiling window management techniques: list-based and tree-based tiling. The former has the user dynamically activate so-called tiling layouts, or predefined arrangement schemes, that automatically tile the windows in use. The latter also has the windows tiled automatically, though the user is offered more control in fine-tuning their position and size. Compositing window managers alter the look and feel of the graphical user interface by applying effects and animations, while reparenting window managers add a frame around application interfaces to, for instance, display an application window's title, or to add buttons.

We have furthermore discussed the X Window System, a network-transparent windowing system that runs on a wide variety of machines. By implementing standard protocols that define policy on top of the X Window System, applications are given a means by which they can communicate. Importantly, communication between the window manager and other, regular applications is facilitated. Two of the most widely implemented such standard protocols, the Inter-Client Communication Conventions Manual (ICCCM) and the Extended Window Manager Hints (EWMH), were also discussed. In the future, window manager developers and users will likely want to move to X's more modern alternative, Wayland. Notwithstanding, we argue X will remain

relevant, if only for compatibility reasons.

As a proof of concept, we have presented and discussed a complete C++ implementation of an ICCCM and EWMH compliant top-level reparenting, tiling window manager, built on top of the X Window System, using Xlib as client-side programming library. In doing so, the process gone through to realize this implementation was documented, as various Agile software development concepts were applied throughout. Specifically, we have assessed the Personal Extreme Programming software development methodology, a mix between regular Extreme Programming and the Personal Software Process designed to guide a lone developer through the development process. We have found no particular issues in its approach, as we were able to efficiently work on the product one feature at a time, to process problems effectively, and ultimately to finish the product in a timely and successful manner.



# Bibliography

- [1] Scheifler, R. W. and Gettys, J., (1990). *The X Window System*. Software: Practice and Experiment.
- [2] Rosenthal, D. and Marks, S. W., (1994). *Inter-Client Communication Conventions Manual*. Retrieved from <https://www.x.org/releases/X11R7.6/doc/xorg-docs/specs/ICCCM/icccm.html>.
- [3] X Desktop Group, (2005). *Extended Window Manager Hints*. Retrieved from <https://specifications.freedesktop.org/wm-spec/wm-spec-1.3.html>.
- [4] Jeuris, S., Tell, P., Houben, S., and Bardram, J. E., (2018). *The Hidden Cost of Window Management*. CoRR. Retrieved from <http://arxiv.org/abs/1810.04673>.
- [5] Myers, B. A., (1995). *User Interface Software Tools*. ACM Transactions on Computer-Human Interaction, 2(1):64–103.
- [6] Myers, B. A., (2003). *Graphical User Interface Programming*.
- [7] Crow, D. and Jansen, B. J., (1998). *Seminal works in computer human interaction*.
- [8] Jansen, B. J., (1998). *The Graphical User Interface: An Introduction*.
- [9] Harding, B., (1989). *Windows and icons and mice, oh my! The changing face of computing*. In *Proceedings 1989 Frontiers in Education Conference*, pages 337–342.
- [10] Nye, A., (1994). *X Protocol Reference Manual (Fourth Edition)*, volume 0 of *The Definitive Guides to the X Window System*. O'Reilly & Associates.
- [11] Nye, A., (1994). *XLIB Programming Manual (Fifth Release, Third Edition)*, volume 1 of *The Definitive Guides to the X Window System*. O'Reilly & Associates.
- [12] Nye, A., (1994). *XLIB Reference Manual (Fifth Release)*, volume 2 of *The Definitive Guides to the X Window System*. O'Reilly & Associates.
- [13] Wiki, (2008). *Policy And Mechanism*. Retrieved from <http://wiki.c2.com/?PolicyAndMechanism>.
- [14] Ji, C., (2014). *How X Window Managers Work, And How To Write One*. Retrieved from <https://jichu4n.com/posts/how-x-window-managers-work-and-how-to-write-one-part-i/>.
- [15] University of Waterloo. *Painter's Algorithm*. Retrieved from <http://medialab.di.unipi.it/web/IUM/Waterloo/node67.html>.
- [16] Waldner, M., Steinberger, M., Grasset, R., and Schmalstieg, D., (2011). *Importance-driven Compositing Window Management*. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 959–968. ACM.
- [17] Hutchings, D. R., Smith, G., Meyers, B., Czerwinski, M., and Robertson, G., (2004). *Display Space Usage and Window Management Operation Comparisons Between Single Monitor and Multiple Monitor Users*. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '04, pages 32–39. ACM.
- [18] Zeidler, C., Lutteroth, C., and Weber, G., (2013). *An Evaluation of Stacking and Tiling Features within the Traditional Desktop Metaphor*.
- [19] Bly, S. A. and Rosenberg, J. K., (1986). *A comparison of tiled and overlapping windows*. In *ACM SIGCHI Bulletin*, volume 17, pages 101–106. ACM.
- [20] Kandogan, E. and Shneiderman, B., (1997). *Elastic Windows: evaluation of multi-window operations*. In *Proceedings of the ACM SIGCHI Conference on Human factors in computing systems*, pages 250–257. ACM.
- [21] Cohen, E. S., Smith, E. T., and Iverson, L. A., (1986). *Constraint-Based Tiled Windows*. In *IEEE Computer Graphics and Applications v6 n5*, pages 35–45.
- [22] Abler, A., (2016). *Tiling Window Managers*. Retrieved from <https://files.project21.ch/LinuxDays-Public/16FS-Expert.pdf>.
- [23] OSDev.org, (2014). *Compositing*. Retrieved from <https://wiki.osdev.org/Compositing>.
- [24] Scheifler, R. W. and Gettys, J., (1990). *The X Window System, Version 11*. Software: Practice and Experiment.

- [25] X.Org Foundation. *X - a portable, network-transparent window system*. Retrieved from <http://man.openbsd.org/X.7>.
- [26] X.Org Foundation. *XOrgFoundation - X.Org Foundation information*. Retrieved from <http://man.openbsd.org/XOrgFoundation.7>.
- [27] Høgsberg, K. *The Wayland Protocol*. Retrieved from <https://wayland.freedesktop.org/docs/html/ch01.html#sect-Motivation>.
- [28] Manrique, D., (2001). *X Window System Architecture Overview*. Retrieved from <https://www.tldp.org/HOWTO/pdf/XWindow-Overview-HOWTO.pdf>.
- [29] Massey, B. and Sharp, J., (2001). *XCB: An X Protocol C Binding*.
- [30] Nye, A. and O'Reilly, T., (1992). *X Toolkit Intrinsic Programming Manual (Third Edition)*, volume 4 of *The Definitive Guides to the X Window System*. O'Reilly & Associates.
- [31] Gettys, J. and Scheifler, R. W., (1985). *Xlib - C Language X Interface*.
- [32] Keith Packard, (1999). *The Xft Font Library: Architecture and Users Guide*.
- [33] Høgsberg, K., (2012). *Documentation 1.3 Wayland*.
- [34] X Consortium. *xterm - terminal emulator for X*. Retrieved from <https://man.openbsd.org/xterm.1>.
- [35] Khan, A. I., Qureshi, M., and Khan, U. A., (2012). *A Comprehensive Study of Commonly Practiced Heavy & Light Weight Software Methodologies*. arXiv.
- [36] Lee, N. and Madej, K. *Disney Stories*. Springer, New York, NY.
- [37] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D., (2001). *Manifesto for Agile Software Development*. Retrieved from <http://www.agilemanifesto.org/>.
- [38] Larman, C. and Basili, V. R., (2003). *Iterative and Incremental Development: A Brief History*. Computer, 36(6):47–56.
- [39] Juric, R., (2000). *Extreme programming and its development practices*. In *Proceedings of the 22nd International Conference on Information Technology Interfaces (Cat. No.00EX411)*, pages 97–104.
- [40] Humphrey, W. *The Personal Software Process (PSP)*. Technical Report CMU/SEI-2000-TR-022, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, (2000). Retrieved from <http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=5283>.
- [41] Dzhurov, Y., Krasteva, I., and Ilieva, S., (2009). *Personal Extreme Programming: An Agile Process for Autonomous Developers*.
- [42] Ambler, S., (2002). *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. John Wiley & Sons, Inc., New York, NY, USA.
- [43] Tomayko, J. E. and Herbsleb, J., (2003). *How Useful Is the Metaphor Component of Agile Methods? A Preliminary Study*.
- [44] Hunt, A. and Thomas, D., (1999). *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- [45] Chacon, S. and Straub, B., (2014). *Pro Git*. Apress, Berkely, CA, USA, 2nd edition.
- [46] Catch Org, (2019). *Catch2: A modern, C++-native, header-only, test framework for unit-tests, TDD and BDD - using C++11, C++14, C++17 and later (or C++03 on the Catch1.x branch)*. Retrieved from <https://github.com/catchorg/Catch2>.
- [47] Allman, E., (2012). *Managing Technical Debt*. ACM, 55(5):50–55.
- [48] Sutter, H. and Alexandrescu, A., (2004). *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ in Depth Series)*. Addison-Wesley Professional.
- [49] Stroustrup, B. et al., (2019). *C++ Core Guidelines: A set of tried-and-true guidelines, rules, and best practices about coding in C++*. Retrieved from <https://github.com/isocpp/CppCoreGuidelines>.
- [50] Hofmann, P., (2016). *katriawm: The adventure of writing your own window manager*. Retrieved from <https://www.uninformativ.de/>

- [blog/postings/2016-01-05/0/POSTING-en.html](#).
- [51] LemonBoy, (2019). *bar: A featherweight, lemon-scented, bar based on xcb*. Retrieved from <https://github.com/LemonBoy/bar>.
- [52] DeVault, D. et al., (2019). *Sway*. Retrieved from <https://swaywm.org/>.
- [53] Carpenter, P. et al., (2019). *Way Cooler*. Retrieved from <http://way-cooler.org/>.
- [54] Welch, N., (2019). *TinyWM*. Retrieved from <http://incise.org/tinywm.html>.
- [55] Marchetti, C., (2019). *SmallWM: Xlib window manager (at one time based on tinywm, but SmallWM has long outgrown it)*. Retrieved from <https://github.com/adamnew123456/SmallWM>.
- [56] Arch Wiki, (2019). *List of window managers*. Retrieved from [https://wiki.archlinux.org/index.php/Window\\_manager#List\\_of\\_window\\_managers](https://wiki.archlinux.org/index.php/Window_manager#List_of_window_managers).
- [57] Arch Wiki, (2019). *List of composite managers*. Retrieved from [https://wiki.archlinux.org/index.php/Xorg#List\\_of\\_composite\\_managers](https://wiki.archlinux.org/index.php/Xorg#List_of_composite_managers).
- [58] Arch Wiki, (2019). *Comparison of tiling window managers*. Retrieved from [https://wiki.archlinux.org/index.php/Comparison\\_of\\_tiling\\_window\\_managers](https://wiki.archlinux.org/index.php/Comparison_of_tiling_window_managers).
- [59] Wikipedia, (2019). *Comparison of X window managers*. Retrieved from [https://en.wikipedia.org/wiki/Comparison\\_of\\_X\\_window\\_managers](https://en.wikipedia.org/wiki/Comparison_of_X_window_managers).
- [60] Wikibooks, (2019). *Guide to X11/Window Managers*. Retrieved from [https://en.wikibooks.org/wiki/Guide\\_to\\_X11/Window\\_Managers](https://en.wikibooks.org/wiki/Guide_to_X11/Window_Managers).
- [61] Orr, G., (2019). *The Comprehensive List of Window Managers for Unix*. Retrieved from <http://gilesorr.com/wm/table.html>.
- [62] Bradley, E., (2018). *An Infinite-Pane, Zooming User Interface Window Manager and Survey of X Window Managers*.
- [63] Maglione, K. et al. *wmii: A small, scriptable window manager, with a 9P filesystem interface and an acme-like layout*. Retrieved from <https://code.google.com/archive/p/wmii/>.
- [64] Garbe, A. R. et al. *dwm: dynamic window manager for X*. Retrieved from <https://dwm.suckless.org/>.
- [65] Danjou, J. et al. *awesome: a highly configurable, next generation framework window manager for X*. Retrieved from <https://awesomewm.org/>.
- [66] Janssen, S., Stewart, D., and Creighton, J. *xmonad: a dynamically tiling X11 window manager*. Retrieved from <https://xmonad.org/>.
- [67] Stapelberg, M. et al. *i3: improved tiling wm*. Retrieved from <https://i3wm.org/>.
- [68] Dejean, B. *bspwm: a tiling window manager based on binary space partitioning*. Retrieved from <https://github.com/baskerville/bspwm>.
- [69] Wißmann, T. *herbstluftwm: a manual tiling window manager for X11 using Xlib and Glib*. Retrieved from <https://herbstluftwm.org/>.

# Appendix A

## List-based Tiling Window Managers

There is a whole host of tiling window managers currently available, many of which are list-based. We will shortly discuss several of the more popular amongst them here. Whenever it is mentioned that a window manager is ICCCM or EWMH compliant (Chapter 4), we mean a subset of these standards is implemented. All of them principally target Linux or other Unix-based platforms, and are open source.

### A.1 wmii

**wmii**<sup>[63]</sup> (for *window manager improved 2*), is a reparenting, non-compositing list-based tiling window manager for X (interfaced with Xlib) that partially complies with the ICCCM and EWMH standards. It supports classic stacking and tiling window management with extended keyboard, mouse, and filesystem based remote control. It replaces the workspace paradigm with a tagging approach, in which every client is assigned one or more tags that it belongs to (Figure A.1, bottom left). **wmii**'s look and functionality is heavily inspired by *acme*, a text editor and graphical shell.

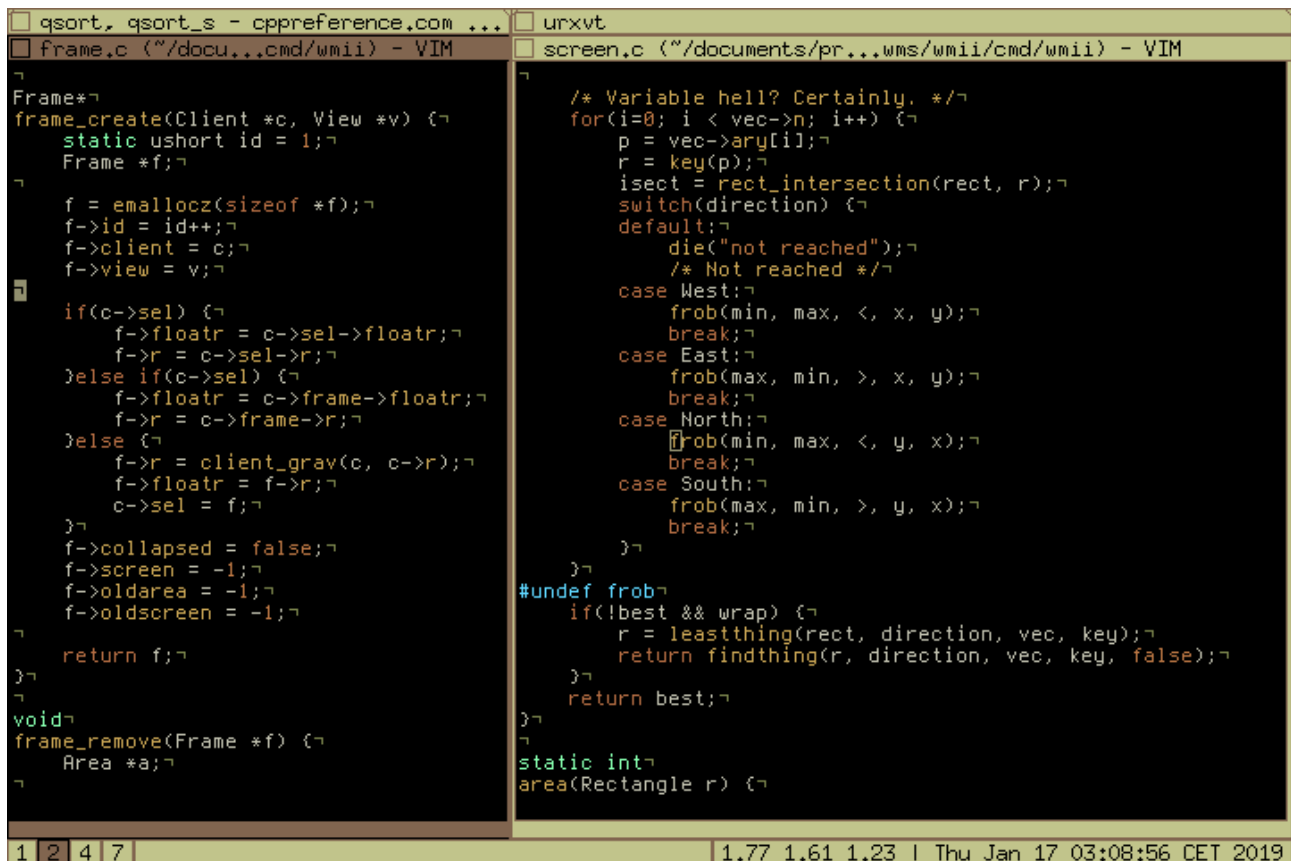


Figure A.1: **wmii** with two columns and four active tags

The default configuration uses key bindings derived from those of the *vi* text editor. Extensive configuration is possible through a virtual filesystem that uses the 9P filesystem protocol, similar to that offered by the *Plan 9 from Bell Labs* operating system. Every window, tag, and column is represented in the virtual filesystem, and windows are controlled by manipulating their file objects.

Configuration is done through a script that interfaces the virtual filesystem. This approach, leveraging remote procedure calls, allows for many different configuration styles, including those in the *plan9port* (which is a port of *Plan 9 from Bell Labs* libraries and applications to Unix-like operating systems) and the *bourne shell* (*bash*).

**wmii** includes a keyboard-based menu program used to spawn external programs, called *wimenu*, that features history and programmable completion. There is built-in multi-monitor support using *Xinerama*.

## A.2 dwm

**dwm**<sup>[64]</sup> (for *dynamic window manager*, not to be confused with Windows' DWM) is a non-reparenting, non-compositing list-based tiling window manager for the X Window System. It uses Xlib as client-side programming library, and acts largely in accordance with ICCCM and EWMH. It manages windows in tiled, stacked, and fullscreen layouts. Layouts can be applied dynamically by the user, such as to optimize the environment for the applications in use and the problems at hand. A floating state can be toggled on a per-window basis, making it possible for some windows to be in a tiled layout, while others are floating in front of them. dwm is lightweight and fast; it is written in C, with a stated design goal of keeping the code base under 2000 source lines of code. It provides multi-head support for XRandR and Xinerama.

```

top - 23:29:28 up 3:08, 1 user, load average: 0.59, 0.68, 0.72
Tasks: 189 total, 1 running, 188 sleeping, 0 stopped, 0 zombie
%Cpu0: 0.0 us, 0.0 ni, 82.2 id, 0.0 wa, 0.4 hi, 0.2 si
MiB Mem: 4578.5 free, 1736.7 used, 1540.6 buff/
MiB Swap: 8192.0 free, 0.0 used, 5632.0 avail

PID    USER    VIRT    RES     SHR    S    %CPU  %MEM    TIME+
24143   root     81132   657136  85780  S    54.8   8.2    28:24.07
20819   root     39600   68708   45444  S    3.3    0.9    1:38.20
2426    root     84748   13752   10208  S    3.0    0.2    2:55.71
20907   root     12280   113348  54000  S    2.6    1.4    1:26.12
375     root      0        0        0      S    0.7    0.0    1:53.84
20902   root     13812   1460    1284   S    0.7    0.0    0:02.59
28214   root     13296   3572    3120   R    0.7    0.0    0:07.24
1       root     24544   8828    6840   S    0.3    0.1    0:07.52
27      root      0        0        0      S    0.3    0.0    0:00.62
20897   deurzen  210644  15736   12680  S    0.3    0.2    0:05.27
20900   deurzen  7240    3800    3244   S    0.3    0.0    0:03.19
24000   deurzen  50052   23168   16672  S    0.3    0.3    0:01.66
24396   root      0        0        0      I    0.3    0.0    0:00.31
25408   deurzen  50024   23324   16784  S    0.3    0.3    0:01.92
2       root      0        0        0      S    0.0    0.0    0:00.00
3       root      0       -20      0      I    0.0    0.0    0:00.00
4       root      0       -20      0      I    0.0    0.0    0:00.00
5       root      0        0        0      I    0.0    0.0    0:00.00
6       root      0       -20      0      I    0.0    0.0    0:00.00
8       root      0       -20      0      I    0.0    0.0    0:00.00
9       root      0        0        0      S    0.0    0.0    0:00.31
10      root     -2        0        0      I    0.0    0.0    0:01.18
11      root     -2        0        0      S    0.0    0.0    0:00.64
12      root     -2        0        0      S    0.0    0.0    0:00.00
13      root      rt        0        0      S    0.0    0.0    0:00.10
14      root    -51        0        0      S    0.0    0.0    0:00.00
16      root     20        0        0      S    0.0    0.0    0:00.00
17      root     20        0        0      S    0.0    0.0    0:00.00
18      root    -51        0        0      S    0.0    0.0    0:00.00
19      root      rt        0        0      S    0.0    0.0    0:00.40
20      root     -2        0        0      S    0.0    0.0    0:00.64

-[master]-- w
23:29:10 up 3:08, 1 user,
load average: 0.57, 0.68, 0.72
USER      TTY      LOGIN@  I
DLE  JCPU  PCPU  WHAT
deurzen  tty1    20:26   55:
53      5:27  0.00s xinit /
[~/documents/projects/wms/dwm]
-[master]-- who
deurzen  tty1    2019-01-
16 20:26
[~/documents/projects/wms/dwm]
-[master]-- ps
PID TTY      TIME CMD
16620 pts/11   00:00:00 zsh
18368 pts/11   00:00:00 ps
[~/documents/projects/wms/dwm]
-[master]-- 
[~/documents/projects/wms/dwm]
-[master]-- ls
config.def.h  dwm.o
config.h      dwm.png
config.mk     LICENSE
drw.c         Makefile
drw.h         README
drw.o         transient.c
dwm*          util.c
dwm.i          util.h
dwm.c         util.o
[~/documents/projects/wms/dwm]
-[master]-- 

```

**Figure A.2:** dwm in stack mode, with a floating xclock window

Configuring dwm is done by editing its source code; it has no configuration interface. This is also the only way to add custom layouts, though there are optional patches that can be applied, made available by the community.

dwm has influenced the development of several other list-based tiling window managers, including awesome (Section A.3) and xmonad (Section A.4).

Similar to wmii's approach (Section A.1), windows are grouped by tags. Each window can be assigned one or multiple tags. Selecting a tag displays all windows with this tag (see Figure A.2, tag numbers in the top left), and multiple tags can be selected at once. By default, dwm draws a single-pixel border around windows to indicate the focus state.

Each screen contains a status bar that displays the available tags, the currently active layout, the title of the focused window, and the text read from the root window name property (Section 3.13 and Chapter 4). In the status bar, above the title of the focused window, an empty square indicates that that window is in the floating state, and a filled square indicates that the window is in fullscreen mode. No square means that the window is tiled along the active layout. The selected tags are indicated with a different color. The tags of the focused window are indicated with a filled square in the top left corner. If one or more windows has been assigned to a tag that is not currently selected, an empty square is displayed to the top left of the tag number.



## A.3 awesome

**awesome**<sup>[65]</sup> is a highly customizable dwm derivative that implements the XDG Base Directory, XEmbed, Desktop Notification, System Tray and EWMH standards. Its core is written in C, and it has a configuration and extension system written in Lua. It is therefore often referred to as a (self-proclaimed) framework window manager. awesome was originally a dwm fork, but was later ported from its Xlib interface to XCB. It has since also become a reparenting window manager, though it still shares many characteristics with dwm, such as list-based tiling and layouts based on the master-client paradigm, the floating state, and tags. It aims to be extremely small and fast, yet extensively customizable. As with dwm, it makes it possible for the user to manage windows with the use of the keyboard.

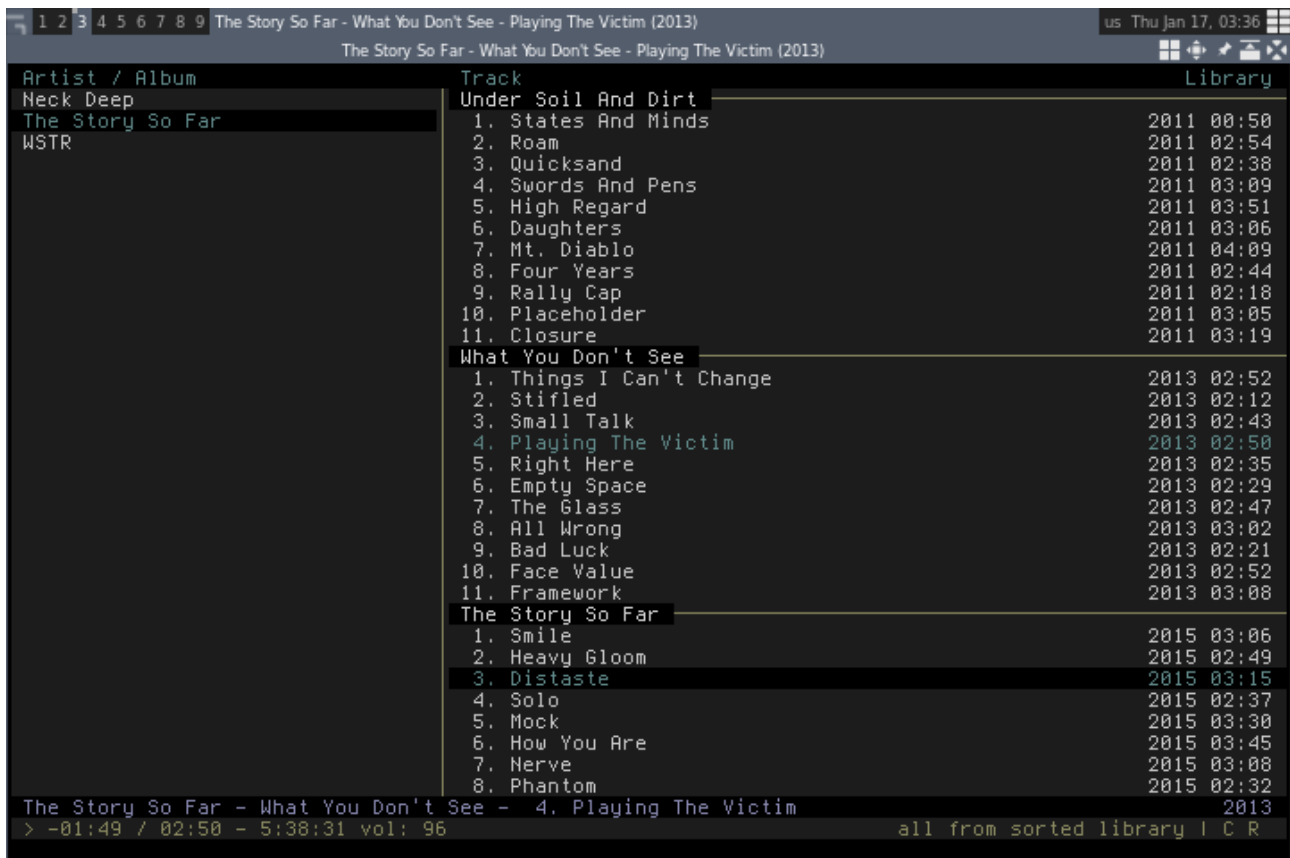


Figure A.3: awesome with a single window in fairv layout

awesome supports inter-process communication through D-bus, text styling in its status bar with pango, and it offers multi-monitor support for XRandR, Xinerama and even Zaphod mode (single-display screen sharing).

As opposed to dwm, awesome enables the user to have different layouts active on different tags. On one tag, the user may have its windows in the floating state, while a tiled layout is active on another.

As can be gleaned from Figure A.3, awesome's aesthetic is quite similar to that of dwm. It has a status bar with its tag set, currently active layout, window titles, and an area for arbitrary text rendering.

## A.4 xmonad

**xmonad**<sup>[66]</sup> is a non-reparenting, non-compositing list-based tiling window manager similar to dwm. It is written in Haskell, and abides by most of the conventions in the ICCCM and EWMH.

Quoting from the haskell xmonad package description: "By utilizing the expressivity of a modern functional language with a rich static type system, xmonad provides a complete, feature-rich window manager in less than 1200 lines of code, with an emphasis on correctness and robustness. Internal properties of the window manager are checked using a combination of static guarantees provided by the type system, and type-based automated testing. A benefit of this is that the code is simple to understand, and easy to



modify.”

As is with most tiling window managers, xmonad allows the user to control windows using solely the keyboard, rendering the mouse strictly optional. A principle of xmonad is predictability: the user should know in advance precisely the window arrangement that will result from any action. Whereas *wmii*, *dwm* and *awesome* employ tags, xmonad has workspaces—which it calls *virtual screens*—that allow the user to organize the windows of currently running programs. Workspaces retain their active layout, number of master clients, and master factor.

```

[1] 2 3 4 5 : Tall : xmonad.hs (~/.xmonad) - VIM
Linux * Thu Jan 17 2019 * 05:32:50

import XMonad
import XMonad.Hooks.DynamicLog
import XMonad.Hooks.ManageDocks
import XMonad.Util.Run(spawnPipe)
import XMonad.Util.EZConfig(additionalKeys)
import System.IO

myManageHook = composeAll
  [ className ==? "Gimp" --> doFloat
  , className ==? "Vncviewer" --> doFloat
  ]

main = do
  xmproc <- spawnPipe "/usr/bin/xmobar -B white -a right -F bl
ue -t '%LIPB%' -c '[Run Weather \"LIPB\" [] 36000]'"

  xmonad $ defaultConfig
    { terminal      = "urxvt"
    , modMask      = mod1Mask
    , borderWidth = 1
    , manageHook   = manageDocks <+> manageHook defaultConfig
    , layoutHook   = avoidStruts $ layoutHook defaultConfig
    , handleEventHook = handleEventHook defaultConfig <+> do
      cksEventHook
    , logHook      = dynamicLogWithPP xmobarPP
      { ppOutput = hPutStrLn xmproc
      , ppTitle  = xmobarColor "green" "" . shorten 50
      }
    , `additionalKeys`
    [ ((mod1Mask .|. shiftMask, xK_z), spawn "7lock")
    , ((controlMask, xK_Print), spawn "scrot -s")
    , ((0, xK_Print), spawn "scrot")
    ]

~
~
~
~

```

```

[~/xmonad]— - xmonad --recompile
XMonad will use ghc to recompile, b
ecause "/home/deurzen/.xmonad/build
" does not exist.
XMonad recompilation process exited
with success!
[~/xmonad]— - ls
xmonad.errors
xmonad.hi
xmonad.hs
xmonad.o
xmonad-x86_64-linux*
[~/xmonad]— -

(18/20) upgrading virtualbox-host-m
(18/20) upgrading virtualbox-host-m
odules-arch
(19/20) upgrading youtube-dl
(19/20) upgrading youtube-dl
(19/20) upgrading youtube-dl
(20/20) upgrading zeromq
(20/20) upgrading zeromq

:: Running post-transaction hooks..
(1/7) Updating linux module depende
ncies...
(2/7) Install DKMS modules
==> dkms install wireguard/0.0.2018
1218 -k 4.20.2-arch1-1-ARCH

```

**Figure A.4:** xmonad with xmobar, and three windows in stack mode

xmonad can be extensively configured with Haskell. As opposed to *dwm*, layouts for Haskell’s tiling algorithm can also be defined in Haskell configuration files. Even so, a recompilation of the binary is still required for any changes to take effect. Multi-monitor support is offered for Xinerama.

*Out of the box*, xmonad is quite bare-bones. Like most other tiling window managers, it has key bindings in place for spawning a terminal emulator, for navigating between windows, for changing layouts, for incrementing and decrementing the number of master clients, for increasing and decreasing the master factor, for toggling the floating state for the focused window, for swapping windows’ positions in the list, for moving windows to different workspaces, and for changing the active workspace. A popular addition to xmonad is *xmobar*, a lightweight, text-based status bar also written in Haskell (see Figure A.4).

# Appendix B

## Tree-based Tiling Window Managers

Tree-based tiling window managers are much more flexible than their list-based counterpart (Section 2.2.3 and Appendix A). Instead of having fixed layouts that are defined before the user even begins using the windows in them, tree-based tiling allows for custom layouts to be set up on the fly. As we shall see, different window managers handle this in quite differing ways, but one thing remains the same: the tree as the underlying data structure. We will discuss several of the more popular tree-based window managers here. Whenever it is mentioned that a window manager is ICCCM or EWMH compliant (Chapter 4), we mean a subset of these standards is implemented. All of them principally target Linux or other Unix-based platforms, and are open source.

### B.1 i3

i3<sup>[67]</sup> is a reparenting, non-compositing tree-based tiling window manager for X that was originally inspired by wmii (Section A.1). i3 is targeted at advanced users and developers, and has stated design goals of having well readable, well documented code, offering more flexible layouts using the tree as a data structure, implementing different modes (like in vim; for example, in *resize mode*, other key bindings are active than are in *normal mode*), implementing an inter-process communication interface, and not being bloated or fancy.

i3 is written in C, and interfaces with X using XCB. It supports tiling, stacking and tabbing modes, and allows the user to switch between them dynamically. It implements both the ICCCM and EWMH.

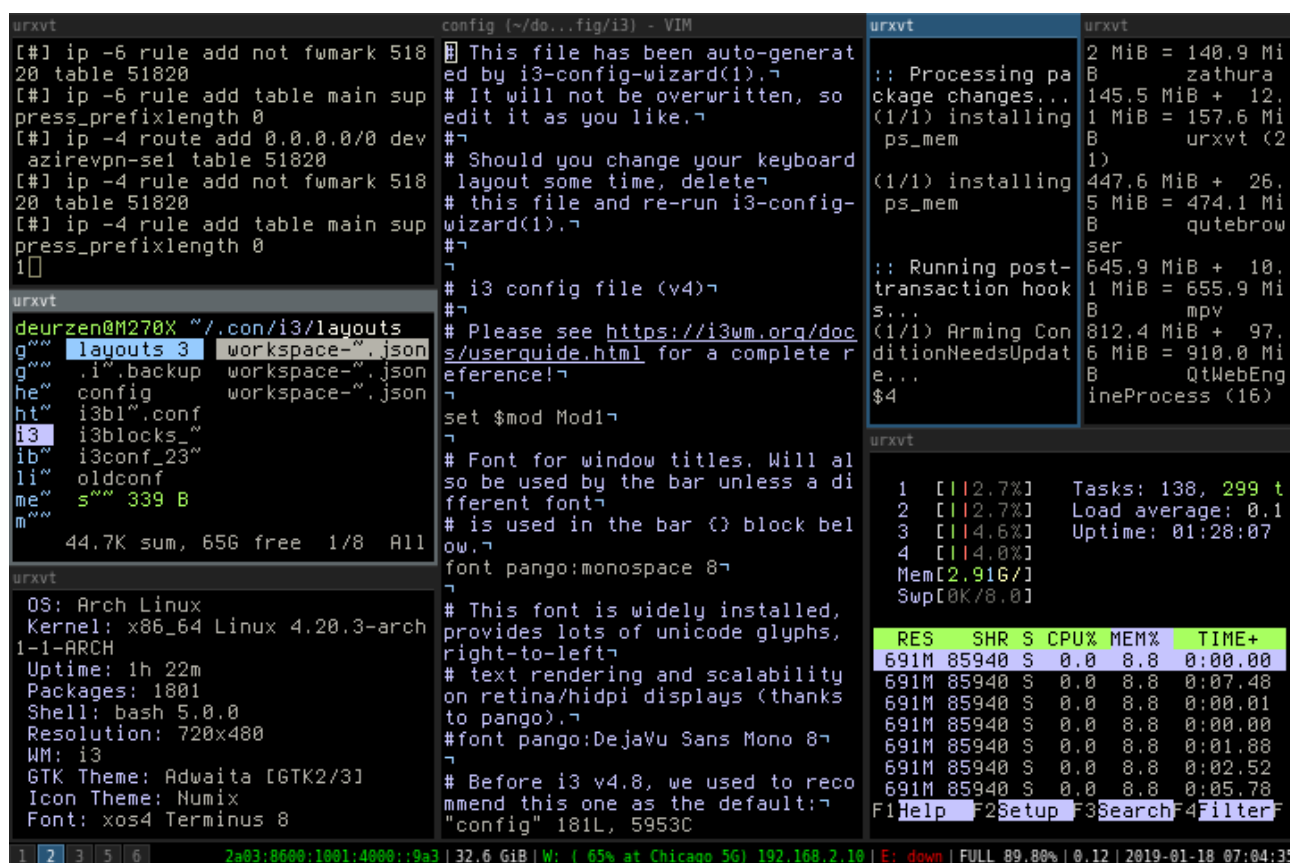
The screenshot displays the i3 window manager interface with seven tiled windows. The top-left window is a terminal showing the output of the 'i3-config-wizard' script, which configures the window manager's settings. The top-right window shows the output of the 'i3-nwmsg' command, displaying system statistics such as memory usage (2 MiB) and CPU usage (0.0%). The bottom-left window is a terminal showing the output of the 'i3-nwmsg' command, displaying system statistics such as memory usage (2 MiB) and CPU usage (0.0%). The bottom-right window is a terminal showing the output of the 'i3-nwmsg' command, displaying system statistics such as memory usage (2 MiB) and CPU usage (0.0%). The central area of the screen is occupied by a large terminal window displaying the i3 configuration file (i3config.tcl) in a vim editor. The terminal shows the configuration file's content, which includes settings for the window manager's behavior, such as the font used for window titles and the key bindings for window management. The terminal also shows the output of the 'i3-nwmsg' command, displaying system statistics such as memory usage (2 MiB) and CPU usage (0.0%). The bottom status bar of the terminal shows the system's uptime (1h 22m), the number of packages (1801), the shell (bash 5.0.0), the resolution (720x480), the window manager (i3), the GTK theme (Adwaita [GTK2/3]), the icon theme (Numix), and the font (xos4 Terminus 8).

Figure B.1: i3 with 7 self laid out tiled windows

i3 is configured by editing a plaintext file. This makes the window manager customizable without knowledge of programming. The contents of this file can be parsed and applied while the window manager is running (that is, the user can instruct the window manager to reload its configuration on demand).

Windows are stored in a general tree (such as those brought up in Section 2.2.3), in which dividers (called *split containers*) define either a horizontal or vertical arrangement of the windows under it. In

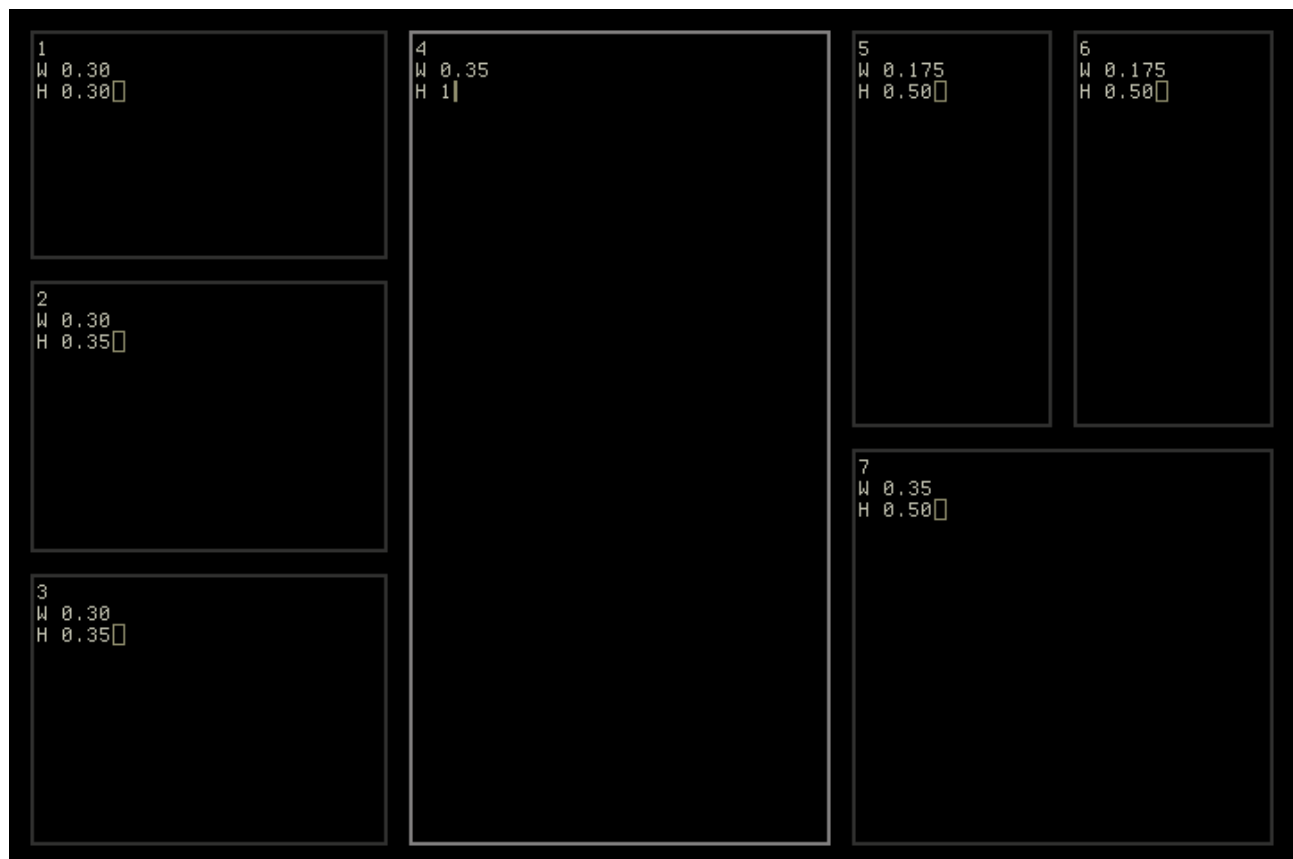
Figure B.1 we have mimicked the layout presented in Section 2.2.3.

Like other window managers that are EWMH compliant, *i3* can be used in conjunction with external status bars, docks, and program launchers. The developers of *i3* offer a status bar, called *i3status*, that is enabled by default. Of course, if the user so wishes, this can be changed through the configuration file.

## B.2 bspwm

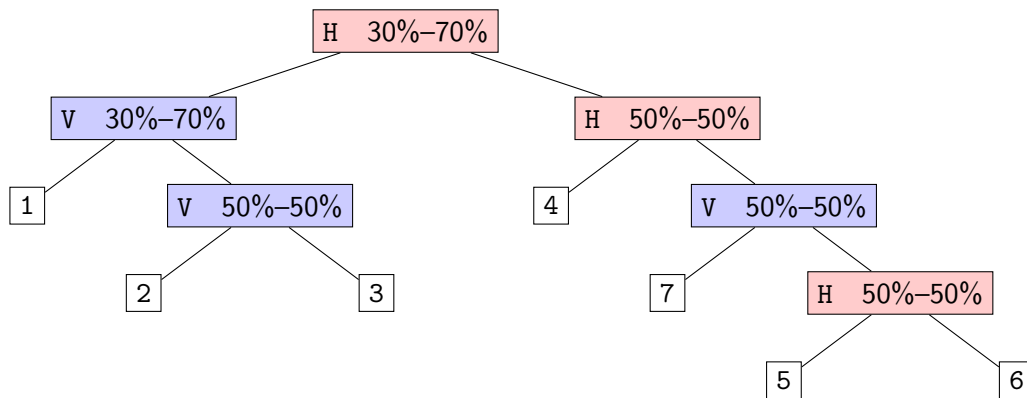
**bspwm**<sup>[68]</sup> (for *binary space partitioning window manager*) is a non-reparenting, non-compositing tiling window manager for X that represents windows as the leaves of a full binary tree. It only responds to X events, and the messages it receives on a dedicated socket. In fact, it is so bare-bones that it does not even handle keyboard or mouse input itself; an external program (such as *sxhkd*) is required to configure and conveniently interact with the window manager. The external input handler is set up to command *bspwm* using *bspc*, the window manager's client program used to configure it, and to send it instructions. *bspwm* implements the ICCCM and EWMH, and offers multi-monitor support for XRandR and Xinerama.

Whereas *i3* (Section B.1) uses a general tree to represent its windows and is therefore capable of evenly partitioning the window space amongst an arbitrary amount of windows, *bspwm*'s splits only allow for two constituent partitions. In particular, this means that every internal node has exactly two children.



**Figure B.2:** bspwm with 7 self laid out terminal emulator windows

Figure B.2 shows a best attempt to mimick the layout we achieved with *i3* in Figure B.1. Because we can only have local arrangements with two containers (being either a divider or a window), we do not have exactly the same amount of flexibility we have in tree-based tiling window managers that employ general trees. The terminal emulators' prompts each denote their window's width and height ratio relative to the width and height of the screen, respectively. As can be seen, the window in the top-left has a width of 30% of that of the screen, while the two partitions next to it to the right (which appear to be its siblings, but are not), have an accumulative width of 70%, or 35% each. Ideally, like in our *i3* example, each would have the same width of 33%. Here is the tree that effectuates the shown layout on screen.



A new window can be added above, below, to the left, or to the right of a window. By default, both windows will take up 50% of the designated window space. For example, if we were to add a window below window 4 in Figure B.2, window 4 will take up the top half of the screen, and the new window will take up the bottom half. Split ratios can be adjusted; in our example, we can specify that we want a split's new partition to take up 70% of the window space. So, by adding a new window below window 4, we get that window 4 then takes up 30% of the screen, and the new window 70%.

Besides manual splitting, `bspwm` also has *automatic mode*, in which windows are laid out along a predefined layout, comparable to the approach in list-based tiling.

### B.3 herbstluftwm

**herbstluftwm**<sup>[69]</sup> is a non-reparenting, non-compositing tree-based tiling window manager for X. It is written in C++, and it interfaces with X through Xlib and GLib. `herbstluftwm` incorporates a list-based tiling scheme within the partitions at the leaves of the tree. It is much like `bspwm` (Section B.2) in that its underlying data structure is a full binary tree.

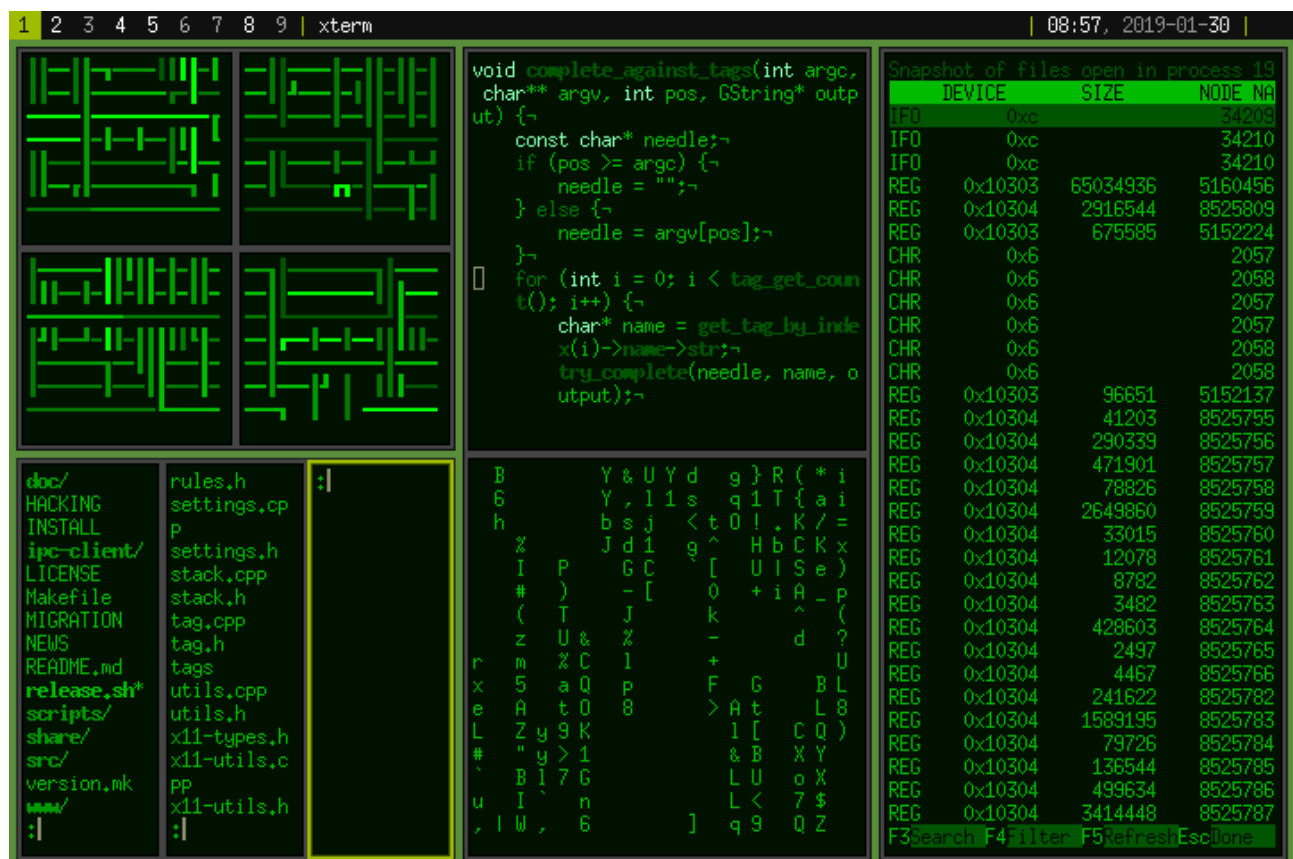


Figure B.3: `herbstluftwm` with 4 frames and 12 windows

herbstluftwm's arrangement policy is based on *frames* that are split into *subframes*, which can be split again, or filled with windows. Frames are arranged in a tree-based, user-determined manner, whereas the windows within frames are arranged along a layout, like those seen in list-based tiling. By default, herbstluftwm has four layouts: *vertical*, *horizontal*, *max*, and *grid*. The first two partition the frame in equal parts along the height or width of the frame respectively, the third gives each window in the frame the size of that frame (comparable to *monocle mode* in list-based tiling schemes), and the fourth divides the frame up into equally sized blocks horizontally and vertically.

Consider Figure B.3. We have four frames (top-left, bottom-left, center, and right), which are not children of the root, and therefore not all siblings (as this is not possible, for a binary tree is used to represent them). Instead, the frame in the center and the frame to the right are actually subframes of a frame that *is* top-level. The same holds for the two frames to the left. The subframe in the top-left contains four windows that are arranged in the grid layout. The subframe below it contains three windows, and it has the horizontal layout active. The subframe in the center is arranged along the vertical layout, and the subframe to the right actually contains three windows, of which only one is visible, due to the fact that the max layout is active in it.

herbstluftwm supports multiple monitor-independent *tags* (which it also calls *workspaces* or *virtual desktops*) that can be added or removed at runtime; each tag has its own layout, and retains its layout upon switching tags. The window manager can be configured and controlled at runtime through IPC calls by using *herbstclient*, a separate program used to send it commands, similar to bspwm's *bspc*.

# Appendix C

## X.Org Foundation Distribution

Programs provided in the core X.Org Foundation distribution include<sup>[25]</sup>:

program name(s)	description
<i>xterm</i>	a terminal emulator
<i>twm</i>	a window manager
<i>xdm</i>	a display manager
<i>xconsole</i>	a console redirect program
<i>xmh</i>	a mail interface
<i>bitmap</i>	a bitmap editor
<i>appres</i> , <i>editres</i>	resource listing/manipulation tools
<i>xauth</i> , <i>xhost</i> , and <i>iceauth</i>	access control programs
<i>xrdb</i> , <i>xcmsdb</i> , <i>xset</i> , <i>xsetroot</i> , <i>xstdcmap</i> , and <i>xmodmap</i>	user preference setting programs
<i>xclock</i> and <i>oclock</i>	clocks
<i>xfd</i>	a font displayer
<i>xlsfonts</i> , <i>xwininfo</i> , <i>xlsclients</i> , <i>xdpyinfo</i> , <i>xlsatoms</i> , and <i>xprop</i>	utilities for listing information about fonts, windows, and displays
<i>xwd</i> , <i>xwud</i> , and <i>xmag</i>	screen image manipulation utilities
<i>x11perf</i>	a performance measurement utility
<i>bdftopcf</i>	a font compiler
<i>xf86</i> , <i>fsinfo</i> , <i>fsfonts</i> , <i>fstobdf</i>	a font server and related utilities
<i>Xserver</i> , <i>rgb</i> , <i>mkfontdir</i>	a display server and related utilities
<i>xclipboard</i>	a clipboard manager
<i>xkbcomp</i> , <i>setxkbmap</i> , <i>xkbprint</i> , <i>xkbbell</i> , <i>xkbevd</i> , <i>xkbvleds</i> , and <i>xkb-watch</i>	keyboard description compiler and related utilities
<i>xkill</i>	a utility to terminate clients
<i>xfwp</i>	a firewall security proxy
<i>proxymngr</i>	a proxy manager to control them
<i>xfindproxy</i>	a utility to find proxies
<i>librx.so</i> and <i>librxnest.so</i>	web browser plug-ins
<i>rx</i>	an RX MIME-type helper program
<i>xrefresh</i>	a utility to cause part or all of the screen to be redrawn