



**CZECH TECHNICAL
UNIVERSITY
IN PRAGUE**

F8

**Faculty of Information Technology
Department of Theoretical Computer Science**

Master's Thesis

SWM – Simple Window Manager

Jan Bína

May 2020

Supervisor: Ing. Filip Křikava, Ph.D.

Acknowledgement / Declaration

I would first like to thank my thesis supervisor, Ing. Filip Křikava, Ph.D., for the introduction to this topic and thesis guidance. I would also like to thank the Czech Technical University in Prague and the University of Helsinki for providing me with a high-quality education. Finally, I would like to express my very profound gratitude to my parents, family, and friends for providing me with unfailing support and continuous encouragement throughout my years of study.

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No.121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the “Work”), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 28th May 2020

.....

Abstract / Abstrakt

This thesis deals with the design and implementation of a stacking window manager for the X Window System. A window manager is the core component of any modern graphical desktop – it is responsible for the placement and appearance of application windows on the screen. While there is a plethora of window managers, especially on the X Window System, most of them are either heavyweight window managers that are part of a desktop environment or lightweight tiling managers. In this thesis, we try to fill the gap by developing a lightweight stackable window manager that complies with the freedesktop standards such as ICCCM and EWMH, and that follows the UNIX philosophy of doing one thing and doing it well. The focus has been on simplicity, code readability, testability, and making it easy to use and extend.

Keywords: window manager, stacking window manager, X Window System, X11, Xorg, ICCCM, EWMH, Go, golang, open source

Tato práce se zabývá návrhem a implementací klasického (nedlaždicového) správce oken pro X Window System. Správce oken se stará o rozmístění a vzhled oken jednotlivých aplikací na obrazovce a je tedy klíčovou součástí jakéhokoliv moderního počítače s grafickým rozhraním. Ačkoliv již existuje mnoho správců oken, zejména pro X Window System, chybí zde jednoduchý nedlaždicový. Tato práce se snaží jeden takový vytvořit. Cílem bylo, aby splňoval freedesktop standardy ICCCM a EWMH a držel se UNIXové filozofie dělat jednu věc a dělat ji dobře. Důraz byl také kladen na jednoduchost, čitelný a testovatelný kód, a na to, aby byl správce oken snadno použitelný a rozšiřitelný.

Contents /

1 Introduction	1
Goals of the Thesis	2
Structure of the Thesis	2
2 Background	3
2.1 Windowing System	3
2.2 Desktop Environment and Window Manager	4
2.3 Types of Window Managers	5
2.3.1 Stacking Window Managers	5
2.3.2 Tiling Window Managers ..	5
2.4 Window Managers	6
2.4.1 Cwm	6
2.4.2 Openbox	7
2.4.3 Dwm	8
2.4.4 Bspwm	9
2.4.5 i3	10
2.4.6 Summary	11
2.5 The X Window System	12
2.5.1 History	12
2.5.2 Architecture	12
2.5.3 The X Protocol	12
2.5.4 Properties and Atoms ...	13
2.5.5 Window Hierarchy	13
2.5.6 X Client Libraries	14
2.6 ICCCM	15
2.6.1 Selection	16
2.6.2 Clients Actions	16
2.6.3 Creating a Top-Level Window	16
2.6.4 Client Properties	18
2.6.5 Changing Window State	20
2.6.6 Configuring the Win- dow	20
2.7 EWMH	21
2.7.1 Pagers and Taskbars	21
2.7.2 Scope of EWMH	22
2.7.3 Additional States	22
2.7.4 Large Desktops	22
2.7.5 Virtual Desktops	22
2.7.6 Sticky Windows	23
2.7.7 Activation	23
2.7.8 Root Window Proper- ties	23
2.7.9 Other Root Window Messages	24
2.7.10 Application Window Properties	24
2.7.11 Stacking Order	27
3 Design	28
3.1 Tools	28
3.1.1 Xdotool	28
3.1.2 Wmctrl	29
3.1.3 Sxhkd	30
3.2 Configuration and Control- ling	30
3.3 Swmctl and Swmrc	31
3.4 Desktops – Groups	32
4 Implementation	33
4.1 Tools	33
4.1.1 Go	33
4.1.2 X Libraries for Go	33
4.1.3 Xephyr	35
4.1.4 Xvfb	36
4.2 Project Structure	36
4.3 Inter-Process Communica- tion	38
4.4 Desktops – Groups	39
4.5 Becoming a Window Man- ager	40
4.6 Window Decorations and Reparenting	41
4.7 Moving and Resizing	43
4.8 Stacking	44
4.9 Window Cycling	45
4.10 Scriptability	45
4.11 ICCCM and EWMH Com- pliance	47
4.12 Testing	47
4.12.1 Testing Architecture	48
4.12.2 Testing Process	49
4.12.3 Test Coverage	49
4.13 Code Management and Con- tinuous Integration	51
4.13.1 Code Style	51
4.13.2 Continuous Integration ..	51
5 Conclusion	54
Future Work	54
References	55
A Acronyms	61

B Contents of enclosed SD card 62



Tables / Figures

2.1. Window properties	14	2.1. X architecture scheme	4
2.2. Summary of Window Manager Property Types	18	2.2. Wayland architecture scheme	4
		2.3. Stacking window manager layout	5
		2.4. Tiling window manager layout ..	6
		2.5. Cwm desktop	7
		2.6. Openbox windows and menu	8
		2.7. Dwm desktop	9
		2.8. Bspwm desktop	10
		2.9. i3 desktop	11
		2.10. Pager	21
		2.11. Taskbar	21
		4.1. Xephyr running swm	36
		4.2. Source code structure	38
		4.3. Info box showing group membership	40
		4.4. Window decorations	42
		4.5. Window decorations of Evince and Google Chrome	42
		4.6. Window cycling UI	45
		4.7. Windows in a grid layout	46
		4.8. Failing CI checks preventing merge	52
		4.9. Passed CI checks	53

Listings /

2.1. Window property lookup using Xlib	15
2.2. Window property lookup using XCB	15
2.3. Selection acquiring mechanism.....	17
3.1. Xdotool commands.....	29
3.2. Wmctrl commands	30
3.3. Example configuration file for sxhkd	30
4.1. Window property lookup using XGB	34
4.2. Xgbutil API showcase.....	35
4.3. X event loop.....	35
4.4. Xgbutil event handling.....	36
4.5. Go module configuration file ..	37
4.6. Cycling commands usage	45
4.7. Script to organize windows on desktop	46
4.8. Testing script	48
4.9. Test output	49
4.10. Test case example	50
4.11. Example of a well-formed Go code.....	52

Chapter 1

Introduction

Window manager is a tool that controls the placement and appearance of windows on the screen. It is probably one of the most important parts of a graphical user interface of today's personal computers. Generally, we can divide window managers into two categories, stacking window managers and tiling window managers. Stacking window managers provide the traditional desktop metaphor and are the most widely used type. They allow users to stack windows on top of each other, move them around, and resize them freely. Tiling window managers, on the other hand, organize windows on the screen so they do not overlap and use all available screen space.

The first computer shipped with a working WIMP (windows, icons, menus, pointer) GUI was the Xerox Alto in the 1970s, which used a stacking window management [1]. Its successor, Xerox Star, was the first commercial personal computer using the desktop metaphor, and it used tiling for most main application windows [1]. The first window manager for X11 was the Ultrix Window Manager (uwm), which was released in 1985 [2]. It was soon replaced by the Tab Window Manager (twm) [3], which is still standard with X.Org Server and is available as part of many X Window System implementations. Probably the first desktop environment for Unix was the Common Desktop Environment (CDE) [4], announced in 1993 [5]. It was developed as a unified desktop environment for many commercial proprietary Unixes that dominated the workstation market: IBM's AIX, Digital's Tru64, or Sun's Solaris [6]. Then, KDE [7] was announced in 1996 with a goal of creating an environment in which users could expect things to look, feel, and work consistently [8]. Its first version was then released in 1998 [9]. By that time, other desktop environments were already released, such as Xfce [10] (1996), or Enlightenment [11] (1997), and they were soon followed by GNOME [12] (1999) [5]. Xfce, Enlightenment, KDE, and GNOME are among the most popular desktop environments to this day [13].

Desktop environments target a wide range of users and provide lots of built-in applications and utilities, so they can be used without much configuration. They also come with stacking window managers, because those are controlled using a pointing device and thus easier for most users. Among power users though, lightweight desktops are becoming more and more popular lately, because they are light on system resources and thus fast. This leads to the increasing popularity of tiling window managers, which are keyboard-oriented and target power users as well. One of the first lightweight tiling window managers was dwm [14], which was created in 2006. This has sparked the development of dozens of other tiling managers.

While lots of lightweight tiling window managers have been created in recent years, lightweight and keyboard-oriented stacking window managers seem to be missing. Existing stacking window managers are either part of a desktop environment and thus not keyboard-oriented, or they are outdated, written in an unmaintainable and inextensible style, and lacking crucial features like EWMH support. Therefore, we feel a need

Chapter 2

Background

2.1 Windowing System

A windowing system is a collection of software that creates the basic GUI (graphical user interface) on computer display screens, including the drawing of windows and other graphics primitives for application programs [18]. From a programmer’s point of view, a windowing system implements graphical primitives such as rendering fonts or drawing a line on the screen, effectively providing an abstraction of the graphics hardware from higher-level elements of the graphical interface like window managers [19].

The most popular windowing system among Linux users is the *X Window System* [20], and its reference open-source implementation *X.Org Server* provided by the X.Org Foundation [21]. The X Window System (also referred to as X or X11) originated at MIT in 1984 [22]. Another windowing system is *Wayland*, which is intended as a simpler replacement for X [23]. Wayland started in 2008 and comes with different architecture, trying to solve problems in X’s approach.

The main problem in X’s approach that Wayland aims to solve, is that it does not isolate applications from each other. As a result, all X applications have access to everything on the screen, all can register to receive every keystroke, even if they do not have input focus, and they can even inject keystrokes into other windows [24]. The architecture schemes of X and Wayland are depicted in Figures 2.1 and 2.2. The numbers show the flow of the events from the input device to the point where the change it affects appears on the screen. The only difference between those two schemes is the compositor, which is responsible for rendering the entire screen contents. Since the window location on the screen is controlled by the compositor and may be transformed in several ways (e.g., scaled-down, rotated), the X server does not have the information to decide which client should receive the event. The X server acts as a middleman that introduces an extra step between applications and the compositor and an extra step between the compositor and the hardware. In Wayland, on the other hand, the compositor is part of the display server. The Wayland protocol lets the compositor send the input events directly to the clients and lets the client send the response directly to the compositor [25].

Even though Wayland is meant to replace X in the future, it is not happening quite yet. One of its problems is apps compatibility – software engineer Samuel Walladge said in his 2019 article “Are we Wayland yet?” [26]: “I soon discovered that many many apps either only supported X11, or crashed on Wayland.” Those apps then require XWayland, which provides an embedded X server for apps that do not support Wayland yet. Another problem is that “screen recording or sharing apps just do not work” [26], which is caused by the Wayland architecture that isolates each client. Walladge also mentions stability issues and concludes that the time for switching to Wayland did not come yet. Similar problems and conclusions are also outlined in articles “X11 Sucks... So What’s Up With Wayland?” [27] and “Wayland v/s Xorg : How Are They Similar

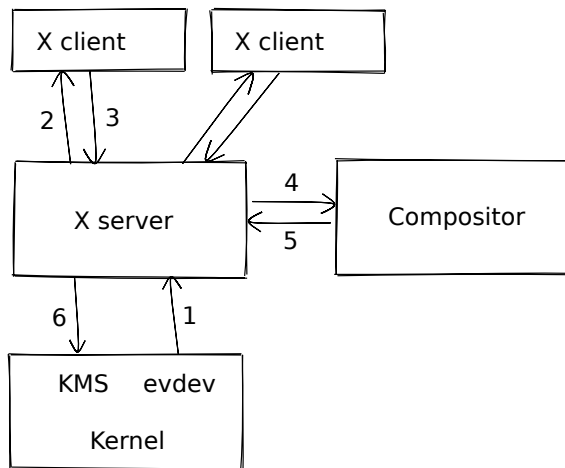


Figure 2.1. X architecture scheme

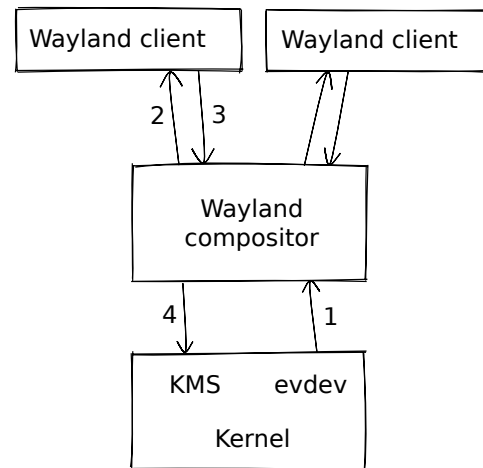


Figure 2.2. Wayland architecture scheme

& How Are They Different” [28]. Because of that, we decided to build swm for the X Window System.

2.2 Desktop Environment and Window Manager

A desktop environment is an implementation of the desktop metaphor, which was introduced by Alan Kay at Xerox PARC in 1970 to help users interact with the computer more easily [29]. The first commercial personal computer to use the desktop metaphor was the Xerox Star [1]. The developer of its interface, David Smith, described it in 1982 article [30]: “Every user’s initial view of the Star is the desktop, which resembles the top of an office desk, together with surrounding furniture and equipment. It represents a working environment, where projects and accessible resources reside. On the screen are displayed pictures of familiar office objects, such as documents, folders, file drawers, in-baskets, and out-baskets. These objects are displayed as small pictures, or icons.”

A desktop environment, as we know it today, bundles together a variety of components to provide common graphical user interface elements such as icons, panels, wallpapers, and desktop widgets [13]. Most desktop environments also include a set of integrated applications and utilities, such as text editor, file manager, or web browser. One of the most important parts of a desktop environment is a window manager. While the desktop environment provides its own window manager, it can be usually replaced with another compatible one [13]. There is a great offer of desktop environments for Linux, for example, Arch Wiki [13] lists more than 20 of them. Some of the most popular are KDE [7], GNOME [12], or Xfce [10].

A window manager is system software that manages windows. That means, it controls the placement and appearance of windows within a windowing system in GUI [31]. It allows windows to be opened, closed, resized, moved, maximized, minimized, and more. The window manager is also responsible for tracking which window is currently active and thus receiving the user’s input. Some window managers are specifically developed to be part of a desktop environment, others are instead designed to be used standalone. Using a standalone window manager allows the user to create a more lightweight and customized environment, tailored to his own specific needs [31]. There is a great offer of window managers for Linux. For example, Arch Wiki provides a list of window

managers categorized by type [31], that lists more than 60 of them. In Section 2.4, we will pick some of them and discuss them in more detail.

2.3 Types of Window Managers

Window managers are usually divided into two classes, based on how windows are drawn on the screen.

2.3.1 Stacking Window Managers

Stacking window managers, also known as floating, provide the traditional desktop metaphor used in commercial operating systems like Windows and OS X [31]. Windows act like pieces of paper on a desk – they can be stacked on top of each other, moved around, and resized freely by the user. Moving or resizing one window does not affect the position or size of other windows, only their visible area. Window manager must maintain the stacking order of windows and only the top window on the stack is guaranteed to be fully visible. A window is usually moved to the top of the stack by the window manager when the user starts to interact with it. Example layout of a stacking window manager is depicted in Figure 2.3. Windows are overlapping and only focused window (highlighted in blue) is fully visible.

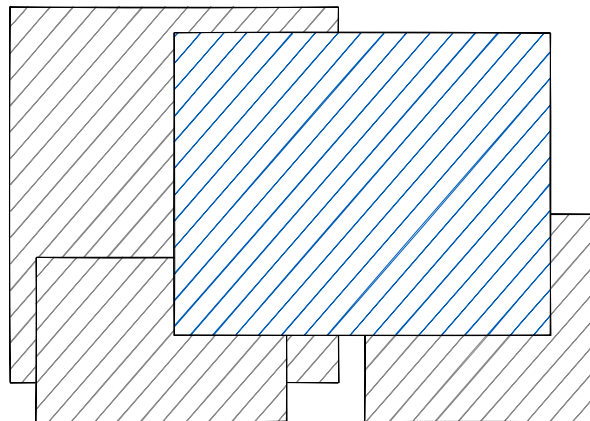


Figure 2.3. The layout of a stacking window manager

2.3.2 Tiling Window Managers

Tiling window managers tile the windows so that none are overlapping and they usually use all the available screen space. They typically make very extensive use of keyboard shortcuts and have less or no reliance on the mouse [31]. It is not possible to move or resize windows freely – enlarging a window shrinks its adjacent windows and vice versa, moving is typically done by swapping the window’s position with another window. Some types of windows, such as dialogs and pop-up windows, are not suited for tiling though. Most tiling window managers detect those windows and leave them in a floating state, keeping them above the windows that are tiled. Example layout of a tiling window manager is depicted in Figure 2.4.

Tiling window managers could be split into two more categories:

- **Dynamic** tiling window managers tile windows based on preset layouts. They usually offer many different layouts of window placement and the user can dynamically switch

between them. For example, one of the most common tiling layouts is called *master-stack* layout, in which one window (master) is considered the most important and has dedicated half of the screen space, while the rest of the windows are displayed one below the other on the second half of the screen.

- **Static (manual)** tiling window managers do not use layouts. They let the user decide where windows should be placed. Most typically, when a new window is created, one of the existing windows is shrunk to half its width/height and the new window takes freed up space. The user can choose the window that will be shrunk and also the direction (vertical or horizontal). Sometimes, the user can also choose the default split ratio, so that the new window does not take half, but, for example, only a third of the existing window space.

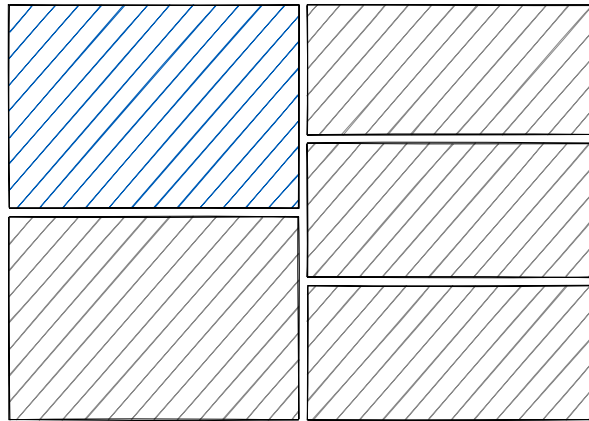


Figure 2.4. The layout of a tiling window manager

2.4 Window Managers

As mentioned before, there is a great offer of window managers for Linux. In this section, we will go through some of them, mainly those that inspired swm in some way.

2.4.1 Cwm

Cwm (Calm Window Manager) is a stacking window manager for the X Window System. It describes itself as a “window manager which contains many features that concentrate on the efficiency and transparency of window management while maintaining the simplest and most pleasant aesthetic” [32]. It is the default window manager on OpenBSD.

Cwm is oriented towards heavy keyboard usage – resizing, moving, hiding, raising, or lowering windows, all can be done using keyboard shortcuts [33]. It could be even configured to move the mouse cursor using the keyboard so that one could use the computer without any pointing device. The user can either define custom shortcuts using cwm’s configuration file or use the defaults.

Interface of cwm is very minimal – it only draws a one-pixel border around windows by default, the width of the border and its color can be configured though. Cwm offers several menus, which can be used to launch applications or switch between running applications. The principle of these menus is that cwm shows a list of relevant content

and user can search in it and finally pick one of the filtered options [34]. Windows are searched by their current title, old titles and by their label, which is a custom string user can assign to every window.

Instead of the traditional virtual desktop concept, cwm is using groups. There are nine groups with IDs 1–9 and a special “sticky” group with ID 0. Each window is assigned to one of those groups, the user can then change its group with a keyboard shortcut. While the sticky group is always visible on the screen, the normal group can be either visible or hidden. This means that unlike virtual desktops, multiple groups could be visible at the same time. The visibility of the group can be controlled by keyboard shortcuts as well.

Figure 2.5 shows a cwm desktop with two windows and one of those menus. This menu lists all the managed top-level windows, there are various information included:

- the group ID on the left,
- “!” indicates active (focused) window,
- “&” indicates hidden windows,
- the label of the window in square brackets,
- and finally the title of the window.

All the cwm’s configuration is done using configuration file `.cwmrc`. The user may add, modify or remove shortcuts, change colors and other aspects of window management behavior [34]. One of the drawbacks of cwm is that it does not support most of the EWMH protocol.

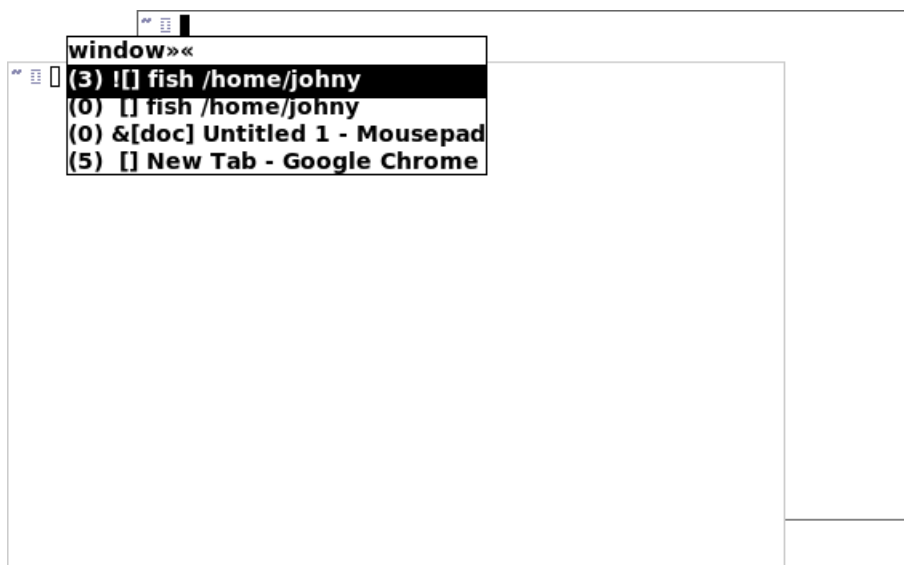


Figure 2.5. Cwm desktop with two windows and a menu showing all managed windows

■ 2.4.2 Openbox

Openbox is a stacking window manager. It describes itself as a “minimalistic, highly configurable, next generation window manager with extensive standards support” [35]. It is designed to be fully compliant with ICCCM and EWMH protocols. It is the default window manager of desktop environments LXDE [36] and its successor LXQt [37], and it is also offered as one of the options for default window manager in many Linux distributions, for example Ubuntu [38] and Manjaro [39].

Although Openbox claims to be minimalistic, it is much more advanced than `cwm`. Its default UI could be seen in Figure 2.6. It comes with full-featured window decorations that include a border around the window and title bar with window title and buttons to minimize, maximize, and close the window. It also includes a *root menu* on the desktop. Using this menu, one can launch applications, bring minimized windows up again, or start various Openbox configuration tools. Openbox does not come with a bundled taskbar, though, and advises its users to use one of many stand-alone EWMH compatible taskbars.

Openbox is configured by two configuration files, `rc.xml` and `menu.xml`. The first one (`rc.xml`) is the main configuration file, responsible for determining the behavior and settings of the overall session, including keyboard shortcuts, theming, virtual desktops settings etc. [40] The second one (`menu.xml`) defines the type, behavior, and items of aforementioned desktop menu. Although the default provided is a static menu (it will not automatically update when new applications are installed), it is possible to employ the use of dynamic menus that will automatically update as well [40].

Configuration files are not meant to be easily readable or editable – Openbox comes with applications to edit them, one for `rc.xml`, called `obconf` and one for `menu.xml`, called `obmenu`. Using `obconf`, user can, for example, easily switch Openbox theme, that controls the looks of window decorations and menus. `Obconf` itself includes many themes to choose from, more themes are available online and one can also easily create new or modify existing theme using provided GUI application [40].

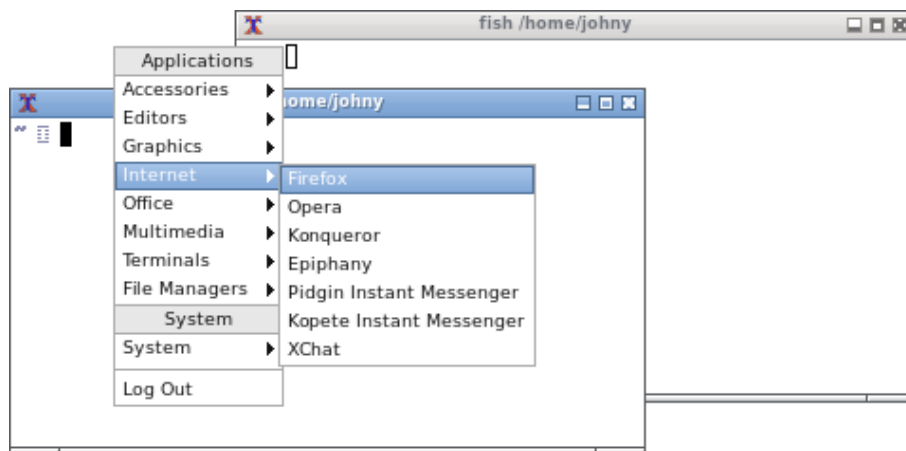


Figure 2.6. Openbox with two opened window and a menu

2.4.3 Dwm

Dwm (Dynamic Window Manager) is a dynamic tiling window manager. It manages windows in tiled, monocle and floating layouts. All of the layouts can be applied dynamically, optimising the environment for the application in use and the task performed. [14]

Dwm follows the philosophy of its authors, *suckless.org*, that is about keeping things simple, minimal and usable [41]. All their projects focus on advanced and experienced computer users. They believe that ingenious software is simple and that as the number of lines of code in software shrinks, the less the software sucks [41]. Following this philosophy, `dwm`'s source code is intended to never exceed 2 000 source lines of code and it has no configuration file, it can only be configured by editing its C source code [14].

Dwm is extremely lightweight and fast and its interface is very minimal. It only draws a small customizable border around windows to indicate the focus state. Windows could be spread across nine desktops called tags and it is possible to show windows from multiple tags at once. It comes with its own status bar which displays all available tags, the layout, the number of visible windows, the title of the focused window, and the text read from the root window name property [14]. It is not really possible to use third party taskbars, because dwm does not support EWMH.

As stated before, dwm comes with three layouts by default. In tiled layout, windows are managed in a master and stacking area. The master area contains the window which currently needs most attention, whereas the stacking area contains all other windows. In monocle layout, all windows are maximized to the screen size and in floating layout, windows can be resized and moved freely.

Interesting concept of dwm are *patches*. Since the core dwm source code is very minimal and is kept under 2000 lines of code, a lot of functionality is missing, and so users started to create patches to add functionality they wanted. Those patches are in a form of simple git diff files¹, and could be found on dwm's project website [42]. Most of those plugins implement new types of layout (e.g., deck, fibonacci, grid), or modifies behavior or style of the status bar.

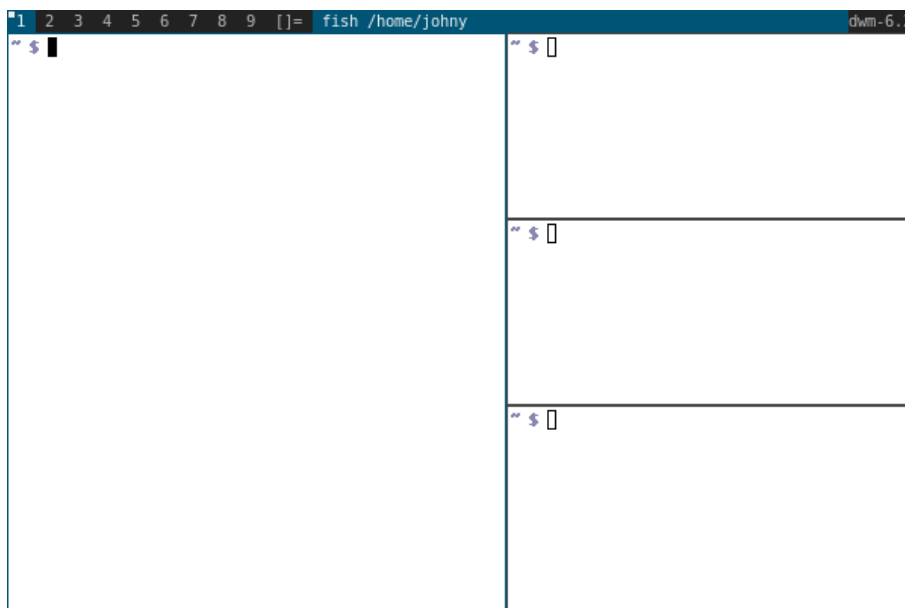


Figure 2.7. Dwm desktop with the status bar and four open windows in its default layout

■ 2.4.4 Bspwm

Bspwm (binary space partitioning window manager) is a manual tiling window manager. It represents windows as the leaves of a full binary tree, supports multiple monitors and is configured and controlled through messages [43–44]. It supports a subset of ICCCM and EWMH protocols, it does not state what is supported and what is not, though [43].

Bspwm is very minimalistic in terms of UI – it only draws a simple solid color border around its windows and has no build-in taskbar or menus. Bspwm only responds to X events, and messages it receives on a dedicated socket. It comes with a standalone

¹ <https://git-scm.com/docs/git-diff>

command-line application called *bspwm*, that writes messages on the *bspwm*'s socket [43]. It also does not handle any keyboard or pointer inputs. For this it relies on a third party program that translates keyboard and pointer events to *bspwm* invocations [43]. It recommends *sxhkd* [45] (developed by the same author). Upon startup, *bspwm* runs its configuration file, *bspwmrc*, which is simply a shell script that calls *bspwm* (and, possibly, other utilities) [43].

Since *bspwm* represents windows as the leaves of a full binary tree, each inner node has exactly two children and each leaf node holds exactly one window. Each inner node is responsible for splitting its screen space in two parts and this split is defined by two parameters: the type (horizontal or vertical) and the ratio (a real number from interval $(0, 1)$) [43]. New windows are inserted into a window tree at the specified insertion point using specified insertion mode, which is configurable [43].

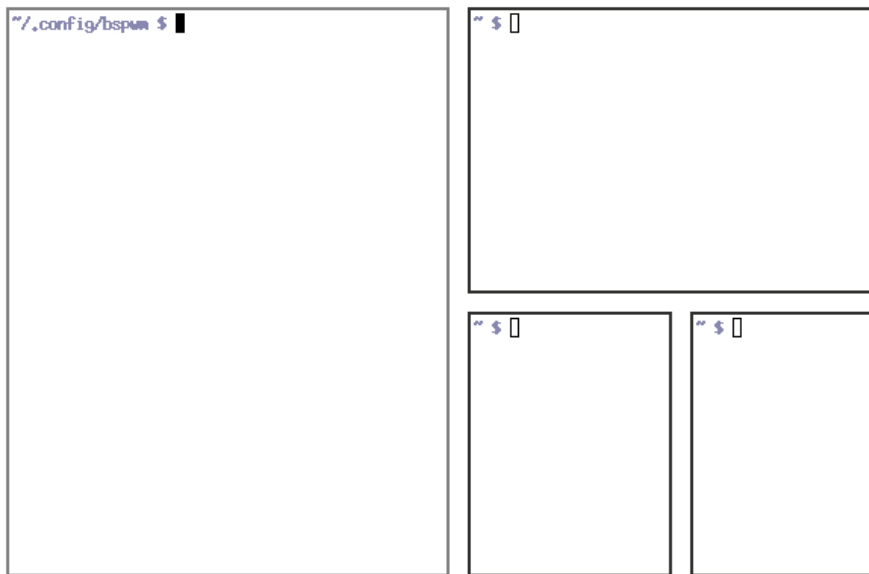


Figure 2.8. Bspwm desktop with four open windows in its default layout

■ 2.4.5 i3

i3 is a dynamic tiling window manager that is primarily targeted at developers and advanced users [46]. It uses a tree as a data structure for windows, which, according to its authors, allows for more flexible layouts than the column-based approach used by other window managers [46].

Unlike *dwm*, *i3* does not try to be as minimal as possible, but it does not want to be bloated either. This is described in one of its initial goals [46] like this: “Don’t be bloated, don’t be fancy (simple borders are the most decoration we want to have). However, we do not enforce unnecessary limits such as a maximum amount of source lines of code. If it needs to be a bit bigger, it will be.”

The default UI of *i3wm* is depicted in Figure 2.9. It is not that simple as UI of *dwm* or *bspwm*, but it is not very complex either. Windows have a simple border and also a title bar, that only shows the name of the window. It also comes with its panel called *i3bar*, which is used by default (see screenshot). Since *i3* supports EWMH, any other standalone panel could be used as well, though.

i3 offers three layouts for windows:

- split vertically/horizontally – windows are sized so that every window gets an equal amount of space. This can be done either vertically or horizontally.
- stacking – only focused window is displayed. On top of it is a list of the rest of the windows.
- tabbed – only focused window is displayed. On top of it are tabs representing the rest of the windows.

All those layouts can be combined on the screen using so-called containers. This means that inside a container that is tabbed can be another container that is using a split layout, for example. An example is depicted in Figure 2.9. The leftmost window is in fact a container that is using a tabbed layout and has two windows. Similarly, the container right next to it has two windows in a stacking layout. A window can be also completely removed from the tiling layout and managed as a floating window.

i3 is highly configurable using its configuration file. User can configure the UI (border width and color for multiple window states, fonts, etc.), keyboard shortcuts, i3bar, and the behavior of the window manager overall. For example, the user can choose that windows will not be focused upon opening, or that specific window will be in a floating state by default.

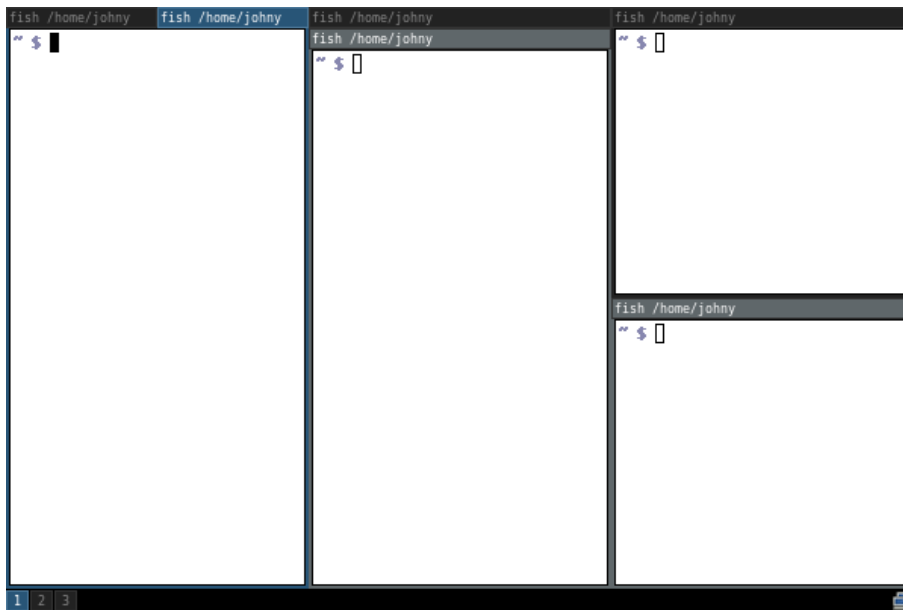


Figure 2.9. Desktop of i3 with four containers and six open windows

■ 2.4.6 Summary

In this section, we discussed five window managers. We picked some from each category – cwm and Openbox represented stacking window managers, dwm, bspwm, and i3 represented tiling window managers. In each category, we also showed different approaches. Cwm and dwm represent minimal window managers with very minimal UI, configuration, and no EWMH support. Bspwm stands somewhere in the middle, with its minimal UI, but quite advanced functionality and EWMH support. Finally, Openbox and i3 represent window managers with advanced UI (full-fledged themeable window decorations in Openbox, title bars and tabs in i3) and advanced configuration.

There are many more window managers out there, both stacking and tiling, their functionalities are not much different from those presented here, though. To mention a few more window managers, we can name Xfwm [10] or Kwin [7], which are both stacking window managers and both are developed as a part of a desktop environment, Xfce and KDE, respectively. For tiling window managers, we can mention awesome [47] or xmonad [48], which were both inspired by dwm and they build upon it. Another popular tiling window managers are herbstluftwm [49] or Qtile [50]. For an extensive list of both stacking and tiling window managers, we can recommend Arch Linux Wiki [31].

2.5 The X Window System

The X Window System is a network transparent window system that runs on a wide range of computing and graphics machines [51].

In this section, we will cover how the X server works, with a focus on communication with its clients. We will not cover how the X server handles resources, color, graphics, text, or input as it is not that important for window management.

2.5.1 History

The X Window System was created in the mid-1980s at the Massachusetts Institute of Technology. To support further development, a member-funded consortium was formed in 1988, which was later moved out of MIT, creating an independent, stand-alone organization – the X Consortium. All rights to the X Window System were assigned to the X Consortium in 1994. When the X Consortium closed its doors in 1996, all rights to the X Window System were transferred to The Open Group (known as the Open Software Foundation). The X.Org Foundation was formed in 2004 as the successor to the X.Org Group at The Open Group. The purpose of the X.Org Foundation is to foster the development, evolution, and maintenance of the X Window System. Membership in the X.Org Foundation is free and open to anyone. The X.Org Foundation hosts a public git repository of the source code on freedesktop.org. [22]

The X protocol has been at version 11 since September 1987 [52]. Many revisions have been released since then, the latest one being X11R7.7 released in June 2012 [52].

2.5.2 Architecture

The X Window System is based on a client-server model. The server controls display and input devices, such as keyboard and mouse. The client is an application that receives input events from the server and sends output and information requests to the server. The X architecture allows the clients and the server either to run on the same machine or on different machines that are connected by a network. [53]

The window manager is a special X Server client that has control over the layout of windows on the screen. To enforce this authority, the window manager is using certain X protocol features that will be discussed later on.

2.5.3 The X Protocol

The X Protocol is a standard protocol that is used by the X Window System to exchange information between the X Server and its clients [54]. Information is exchanged by sending messages. The X protocol defines these four types of messages [54]:

- Request – generated by the client and sent to the server.
- Reply – sent from the server to the client in response to some requests.
- Event – sent from the server to the client.
- Error – like an event, but it is handled differently by clients.

To get some information about the window, for example, the client can send the `GetProperty` request, to which server responds with the `GetProperty` reply (or error, eventually). This could be seen in Listing 2.2. X Server does not send a reply to all requests, though. For example, to change the size and location of the window, the client will send the `Configure` request. This request is processed by the window manager, which sends the `ConfigureNotify` event in response, describing how the size and location of the window was (or was not) changed.

To receive events, the client must register for them. When doing so, the client must specify event types that it is interested in, only those will be sent to him by the X Server. After initialization, typical X Server client runs an event loop – an infinite loop that waits for events coming from the X Server and responds to them.

■ 2.5.4 Properties and Atoms

Properties are arbitrary data attached to the window [54]. Each property is characterized by a name, a type, and a value [54]. The client can get any property of any window by issuing the `GetProperty` request mentioned before. The client can also change some properties of its windows, or send `ChangeProperty` request to request property change of some other client's windows – only window managers and other special clients (pagers) typically do this.

For example, Table 2.1 shows some of the properties retrieved from the top-level window of an application using the `xprop` utility [55]. We can see, for example, a property named `_NET_WM_DESKTOP` with type `CARDINAL` and value 1. This property is set by the window manager and tells other clients on which virtual desktop this window resides. We will discuss the meaning of some significant properties later on in more detail.

Apart from the property name, which is an ASCII string, each property also has a unique integer ID called an *atom*. Atom is just a nickname for a property, so that arbitrary-length property name strings do not have to be transferred back and forth between the client and the server [56]. A property is uniquely identified by an atom and a window [56].

One of the most important uses of properties is to communicate information from applications to the window manager and vice versa [56]. The application sets properties on its top-level window, window manager retrieves them and use them in some way. For example, application sets the `WM_NAME` property (see Table 2.1) and window manager will display this name in the title bar of the window. Another example can be the `_NET_WM_ALLOWED_ACTIONS` – this one is set by the window manager and tells the client which actions are supported by the window manager for that specific window.

■ 2.5.5 Window Hierarchy

X windows are arranged in a tree hierarchy. At the top of this hierarchy is so-called *root window* that has no parents, all other windows always have exactly one parent window. The root window fills the entire screen and is created by the X Server on startup. [57]

Property name and type	Value
WM_STATE(WM_STATE)	window state: Normal icon window: 0x0
_NET_WM_ALLOWED_ACTIONS(ATOM)	_NET_WM_ACTION_MOVE, ...
_NET_WM_DESKTOP(CARDINAL)	1
_NET_WM_WINDOW_TYPE(ATOM)	_NET_WM_WINDOW_TYPE_NORMAL
_NET_WM_STATE(ATOM)	_NET_WM_STATE_MAXIMIZED_VERT, ...
WM_PROTOCOLS(ATOM)	WM_DELETE_WINDOW, WM_TAKE_FOCUS
WM_CLASS(String)	jetbrains-goland, jetbrains-goland
_NET_WM_NAME(UTF8_STRING)	swm [~/projects/swm] - .../cmd/swm/main.go
WM_NAME(String)	swm [~/projects/swm] - .../cmd/swm/main.go

Table 2.1. Window properties retrieved using xprop [55]

When a client of the X Server creates its first window, it is created as a child of the root window. The children of the root window are called top-level windows and those are managed by the window manager. Each top-level window can also have its own children, but those are managed by the client itself. Typically, a client creates many windows inside its top-level window to create application features such as buttons and text boxes. [57]

2.5.6 X Client Libraries

Two official helper libraries that provide API for talking to the X Server exist, xlib and XCB [58].

Xlib

Xlib, also known as libX11, is the original C language X11 API, released in 1985. It was designed to look like a traditional library API, hiding the fact that calls result in protocol requests to a server. Calls that don't require a response from the X server are queued in a buffer to be sent as a batch of requests to the server. Those that require a response flush all the buffered requests and then block until the response is received. This mix of synchronous and asynchronous behavior causes some problems because it is not obvious which calls implicitly flush the buffer and which do not.[58]

XCB

XCB is a second attempt at defining a C language binding for X11. It was first released in 2001, after many years of experience with Xlib, learning from it, as well as from other protocol interface libraries. XCB makes the client-server nature of the protocol explicit in its design. The client decides when to flush the request buffer, when to read results, and when to wait for the server to respond.[58]

Comparison

In Listings 2.1 and 2.2 (both taken from the official developer's guide [58]), we can see a comparison of those two libraries on the task of looking up a window property. Xlib generates the request to the X server to retrieve the property and appends it to its buffer of requests. Because this type of request requires a response, Xlib flushes the buffer to send its contents to the X Server. When Xlib receives the reply from the X Server, it returns it to the client. If there were requests preceding the client's request, the client

must wait until the X Server processes all of them. XCB functions, on the other hand, map directly onto the protocol. There are separate functions to put requests into the outgoing buffer and to read results back from the X Server. Thanks to that, one can send many requests to the X Server at once and then wait for all the replies at once, minimizing the communication overhead. [58]

```

1 XGetWindowProperty(
2     dpy, win, atom, 0, 0, False,
3     AnyPropertyType, &type_ret, &format_ret,
4     &num_ret, &bytes_after, &prop_ret
5 );

```

Listing 2.1. Window property lookup using Xlib

```

1 cookie = xcb_get_property(
2     dpy, False, win, atom, XCB_GET_PROPERTY_TYPE_ANY, 0, 0
3 );
4 // do something while waiting for the response
5 reply = xcb_get_property_reply(dpy, cookie, NULL);

```

Listing 2.2. Window property lookup using XCB

Xlib and XCB are compatible, meaning that one can mix calls to the first with calls to the other. This compatibility was achieved by rebuilding libX11 as a layer on top of libxcb. They share the same X server connection and pass control of it back and forth. That option was introduced in libX11 version 1.2, and since version 1.4, released in 2010, it is always present (not only optional).[58]

Most applications should call Xlib and XCB sparingly, and rather utilize higher-level toolkits that provide more efficient programming models [58]. Window managers have to usually call XCB or Xlib directly, though. For the implementation of swm, we used the X Go Binding library [59], which is Go wrapper of XCB. We will talk about it in more detail in Chapter 4.

2.6 ICCCM

It was an explicit design goal of X Version 11 to specify a mechanism, not a policy. Because of that, a client that communicates with an X11 server using the protocol defined by the X Window System Protocol (discussed in Section 2.5) may operate correctly in isolation but may not coexist properly with others sharing the same server [15].

Standardized communication is important especially for window managers and other special clients like docks, toolbars or pagers, because they need to communicate with the rest of the clients as well as with each other. Inter-Client Communication Conventions Manual is one of two standards that define how X Window System clients should interact with one another. It was designed at the X Consortium in 1988. Its latest version 2.0 was released in 1994 [15].

In this section, we will cover parts of ICCCM that are related to window managers and therefore important for us. ICCCM manual [15] will be constantly referenced here, so we will explicitly refer only other sources if they are used.

■ 2.6.1 Selection

Selections are the primary mechanism that X11 defines for the exchange of information between clients. There can be an arbitrary number of selections, each of them is named by an atom, and they are global to the server. Each selection can be owned by some client and attached to a window created by that client.

For our window manager use case, we will be concerned about ownership of a selection named `WM_Sn`, where `n` is the screen number. To do its job, the window manager needs to register for `SubstructureRedirect` events on the root window of the screen it wants to manage (we will discuss this in more detail in Section 4.5). Since only one client can be registered for substructure redirection on any given window at any given time, we need some mechanism to inform the client that is currently registered for it that we (on behalf of the user) want to replace it. Selection provides this mechanism.

To see this in practice, look at Listing 2.3 (code is in Go, some of its constructs are explained in a comment where needed). Firstly, on line 2, we retrieve the owner of the selection we want to manage (`WM_S2`). We then check if another client owns this selection, in which case we should get confirmation from the user that our application could replace it. If there is no owner or we have the user's permission to replace it, we send the `SetSelectionOwner` request, specifying the selection atom (`WM_S2`) and the window to which the selection would be attached (`X.Dummy()` in this case). If the selection was owned by another client, we then have to wait for its termination, that is, wait for the `DestroyNotifyEvent` of the window that owned the selection. After that, we can finally register for `SubstructureRedirect` events on the root window of screen 2, since we now own the selection `WM_S2`. The last thing we have to do to comply with ICCCM is to listen for the `SelectionClearEvent`. We will receive this event when another client sends the `SetSelectionOwner` request (just like we did earlier) and we have to release managed resources (substructure redirection on root window) and destroy the window that owns the selection. In our case, we simply quit, as there is no point for the window manager to run without the substructure redirection.

■ 2.6.2 Clients Actions

Clients should do exactly what they would do in the absence of a window manager, with following exceptions:

- They should hint to the window manager what resources they would like to obtain.
- They should accept the resources they are allocated, even if they are not those requested.
- They should be prepared for resource allocations to change at any time.

■ 2.6.3 Creating a Top-Level Window

Top-Level window is a window whose `override-redirect` attribute is `false`. It must either be a child of a root window, or it must have been a child of a root window immediately prior to having been reparented by the window manager. From the client's point of view, the window manager will regard its top-level window as being in one of three states:

- Normal
- Iconic


```

1 // Get selection owner of the screen we want to manage
2 reply := xproto.GetSelectionOwner(X.Conn(), "WM_S2").Reply()
3
4 if reply.Owner != xproto.WindowNone {
5     // another client owns this selection
6     // if user did not explicitly allow us to replace it,
7     // we should exit here
8 }
9
10 // if there is no selection owner or we want to replace it,
11 // send the set selection owner request for our window (X.Dummy())
12 xproto.SetSelectionOwner(X.Conn(), X.Dummy(), "WM_S2")
13
14 // if another client owned the selection, wait for its termination
15 if reply.Owner != xproto.WindowNone {
16     for { // timeout mechanism is omitted
17         e := X.Conn().PollForEvent()
18         // check that type of e is "DestroyNotifyEvent"
19         if destroyEvent, ok := e.(DestroyNotifyEvent); ok {
20             if destroyEvent.Window == reply.Owner {
21                 break
22             }
23         }
24     }
25 }
26
27 // now we can register for SubstructureRedirect events
28 // on the root window
29 X.RootWin().Listen(EventMaskSubstructureRedirect);
30
31 // when another client sends the set selection owner request,
32 // we will receive the SelectionClearEvent, and by the ICCCM,
33 // we have to release managed resources and destroy the window
34 // that owned the selection - we just quit, which does the job
35 xevent.SelectionClearFun(func (X *XUtil, e SelectionClearEvent) {
36     if e.Selection == "WM_S2" {
37         xevent.Quit(X)
38     }
39 }).Connect(X, X.Dummy())

```

Listing 2.3. Selection acquiring mechanism (code is simplified and would not compile as is)

■ Withdrawn

Newly created windows start in the Withdrawn state. Transitions between states happen when the top-level window is *mapped* and *unmapped* and when the window manager receives certain messages. For historical reasons related to some initial implementations, showing a window in the X11 protocol is called mapping a window, and hiding a window is called unmapping [60]. Transitions between those states will be described in Section 2.6.5.

2.6.4 Client Properties

Client can inform the window manager of the behavior that it desires by placing properties on its top-level windows. Window manager is free to assume values it finds convenient for any properties that are not supplied. The window manager will not change properties written by the client. Contents of these properties are examined by the window manager upon transition from the `Withdrawn` state, some properties are also monitored for changes while the window is in the `Iconic` or `Normal` state. For example, a file manager usually changes `WM_NAME` property when the user navigates to different directories, and the window manager is expected to observe these changes and reflect them in its UI.

In Table 2.2, we can see a summary of all client properties defined by ICCCM. Some of them will be covered in detail.

Name	Type
<code>WM_CLASS</code>	<code>STRING</code>
<code>WM_CLIENT_MACHINE</code>	<code>TEXT</code>
<code>WM_COLORMAP_WINDOWS</code>	<code>WINDOW</code>
<code>WM_HINTS</code>	<code>WM_HINTS</code>
<code>WM_ICON_NAME</code>	<code>TEXT</code>
<code>WM_ICON_SIZE</code>	<code>WM_ICON_SIZE</code>
<code>WM_NAME</code>	<code>TEXT</code>
<code>WM_NORMAL_HINTS</code>	<code>WM_SIZE_HINTS</code>
<code>WM_PROTOCOLS</code>	<code>ATOM</code>
<code>WM_STATE</code>	<code>WM_STATE</code>
<code>WM_TRANSIENT_FOR</code>	<code>WINDOW</code>

Table 2.2. Summary of window manager property types

`WM_CLASS`

The `WM_CLASS` property contains two consecutive null-terminated strings. These specify the Instance and Class names to be used by both the client and the window manager for looking up resources for the application or as an identifying information. For example, in Table 2.1, we have seen a window with `WM_CLASS` set to *“jetbrains-goland, jetbrains-goland”*.

`WM_NAME`, `WM_ICON_NAME`

The `WM_NAME` property is an uninterpreted string that the client wants the window manager to display in association with the window (for example, in a window title bar). Window managers are expected to make an effort to display this information, ignoring `WM_NAME` is not acceptable behavior, unless the user has taken an explicit action to make it invisible. The `WM_ICON_NAME` property is similar, but it is used when the window is in `Iconic` state.

`WM_NORMAL_HINTS`

The `WM_NORMAL_HINTS` property is used by clients to specify the minimum size that the window can be for the client to be useful, as well as the maximum size. Window managers should honor them, even though they do not have to. Clients can also specify

base size and size increments. If they are specified, the window manager should not just resize the window to an arbitrary size, but the size should reflect those values in this way: $size = base_size + i * size_increment$. This is used, for example, by terminal emulators that want to fit an exact number of characters into the window.

WM_PROTOCOLS

The WM_PROTOCOLS property is a list of atoms. Each atom identifies a communication protocol between the client and the window manager in which the client is willing to participate. Atoms can identify both standard protocols and private protocols specific to individual window managers. There are three protocols defined by the ICCCM at the moment:

- WM_TAKE_FOCUS – assignment of input focus.
- WM_DELETE_WINDOW – request to delete top level window.
- WM_SAVE_YOURSELF – request to save client state (deprecated).

WM_STATE

The WM_STATE property is placed on each top-level client window that is not in the Withdrawn state by the window manager. It specifies the state of the window (Withdrawn, Normal, or Iconic), and ID of icon window (if the window is in Iconic state and the window manager is displaying some).

The WM_STATE property is often used as an indicator of a top-level window. For example, some clients (such as xprop [55]) need to find a top-level window under the pointer (user clicking on a window). They can do it by searching the window hierarchy beneath the selected location for a window with the WM_STATE property.

WM_TRANSIENT_FOR

The WM_TRANSIENT_FOR property might contain ID of another top-level window. The implication is that this window is a pop-up on behalf of the named window, and window managers may decide not to decorate transient windows or may treat them differently in other ways. In particular, window managers should present newly mapped WM_TRANSIENT_FOR windows without requiring any user interaction, even if mapping top-level windows normally does require interaction. Dialogs, for example, are an example of windows that should have WM_TRANSIENT_FOR set.

WM_HINTS

The WM_HINTS property is used to communicate information that does not need separate properties to the window manager. For our use case, particularly useful fields would be flags, initial_state, and window_group. Field flags contains boolean value for UrgencyHint, which is set to true when the window needs user attention. Window manager should communicate this to the user – for example, change color of window's border. The value of the initial_state field determines the state the client wishes to be in when the top-level window is firstly mapped. It could be either NormalState (window is visible) or IconicState (icon is visible) – see Section 2.6.3. The window_group field lets the client specify that this window belongs to a group of windows. Window manager can use this hint to manipulate the group as a whole.

2.6.5 Changing Window State

As already mentioned in Section 2.6.3, window manager will assign each top-level window one of three states:

- **Normal** – top-level window is visible.
- **Iconic** – top-level window is iconic. That usually means that top-level window is not visible, but its `icon_window`, `icon_pixmap` or `WM_ICON_NAME` is.
- **Withdrawn** – neither top-level window nor icon is visible.

Newly created top-level windows are in the `Withdrawn` state. Once the window has been provided with suitable properties, the client is free to change its state as follows:

- `Withdrawn` → `Normal` – when window is mapped with `WM_HINTS.initial_state` being `NormalState`.
- `Withdrawn` → `Iconic` – when window is mapped with `WM_HINTS.initial_state` being `IconicState`.
- `Normal` → `Iconic` – client sends `ClientMessage` event.
- `Normal` → `Withdrawn` – client unmaps window and sends a synthetic `UnmapNotify` event.
- `Iconic` → `Normal` – client maps the window, content of `WM_HINTS.initial_state` is irrelevant in this case.
- `Iconic` → `Withdrawn` – client unmaps window and sends a synthetic `UnmapNotify` event.

2.6.6 Configuring the Window

Clients can resize and reposition their top-level windows by using the `ConfigureWindow` request. The attributes of the window that can be altered with this request are as follows:

- The $[x, y]$ location of the window's upper left-outer corner.
- The width and height of the inner region of the window (excluding borders).
- The width of the border of the window.
- The window's position in the stack.

The coordinate system in which the location is expressed is that of the root (irrespective of any reparenting that may have occurred). Client configure requests are interpreted by the window manager in the same manner as the initial window geometry specified by the `WM_NORMAL_HINTS` property. Clients must be aware that there is no guarantee that the window manager will allocate them the requested size or location and must be prepared to deal with any size and location. Window manager can respond to a `ConfigureRequest` request in three different ways:

- Not change the size, location, border width, or stacking order of the window at all. A client will receive a synthetic `ConfigureNotify` event that describes the (unchanged) geometry of the window. The client will not receive a real `ConfigureNotify` event because no change has actually taken place.
- Move or restack the window without resizing it or changing its border width. A client will receive a synthetic `ConfigureNotify` event following the change that describes the new geometry of the window.

- Resize the window or change its border width.

A client will receive a real `ConfigureNotify` event, provided that it has selected for `StructureNotify` events.

2.7 EWMH

Extended Window Manager Hints is the second standard that defines how X Window System clients should interact with one another. It builds on the ICCCM, providing ways to implement many features that modern desktop users expect [16]. It originated as a sets of extensions to the ICCCM developed by the GNOME and KDE desktop projects. Those were eventually unified into a standardized set of ICCCM additions that any desktop environment can adopt. Its latest version 1.5 was released in 2011 [16].

In this section, we will cover parts of EWMH which are related to window managers and therefore important for us. EWMH specification [16] will be constantly referenced here, so we will explicitly refer only other sources if they are used.

2.7.1 Pagers and Taskbars

Throughout this section, we will talk about special X Server Clients. We already know what window manager is, now we will also use term *pager* to refer to desktop utility applications, such as pagers and taskbars.

Pager shows a miniature view of the desktops, representing managed windows by small rectangles and allows the user to initiate various window manager actions by manipulating these representations. Typically offered actions are activation, moving, iconification, maximization and closing. On virtual desktops, the pager may offer ways to move windows between desktops and to change the current desktop.

Taskbar typically represents client windows as a list of buttons labelled with the window titles and possibly icons. Pressing a button initiates a window manager action on the represented window, typical actions being activation and iconification.

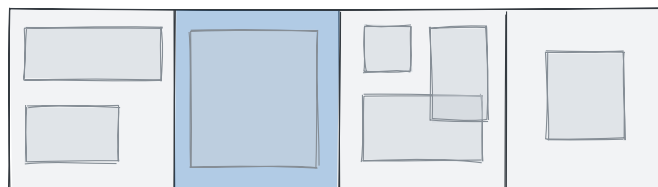


Figure 2.10. Pager displaying four desktops with some windows on each of them



Figure 2.11. Taskbar displaying three windows

■ 2.7.2 Scope of EWMH

EWMH tries to address the following issues:

- Allow clients to influence their initial state with respect to maximization, shading, stickiness, desktop, stacking order.
- Improve the window manager's ability to vary window decorations and maintain the stacking order by allowing clients to hint the window manager about the type of their windows.
- Improve the compositing manager's ability to apply decorations and effects to override-redirect windows.
- Enable pagers and taskbars to be implemented as separate clients and allow them to work with any compliant window manager.

■ 2.7.3 Additional States

The ICCCM allows window managers to implement additional window states, which will appear to clients as substates of `Normal` and `Iconic` states. As a two common examples, we can name *maximized* and *shaded* states. A window manager may implement these additional states as proper substates of `Normal` and `Iconic` states, or it may treat them as independent flags, allowing for example a maximized window to be iconified and to re-appear as maximized upon de-iconification.

- **Maximization** – maximizing should give window as much of the screen area as possible. This may not be the full screen area, but only a smaller work area, because window manager may reserve some space for other windows, such as taskbars. A window manager is expected to remember the geometry of a maximized window and restore it upon demaximization. Modern window managers typically allow separate horizontal and vertical maximization.
- **Shading** – an alternative to iconification. A shaded window typically shows only the titlebar, the client window is hidden, thus shading is not useful for windows which are not decorated with a titlebar.

■ 2.7.4 Large Desktops

The window manager may offer to arrange the managed windows on a desktop that is larger than the root window. The screen functions as a viewport on this large desktop. Different policies regarding the positioning of the viewport on the desktop can be implemented: The window manager may only allow the viewport position to change in increments of the screen size (paging) or it may allow arbitrary positions (scrolling).

To fulfill the ICCCM principle that clients should behave the same regardless whether a window manager is running or not, window managers which implement large desktops must interpret all client-provided geometries with respect to the current viewport.

■ 2.7.5 Virtual Desktops

Most X servers have only a single screen. The window manager may virtualize this resource and offer multiple so-called 'virtual desktops', of which only one can be shown on the screen at a time. There is some variation among the features of virtual desktop

implementations. There may be a fixed number of desktops, or new ones may be created dynamically. The size of the desktops may be fixed or variable.

A window manager which implements virtual desktops generally offers a way for the user to move clients between desktops. Clients may be allowed to occupy more than one desktop simultaneously.

■ 2.7.6 Sticky Windows

A window manager which implements a large or virtual desktops typically offers a way for the user to make certain windows *sticky*. That means that these windows will stay at the same position on the screen when the viewport is moved on large desktop.

■ 2.7.7 Activation

In the X world, activating a window means to give it the input focus. This may not be possible if the window is unmapped, because it is on a different desktop. Thus, activating a window may involve additional steps like moving it to the current desktop (or changing to the desktop the window is on), deiconifying it, or raising it.

■ 2.7.8 Root Window Properties

As stated in Section 2.5.4, clients of X Server are communicating with each other using properties. EWMH defines many new properties. In this section, we will take a look at some of those properties that are set on the root window. Root window properties are useful for communication between the window manager and pagers.

`_NET_SUPPORTED`

This property must be set by the window manager to indicate which hints it supports. If the hint has some states, both the hint and all its supported states must be set – for example `_NET_WM_STATE` and its states, `_NET_WM_STATE_MODAL` and `_NET_WM_STATE_STICKY`.

`_NET_CLIENT_LIST`, `_NET_CLIENT_LIST_STACKING`

These properties contain list of all windows managed by the window manager. While `_NET_CLIENT_LIST` is ordered by initial mapping time, starting with the oldest window, `_NET_CLIENT_LIST_STACKING` has bottom-to-top stacking order. These properties should be set and updated by the window manager.

`_NET_NUMBER_OF_DESKTOPS`

This property should be set and updated by the window manager to indicate the number of virtual desktops. A pager can request a change in the number of desktops, window manager is free to honor or reject this request. If the number of desktops is shrinking, clients that are still present on desktops that are out of the new range must be moved to the very last desktop from the new set and their `_NET_WM_DESKTOP` property must be updated. If the `_NET_CURRENT_DESKTOP` is out of the new range of available desktops, it must be set to the last available desktop from the new set.

`_NET_CURRENT_DESKTOP`

An integer from interval $[0, \text{_NET_NUMBER_OF_DESKTOPS})$ that specifies the index of the current desktop. It must be set and updated by the window manager. A pager can request to switch to another virtual desktop by sending a message to the root window.

`_NET_DESKTOP_NAMES`

The names of all virtual desktops. This is a list of NULL-terminated strings in UTF-8 encoding. It might be changed by a pager or the window manager at any time. Number of names could be different from number of desktops:

- If there is less names than desktops, desktops with high numbers are unnamed.
- If there is more names than desktops, excess names are reserved in case the number of desktops is increased.

`_NET_ACTIVE_WINDOW`

The window ID of the currently active window or None if no window has the focus. This is a read-only property set by the window manager. Client may request to activate another window by sending a message to the root window. The window manager may decide to refuse the request.

`_NET_SUPPORTING_WM_CHECK`

This property must be set by the window manager on the root window to be the ID of a child window created by himself, to indicate that a compliant window manager is active. That child window must also have this property set to the ID of child window. The child window must also have the `_NET_WM_NAME` property set to the name of the window manager.

2.7.9 Other Root Window Messages

Client messages sent to the root window that are not connected with any property (they are stateless).

`_NET_CLOSE_WINDOW`

Message to be sent to the root window by pagers wanting to close a window. Window manager must attempt to closed the specified window. This is preferred to the `WM_DELETE` message or killing the client directly by pager.

`_NET_WM_MOVERESIZE`

This message allows clients to initiate window movement or resizing. They can define their own move and size grips, whilst letting the window manager control the actual operation. This means that all moves/resizes can happen in a consistent manner as defined by the window manager.

2.7.10 Application Window Properties

Application window properties are set on the top-level window either by themselves or by the window manager or pager. Some of them are updated throughout the lifetime of the application, so window managers and pagers should listen for these changes and update accordingly. We will cover those that are important for our window manager.

`_NET_WM_DESKTOP`

Index of the desktop the window is on (or wants to be). It must be an integer from interval $[0, \text{_NET_NUMBER_OF_DESKTOPS})$, or a special value `0xFFFFFFFF`. If it is not specified by the client upon transition from the `Withdrawn` state, window manager should place it as it wishes. Value `0xFFFFFFFF` indicates that the window should appear on all desktops. Client can request a change of desktop for non-withdrawn window by sending a client message to the root window.

`_NET_WM_WINDOW_TYPE`

This should be set by the client before mapping to a list of atoms indicating the functional type of the window. It should be used by the window manager in determining the decoration, stacking position and other behavior of the window. Extensions can define new window types, but each client must include at least one of the basic types defined here. The most important window types are as follows, listed without the `_NET_WM_WINDOW_TYPE_` prefix:

- `NORMAL` – indicates that this is a normal, top-level window, either managed or override-redirect.
- `DESKTOP` – indicates a desktop feature. For example single window containing desktop icons.
- `DOCK` – indicates a dock or panel feature. Those windows would be typically kept on top of all other windows by window manager.

`_NET_WM_STATE`

A list of hints describing the window state. Window manager should honor the state whenever withdrawn window requests to be mapped. Window manager must keep this property updated to reflect the current state of the window. Possible states are as follows, listed without the `_NET_WM_STATE_` prefix:

- `MODAL` – indicates that this is a modal dialog box.
- `STICKY` – indicates that the window manager should keep the window's position fixed on the screen, even when the virtual desktop scrolls.
- `MAXIMIZED_VERT`, `MAXIMIZED_HORZ` – indicates that the window is vertically/horizontally maximized.
- `SHADED` – indicates that the window is shaded.
- `SKIP_TASKBAR` – indicates that the window should not be included on a taskbar.
- `SKIP_PAGER` – indicates that the window should not be included on a pager.
- `HIDDEN` – should be set by the window manager to indicate that a window would not be visible on the screen if its desktop/viewport were active and its coordinates were within the screen bounds. The canonical example is that minimized windows should be in the `HIDDEN` state. Pagers and similar applications should use this state instead of `WM_STATE` to decide whether to display a window in miniature representations of the windows on a desktop.
- `FULLSCREEN` – indicates that the window should fill the entire screen and have no window decorations. Additionally the window manager is responsible for restoring the original geometry after a switch from fullscreen back to normal window.
- `ABOVE`, `BELOW` – indicates that the window should be on top of/below most windows. See Section 2.7.11 for details. They are mainly meant for user preferences and should not be used by applications.

- **DEMANDS_ATTENTION** – indicates that some action in or with the window happened. This state may be set by both the client and the window manager. It should be unset by the window manager when it decides the window got the required attention (usually, that it got activated).
- **FOCUSED** – indicates whether the window’s decorations are drawn in an active state. Clients must regard it as a read-only hint. It cannot be set at map time or changed via a `_NET_WM_STATE` client message. The window given by `_NET_ACTIVE_WINDOW` will usually have this hint.

An implementation may add new atoms to this list. Implementations without extensions must ignore any unknown atoms, effectively removing them from the list. These extension atoms must not start with the prefix `_NET`.

A client can request a change of the state by sending a client message to the root window. This message contains mainly id of the window to which the change should be applied, the action, which is one of `REMOVE`, `ADD`, `TOGGLE`, and one or two properties to alter. It allows two properties to be changed simultaneously, specifically to allow both horizontal and vertical maximization to be altered together.

`_NET_WM_ALLOWED_ACTIONS`

Set by the window manager to indicate which operations it supports for this window. The window manager must keep this property updated to reflect the actions which are currently available for a window. Taskbars, pagers, and other tools use this property to decide which actions should be made available to the user. Possible atoms, listed without the `_NET_WM_ACTION` prefix:

- `MOVE`
- `RESIZE`
- `MINIMIZE`
- `SHADE`
- `STICK`
- `MAXIMIZE_HORZ`
- `MAXIMIZE_VERT`
- `FULLSCREEN`
- `CHANGE_DESKTOP`
- `CLOSE`
- `ABOVE`
- `BELOW`

An implementation may add new atoms to this list. Implementations without extensions must ignore any unknown atoms, effectively removing them from the list. These extension atoms must not start with the prefix `_NET`.

`_NET_WM_STRUT_PARTIAL`, `_NET_WM_STRUT`

Property `_NET_WM_STRUT_PARTIAL` in newer variant of property `_NET_WM_STRUT` and adds some additional parameters. This property must be set by the client if the window is to reserve space at the edge of the screen. The property contains 4 cardinals specifying the width of the reserved area at each border of the screen, and an additional 8 cardinals specifying the beginning and end corresponding to each of the four struts.

The purpose of struts is to reserve space at the borders of the desktop. This is useful for a docking area, a taskbar or a panel, for instance. The window manager should take this reserved area into account when constraining window positions - maximized windows, for example, should not cover that area.

■ 2.7.11 Stacking Order

To obtain good interoperability between different desktop environments, the following layered stacking order is recommended, from the bottom:

- windows of type `_NET_WM_TYPE_DESKTOP`
- windows having state `_NET_WM_STATE_BELOW`
- windows not belonging in any other layer
- windows of type `_NET_WM_TYPE_DOCK` (unless they have state `_NET_WM_STATE_BELOW`) and windows having state `_NET_WM_STATE_ABOVE`
- focused windows having state `_NET_WM_STATE_FULLSCREEN`

Windows that are transient for another window should be kept above this window.

The window manager may choose to put some windows in different stacking positions, for example to allow the user to bring currently active window to the top and return it back when the window loses focus.

Chapter 3

Design

The primary design goal for swm was to be small, easy to use yet hackable stacking window manager. To design such window manager, we based it on one of the core principles of Unix philosophy, that programs should do one thing well, as stated by Doug McIlroy, one of the founders of the Unix tradition, in the Bell System Technical Journal [61]: “Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features.”

To honor this principle, swm does not come with its own panel or application launcher, as some window managers do. Instead, it focuses on that one thing, window management, and the rest is left for third party applications. Thanks to ICCCM and EWMH specifications, third party applications can interact with swm seamlessly and the user can choose any application that fits his needs instead of using the one bundled with the window manager.

For example, there are many open source panel implementations, like polybar [62], lemonbar [63], or pypanel [64], just to name a few. To mention some application launchers that could be used with swm, we can name dmenu [65] or rofi [66].

In this chapter, we will discuss some of the design choices made during the development of swm into detail.

3.1 Tools

In this section, we will describe some tools that can be used to control swm.

3.1.1 Xdotool

Xdotool is a simple command-line tool for communication with the X server. While its main use case is to fake input from the mouse and keyboard, it also supports some parts of EWMH protocol, and thus lets you perform various window manager actions [67]. Actions important for our use-case are:

- `(get|set)_desktop` – get and set the current desktop.
- `(get|set)_num_desktops` – get and set number of desktops.
- `(get|set)_desktop_for_window` – get and set the desktop for window.
- `windowactivate` – activate a window. Sends `_NET_ACTIVE_WINDOW`, window manager should do necessary changes so that specified window could be shown, for example, switch to the desktop the window is on, and then activate it (give it input focus).
- `windowminimize` – minimize a window.

To specify a window on which the action should be performed, one can either directly provide its ID, or use one of those three commands to obtain it:

- `getactivewindow` – get an ID of active window.
- `selectwindow` – get an ID of a window by clicking on it.
- `search` – get IDs of windows whose name or class is matching the search term. Various filters are available, one can, for example, limit the search only to windows located on the specified desktop.

Xdotool can chain commands, so you can use one command to get a window ID first and then perform an action on it. See Listing 3.1 for some examples of xdotool commands.

```

1  xdotool windowactivate 56623105
2  xdotool getactivewindow windowminimize
3  xdotool search "chromium" set_desktop_for_window 3

```

Listing 3.1. Xdotool commands

■ 3.1.2 Wmctrl

Wmctrl is a command-line tool for communication with the X server, similar to xdotool. Unlike xdotool, wmctrl is specialized directly on interaction with EWMH compatible window manager, though. It is a bit more complex than xdotool, because, as stated in its documentation, it provides access to almost all the features defined in the EWMH specification [68]. This means that it can do all the window manager related tasks xdotool can do and much more, but the commands are usually not as straightforward. That is also caused by arguments being named by a single letter, which mostly seems to be chosen by random. We will show two commands which will be important for our use case, the rest could be seen in the documentation [68].

- `wmctrl -c <WIN>` – gracefully close specified window.
- `wmctrl -r <WIN> -b <STATE>` – change the state of the specified window. This command could be used, for example, to make the window maximized, minimized, or fullscreen. It sends the `_NET_WM_STATE` client message as specified in EWMH. The format of the state argument is: `(remove|add|toggle),<PROP1>[,<PROP2>]`, and the following properties are supported:
 - `modal`,
 - `sticky`,
 - `maximized_vert`, `maximized_horz`,
 - `shaded`,
 - `skip_taskbar`, `skip_pager`,
 - `hidden`,
 - `fullscreen`,
 - `above`, `below`.

The window, on which the action should be performed, could be specified in one of these ways:

- By default, window argument is interpreted as a string matched against the window title and the first matching window is used.
- Using the `-i` option, the argument will be interpreted as a numerical window ID represented as a decimal or hexadecimal (prefix `0x`) number.
- Special strings `:ACTIVE:` and `:SELECT:` may be used to use the currently active window or to let the user select the window by clicking on it, respectively.

```

1  wmctrl -c -i 56623105
2  wmctrl -r "spotify" -b toggle,below
3  wmctrl -r :ACTIVE: -b toggle,maximized_vert,maximized_horz

```

Listing 3.2. Wmctrl commands

Wmctrl can be used to, for example, close the window or toggle its maximized state. See Listing 3.2 for some examples of wmctrl commands.

■ 3.1.3 Sxhkd

Sxhkd, standing for *Simple X hotkey daemon*, is an X daemon that reacts to input events by executing commands [45]. You provide it with one or more configuration files, which define the associations between the input events and the commands.

Example configuration file can be seen in Listing 3.3. It demonstrates how sxhkd makes it very easy to map multiple shortcuts to multiple commands at once. It is done using syntax `{_,A,B} + X`, which will define three shortcuts – `X`, `A + X`, and `B + X`. Underscore is representing an empty sequence element – this is useful if we want to have different variants of command based on modifiers used (as in the example, line 7).

```

1  ctrl + alt + {h,j,k,l}
2      swmctl resize -{w,s,n,e} 20
3
4  super + alt + m : {h,j,k,l}
5      swmctl move -{w,s,n,e} 20
6
7  {_,shift + ,super + }XF86MonBrightness{Down,Up}
8      bright {-1,-10,min,+1,+10,max}

```

Listing 3.3. Example configuration file for sxhkd

Sxhkd can execute defined commands both on key press (default behavior) and on key release (using modifier “@”). It has also very interesting feature called *chord chain* – one can specify multiple chords separated by semicolons, the command will then only be executed after receiving each chord of the chain in consecutive order. One can also use colon instead of semicolon to indicate that the chord chain shall not be aborted when the chain tail is reached – see the example, line 4. That means that if we take our example, upon pressing `super + alt + m`, *moving mode* will be activated and we can then move the window just by pressing keys `h`, `j`, `k`, `l`.

■ 3.2 Configuration and Controlling

Most of the window managers are using configuration files with lots and lots of options you can configure. With swm, we decided to take a different approach:

- There is no need for swm to have commands for actions like maximizing or minimizing the window – since it supports EWMH and there are tools for sending EWMH messages (xdotool, wmctrl), we can just make use of them.

- There is no need to handle keyboard shortcuts – users can use `sxhkd` that does it better than we could ever do.

This means that to maximize and minimize the window, as well as to do many more actions, the user can use `sxhkd` in combination with `xdotool` or `wmctrl` and `swm` does not have to be involved at all (apart from implementing the EWMH protocol).

Not everything could be done through the EWMH and third party tools, though. To address those cases, `swm` comes with two tools – `swmctl` and `swmrc` – that will be discussed in the next section.

3.3 Swmctl and Swmrc

`Swmctl` is a simple command-line utility that can be used to send commands to `swm`. It is primarily meant to be used in combination with `sxhkd` to invoke the commands with keyboard shortcuts, but it can also be used directly from the command-line. This makes it easy to test the command first and edit the configuration file only after it works exactly how one wants it to.

All the `swmctl`'s commands will be discussed one by one in Chapter 4 later, but to get an idea of how `swmctl` works and why it is better than simple configuration file, lets have a look at following example.

To move a window using keyboard in `cwm`, one has to bind keyboard shortcuts in its configuration file like this:

```
moveamount 1
bind CM-k window-move-up
bind CMS-k window-move-up-big
```

As you can see, there is an option `moveamount`, which defines how many pixels the window will move. Then, there are two commands for each direction `window-move-X` and `window-move-X-big`. The first one moves the window by `moveamount` pixels, the second one moves it by ten times the `moveamount` pixels.

In `swmctl`, on the other hand, there is a `move` command, that takes arguments for all the directions and amount of pixels to move the window by. User can then bind this command to keyboard shortcut using `sxhkd` like this:

```
ctrl + alt + {h,j,k,l}
swmctl move -{w,s,n,e} 20
```

This approach is both much more simple and versatile than `cwm`'s:

- while `cwm` is limited to two move amounts (and one is even dependant on the other), `swm` can handle arbitrary move amount,
- `swm` can move the window in multiple directions in one command, and it can even be moved by different amount of pixels in each direction.

`Swmrc` is just a shell script that is executed upon startup of `swm`. It is executed after all the initialization has been done, so the window manager is running and can accept any command (`swmctl`'s one, EWMH message sent using `xdotool`), but right before all the existing windows are adopted by `swm`. This makes it ideal for all the configuration commands – one can, for example, set the names of the groups, color of borders, etc.

3.4 Desktops – Groups

Virtual desktops, sometimes called workspaces or window groups, offer a way of organizing applications. Switching to different desktop hides all the applications from the previous desktop and shows applications from a new desktop. This way, the user can group applications that are used together and switch between different tasks easily.

Many slightly different models of virtual desktops exist among window managers. The most standard model is using multiple desktops, one of them being visible at any given time and window always belonging to exactly one of them. In other models, more than one desktop could be visible at a time and a window can belong to more than one group.

Firstly, let's briefly summarize what EWMH tells us about virtual desktops [16]:

- only one desktop can be shown on the screen at a time,
- there may be a fixed number of desktops, or new ones may be created dynamically,
- window manager offers a way for the user to move clients between desktops,
- clients may be allowed to occupy more than one desktop simultaneously.

For more detailed description, see Section 2.7.5.

For swm, we decided to build on cwm's group model and design it in a similar way. There is an unlimited amount of standard groups (cwm has a fixed amount of nine groups), and one special group (called sticky), that is always visible. Every standard group could be either visible or hidden and each window can belong to an arbitrary number of groups. A window is visible if any of its group is visible, otherwise hidden.

Although this group model violates EWMH specification ("only one desktop can be shown on the screen at a time"), it is the most versatile solution and, in a way, a superset of all others. Depending on the implementation (which will be discussed in Section 4.4), the user can always opt to have only one group visible at a time and each window assigned to exactly one group.

Chapter 4

Implementation

In this chapter, we will describe the implementation process of swm. We will go through the tools used for the implementation and testing and describe the implementation and testing process. We will also look at code management, publishing, and continuous integration.

4.1 Tools

In the section, we will go through the tools used in the implementation process – the Go programming language, libraries used to access X11 API, and tools used for testing.

4.1.1 Go

Go programming language [69] in version 1.14 was used for the implementation of both swm and swmctl. Firstly released in 2009, Go is a statically typed, compiled programming language. It was designed at Google by Robert Griesemer, Rob Pike, and Ken Thompson [69]. Go refers to itself as being expressive, concise, clean, and efficient [69].

Go was designed to combine the efficiency and safety of languages like Java or C++ and fluidity of Python. It tries to reduce clutter and complexity. Go has no forward declarations or header files – everything is declared exactly once. Variable types are derived when using the declare-and-initialize construct, so the type has not to be specified explicitly. It is an object-oriented language – it has types and methods and allows an object-oriented style of programming. There is no type hierarchy though, only interfaces. Interfaces are not implemented explicitly – a type automatically satisfies any interface that specifies a subset of its methods.

One of Go's most important features, and feature which puts it apart from other system programming languages (such as C, C++, Rust), is garbage collection. According to authors of Go, managing the lifetimes of allocated objects is one of the biggest sources of bookkeeping in system programs. Manual memory management consumes a significant amount of programmer time and is often the cause of bugs. Go wants to eliminate such programmer overheads by garbage collection. Its introduction to go language was possible thanks to advances in its technology in the last few years prior to Go launch. Go authors are confident that it can be implemented cheaply enough, and with low enough latency, that it could be a viable approach even for networked systems [69].

4.1.2 X Libraries for Go

There are two unofficial libraries for accessing X11 API from Go, XGB [59] and xgbutil [70].

XGB, standing for X Go Binding, is closely modeled after XCB, so it is just a low-level API to communicate with the core X protocol and many of the X extensions (such as ICCCM, EWMH or Xinerama). It claims to be thread safe and according to benchmarks, it gets immediate improvement from parallelism [59].

To compare XGB with XCB, we will use the example of window property lookup. Window property lookup using XCB was shown in Listing 2.2, the same call implemented using XGB can be seen in Listing 4.1. As expected, both versions are very similar, the only difference is that in XGB, we can simply call `Reply` method on the cookie object returned from `GetProperty` function, instead of passing the cookie to another function.

```

1  cookie := xproto.GetProperty(
2      conn, false, win, atom, xproto.GetPropertyTypeAny, 0, 0,
3  )
4  // do something while waiting for the response
5  reply, err := cookie.Reply()

```

Listing 4.1. Window property lookup using XGB

Xgbutil, on the other hand, is higher level utility library working on top of the XGB. Its main goal is to make various X related tasks easier [70]. Those are, for example:

- binding keys,
- using the EWMH or ICCCM specs with the window manager,
- moving and resizing windows,
- assigning function callbacks to particular events, and others.

To get an idea about the design of xgbutil, we can have a look at some functions in Listing 4.2. Functions like `icccm.WmNormalHintsGet` could be used to get specific window property (`WM_NORMAL_HINTS` in this example). Internally, it calls `xprop.GetProperty`, which is a wrapper for `xproto.GetProperty` from XGB. It has no cookie/reply mechanism though, so we are losing the asynchronicity of XCB here. We can, however, always fall back to using XGB for cases in which we need to be asynchronous. To process raw `GetPropertyReply`, which itself is just a byte array with some meta data, xgbutil defines set of functions like `PropValNums()` and `PropValWindows()`. These extract slice of integers or slice of window identifiers, respectively, out of the `GetPropertyReply`. Users of the xgbutil library have the possibility to either use high-level functions like `icccm.WmNormalHintsGet`, or stay with the XGB, optionally utilise those helper functions from xgbutil.

Xgbutil can help us also with event handling, core part of each application interacting with the X server using X protocol. Applications usually deal with X events using so-called event loop, its typical implementation can be seen in Listing 4.3. It is an infinite for loop, which starts by waiting for next X event and then branches based on the type of the event. Application will usually have to handle much more than three events and there will also be another branching for each of them, because, for example, events on root window are handled differently than events on application's top level windows. Because of multiple branching and many possible cases, this will quickly become unclear and difficult to maintain. Xgbutil offers callback mechanism to handle X events. You can simply define functions (callbacks) and specify the event type and window for which that callback will be executed. Whole event loop is then

```

1 // WM_NORMAL_HINTS
2 func WmNormalHintsGet(
3     xu *xgbutil.XUtil, win xproto.Window,
4 ) (*NormalHints, error)
5
6 prop, err := xprop.GetProperty(x, win, atomName)
7
8 func PropValNums(
9     reply *xproto.GetPropertyReply, err error
10 ) ([]uint, error)
11
12 func PropValWindows(
13     reply *xproto.GetPropertyReply, err error
14 ) ([]xproto.Window, error)

```

Listing 4.2. Xgbutil API showcase

```

1 for {
2     xev, err := x.WaitForEvent()
3     if err != nil {
4         // handle error
5         continue
6     }
7     switch e := xev.(type) {
8     case xproto.DestroyNotifyEvent:
9         // handle destroy notify event
10    case xproto.PropertyNotifyEvent:
11        // handle property notify event
12    case xproto.ConfigureRequestEvent:
13        // handle configure request event
14    }
15 }

```

Listing 4.3. X event loop

implemented inside `xgbutil`'s `xevent.Main` function, which calls appropriate callbacks for each event type and window. Example callbacks definition can be seen in Listing 4.4.

Xgbutil also defines `xwindow.Window` structure which is a wrapper around standard X window identifier (unsigned integer). This structure contains several methods for easier manipulation with windows, such as `Move`, `Resize`, `Map`, `Unmap` and much more.

Xgbutil was a great help in implementation of `swm` and its higher-level functions and mechanisms were used whenever it was possible/beneficial. There were a few usecases for which no high-level api was available and in those cases, XGB was used. One example is setting border width of the window, which is not possible using the `xwindow.Window.Configure` method from Xgbutil, so XGB's `xproto.ConfigureWindowChecked` must have been used.

■ 4.1.3 Xephyr

Xephyr is a nested X server that runs as an X application [71]. This is very useful for testing the window manager during the development, because one can make the window

```

1  xevent.DestroyNotifyFun(
2      func(x *xgbutil.XUtil, e xevent.DestroyNotifyEvent) {
3          // handle destroy notify event on window *win*
4      },
5  ).Connect(X, win)
6
7  xevent.PropertyNotifyFun(
8      func(x *xgbutil.XUtil, e xevent.PropertyNotifyEvent) {
9          // handle property notify event on window *win*
10     },
11 ).Connect(X, win)
12
13 xevent.ConfigureRequestFun(
14     func(x *xgbutil.XUtil, e xevent.ConfigureRequestEvent) {
15         // handle configure request event on window *win*
16     },
17 ).Connect(X, win)

```

Listing 4.4. Xgbutil event handling

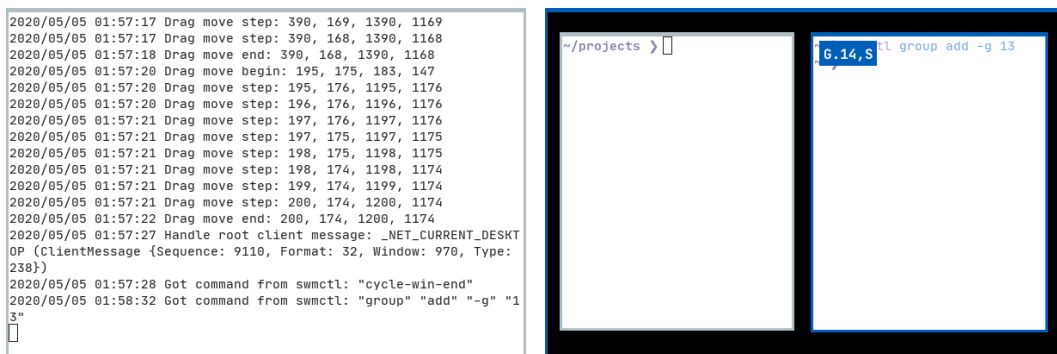


Figure 4.1. Swm running in Xephyr which runs in another instance of swm

manager manage the Xephyrs display and see its debug log in the terminal right next to it. This is depicted in Figure 4.1.

4.1.4 Xvfb

Xvfb, standing for *X virtual framebuffer* is an in-memory display server implementing the X11 display server protocol [72]. Xvfb acts exactly like normal X display server, serving requests and sending events as appropriate, but performing all graphical operations in virtual memory without showing any output on the screen. This makes it ideal for testing X clients, including window managers, on machines with no display hardware and no physical input device.

4.2 Project Structure

The project structure is based on Go modules. Go modules were first introduced in Go version 1.11 and became the default in version 1.13 [73]. It is basically a dependency management system which makes dependency version information explicit and easier to manage [73]. A module is a collection of Go packages stored in a file tree, backed

by `go.mod` file. In Listing 4.5, you can see excerpt of `swm`'s `go.mod` file. It defines the module path, which is also the import path used for the root directory, version of Go, which should be used for compilation, and dependency requirements, which are the other modules needed for a successful build [73].

```

1  module github.com/janbina/swm
2
3  go 1.14
4
5  require (
6      github.com/BurntSushi/xgbutil v0.0.0-20190907113008-ad855c713046
7      //...
8  )

```

Listing 4.5. Go module configuration file

Since there is no official guide on how to structure packages inside Go module, we took inspiration from *Standard Go Project Layout* [74]. They propose a structure with four top-level directories, *cmd*, *internal*, *pkg* and *vendor*. This structure is based on the structure of some popular Go projects, such as Docker [75] or Kubernetes [76], as well as on choices made by Go team in Go's standard library.

- **cmd** directory should contain main applications for the project – in our case, that is *swm* and *swmctl*. Common practice is to have small `main` function which imports and invokes the code from the *internal* and *pkg* directories.
- **internal** directory should contain private application and library code – the code you don't want others importing in their applications or libraries. Since *swm* is not a library and doesn't contain any code which could be reused in other applications, all of its code will reside inside this directory. As of Go version 1.14, Go compiler itself prevents others from importing the code inside this directory [77].
- **pkg** directory should contain code that's meant to be used by other applications. If the project is small, it is not necessary to put packages inside *pkg* directory – unlike *internal*, it does not provide any extra value, so those packages could stay in root directory. It can be, however, useful for large projects or projects with lots of non-Go components.

Complete project structure with all the packages, helper files and directories could be seen in Figure 4.2. All the source code for *swm* and *swmctl* is in directories *cmd* and *internal*. According to the recommendation, `main` functions of both *swm* and *swmctl* are very minimal and they basically only invoke the code from the *internal* directory. Then, there is directory with examples, which contain example configuration file for *sxhkd* and example *swmrc* startup script. They are useful for newcomers, to show them what is possible to do and provide them with some basic usable configuration. Directory *test* contains tests, which are written in Go as well, with an accompanying shell script which does the compilation and prepares the testing environment. Finally, `go.mod` and `go.sum` files are module configuration files used by Go compiler to fetch and verify dependencies.

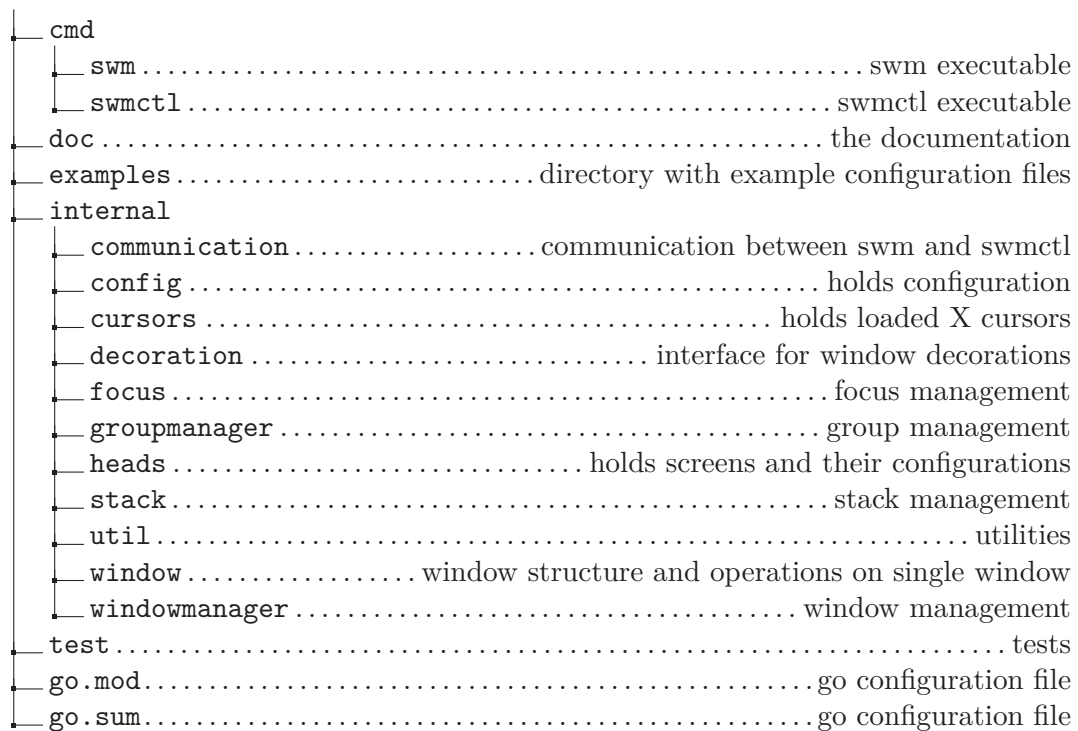


Figure 4.2. Source code structure

4.3 Inter-Process Communication

Communication between `swm` and `swmctl` is done using unix sockets, which is a socket family used to communicate between processes on the same machine efficiently [78], and therefore ideal for our needs. Sockets bound to a filesystem pathname are used. For this purpose, `swm` creates its own directory inside user’s runtime directory defined by an environment variable `XDGRUNTIME_DIR`, which usually points to `/run/user/$USER_ID/`. Each socket is named like “`:$1.$2`”, where `$1` is X display number and `$2` its default screen. The socket address can then look like this:

```
/run/user/1000/swm/:0.0
```

This guarantees that multiple `swm` instances can run on different X screens simultaneously (for example using Xephyr) and they all have different communication channel.

Communication is done by sending null-terminated strings back and forth over the socket. `Swmctl` is designed to be as simple as possible – it only collects its command-line arguments, sends them to the socket and waits for reply. All the heavy lifting is done inside `swm` itself – it listens on the socket for command, parses it, and sends back a reply. Reply is just an arbitrary null-terminated string which is then written to the standard output by `swmctl`.

Updating existing or adding new command then requires changes inside `swm` only, `swmctl` does not have to be modified.

4.4 Desktops – Groups

As described in design section, virtual desktops in swm are replaced with groups. Each window belongs to at least one group. Each group could be either visible or hidden, except for sticky group, which is always visible. In this section, we will discuss groups from the implementation point of view.

Since we tried to stay as EWMH compatible as possible, all the EWMH properties are honored and updated by swm in a way that makes the most sense:

- Root window property `_NET_NUMBER_OF_DESKTOPS` is fully supported – number of desktops corresponds to number of groups and is updated by swm upon change. Requests to change the number of desktops are always honored and results in changing the number of groups. When the change leads to removal of some groups, windows whose only groups is going to be removed are reassigned to a different, valid group.
- Root window property `_NET_CURRENT_DESKTOP` is supported. It is set by swm to ID of the group which is visible and which was made visible most recently. Request to change the current desktop are honored by making the group of corresponding ID the only visible group.
- Root window property `_NET_DESKTOP_NAMES` is fully supported. It is updated by swm to the names user sets using `swmctl` and request to change it are honored and names reported by swm updated accordingly.
- Client property `_NET_WM_DESKTOP` is supported and is set to list of IDs in ascending order of all groups the window is part of. Even though ewmh states that “clients may be allowed to occupy more than one desktop simultaneously” (see Section 2.7.5), it is not very common and most tools treat `_NET_WM_DESKTOP` as a single integer. To address this issue, `swmctl` provides a command to retrieve all the groups the window is part of.

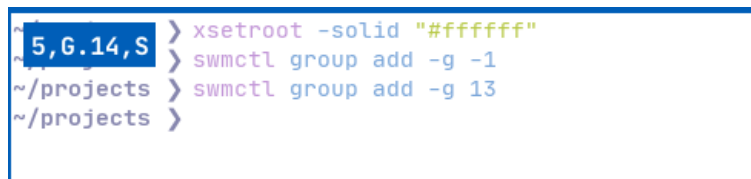
The only feature that cannot be captured by any existing EWMH property is the possibility to have more groups (desktops) visible at the same time. This makes sense, because it goes against the EWMH specification, which says: “only one virtual desktop can be shown on the screen at a time.” To address this, `swmctl` provides a command to obtain all visible group IDs, but also, we created custom root window property called `_SWM_VISIBLE_GROUPS`. It is set by swm to list of IDs (in ascending order) of all groups which are visible. This might be useful for existing tools which already make use of some root window properties – they will not need to incorporate `swmctl`, but instead, they will look just for one more property. For example, `polybar` [62] provides module for EWMH desktops [79]. It has an indication of active desktop, which makes use of `_NET_CURRENT_DESKTOP`. Using `_SWM_VISIBLE_GROUPS` property, it should be quite easy to modify it to indicate not only active desktop, but also all visible groups of swm.

To achieve the functionality described in design section, `swmctl` provides this set of commands:

```
swmctl group mode (sticky|auto)
swmctl group (toggle|show|hide|only) <GROUP-ID>
swmctl group (set|add|remove) [-id <WINDOW-ID>] [-g <GROUP-ID>]
swmctl group names <NAME> [<NAME>...]
swmctl group get [-id <WINDOW-ID>]
swmctl group get-visible
```

- `mode (sticky|auto)` configures the grouping mode. If the group mode is `sticky`, newly created windows are always assigned to sticky group. If the mode is `auto`, `_NET_WM_DESKTOP` client property is used to determine group for window and if it is not set, window is assigned to the group which is visible and was made visible most recently (same logic as `_NET_CURRENT_DESKTOP`).
- `(toggle|show|hide|only) <GROUP-ID>` is there to change the visibility of the group. Option `only` makes only the group with provided ID visible, the rest of groups will be made invisible (except for sticky group).
- `(set|add|remove) [-id <WINDOW-ID>] [-g <GROUP-ID>]` changes the groups the window is part of. You can `set` its group, which will make it belong to specified group only, `add` a group, which will add specified group to all the groups the window is already part of, or `remove` the window from specified group. If you remove the window from its only group, new group will be automatically assigned to it, since every window has to be part of some group. This group is based on grouping mode. Both the group ID and window ID arguments are optional. If not provided, window ID defaults to active window and group ID to current group.
- `names <NAME> [<NAME>...]` sets the group names.
- `get [-id <WINDOW-ID>]` returns list of group IDs the window belongs to. Window ID argument is optional and defaults to active window.
- `get-visible` returns list of group IDs which are in visible state.

Swm also indicates the group membership of window in its UI. Every time the group membership changes, either using `swmctl` command or EWMH client message, small info box in the top left corner of the window is shown for three seconds. This info box lists names of all the groups the window is member of. Example can be seen in Figure 4.3. The window in the example is member of groups named 5, G.14 and S, where S is the name used for the sticky group. This info box is also shown upon execution of `swmctl group get` command.



```

~ 5, G.14, S > xsetroot -solid "#ffffff"
~ /projects > swmctl group add -g -1
~ /projects > swmctl group add -g 13
~ /projects >

```

Figure 4.3. Info box showing group membership

4.5 Becoming a Window Manager

In Section 2.5.2, it was already said that “window manager is a client that has authority over the layout of windows on the screen” and that “certain X protocol features are used only by the window manager to enforce this authority” [53]. In this section, we will look at this mechanism in more detail and discuss how can X client become a window manager.

The first thing swm has to do after the start is to get a connection to the X server, which is the case for all applications that want to communicate with the X server, not only for window managers. After swm connects to the X server, it becomes its regular client. They can already communicate with each other, smw can, for example, send a request to create a window.

To do its job, the window manager needs to intercept requests of other clients to change the state of their top-level windows. This is done using mechanism called *substructure redirection*. Substructure refers to the size, position, and overlapping order of the children of a window. Substructure redirection allows a window manager to intercept any request by an application to change the size, position, border width, or stacking order of its top-level windows on the screen. [56]

This means that to become a window manager, client has to register for substructure redirection on the root window. Xlib programming manual says [56]: “When the window manager selects `SubstructureRedirectMask` on the root window, an attempt by any other client to change the configuration of any child of the root window will fail. Instead an event describing the layout change request will be sent to the window manager. The window manager then reads the event and determines whether to honor the request, modify it, or deny it completely.” For this to work, the X server only allows one running program to register for substructure redirection on any given window at any given time [80]. If there is already a window manager running, attempts to register for substructure redirection on the root window will fail.

After connecting to the X server, swm tries to register for substructure redirection on the root window. If it does not succeed, swm assumes there is already another window manager running, writes that information to the standard output, and quits. There is, however, an option for swm to replace the running window manager. To do that, the user has to run swm with `-replace` flag.

To replace running window manager, swm makes use of mechanism called *selection*, which is part of the ICCCM specification and was discussed in Section 2.6.1. Swm sends the `SetSelectionOwner` request, resulting in running window manager receiving the `SelectionClear` event. It must react to it by releasing all resources it has managed and must then destroy the window that owned the selection. ICCCM specifically says: “For example, a window manager losing ownership of `WM_S2` must deselect from `SubstructureRedirect` on the root window of screen 2 before destroying the window that owned `WM_S2`.” Swm is therefore, after sending the `SetSelectionOwner` request, waiting for `DestroyNotify` event, indicating that previous window manager destroyed its window and then tries to register for substructure redirection again – this time it should succeed. This was also shown in Listing 2.3.

Unfortunately, many window managers do not support this selection mechanism. For example, out of the window managers that we discussed in Section 2.4, only Openbox supports the selection mechanism and quits when another client sends the `SetSelectionOwner` request. Cwm, dwm, bspwm, and i3 do not support it. However, if such a window manager supports at least EWMH, there exists one last possibility for swm to replace it. EWMH requires supporting window managers to set `_NET_SUPPORTING_WM_CHECK` root window property to be the ID of a child window created by themselves, as described in Section 2.7.8. Swm could obtain this ID, forcibly kill that window, and take the substructure redirection for itself.

4.6 Window Decorations and Reparenting

Window manager usually adds its own graphical elements to top-level windows, called window decorations. Window decoration could be anything – from simple one-pixel border to a title bar with window name and buttons to manipulate it. An example

of such a window decoration is shown in Figure 4.4. It depicts a window with a blue border and a title bar with three buttons – those buttons are typically used to minimize, (de)maximize, and close the window. Another common functionality of window decorations is moving and resizing the window using a pointing device – by dragging its borders or title bar.

Window decorations are usually created and managed by the window manager, mainly to unify the looks across all applications. However, some applications opt for custom implementation. In this case, they need to give a hint to the window manager not to draw any other decorations. To implement the moving and resizing functionality, they can use the `_NET_WM_MOVERESIZE` EWMH client message (see Section 2.7.9), minimizing, maximizing, and closing the window is also possible through EWMH protocol. This might be beneficial for applications that need to show some kind of toolbar in their UI anyway, so they include window manipulating buttons and borders as well for better integration. For example, Figure 4.5 shows windows of document viewer Evince and web browser Google Chrome, that come with custom window decorations.

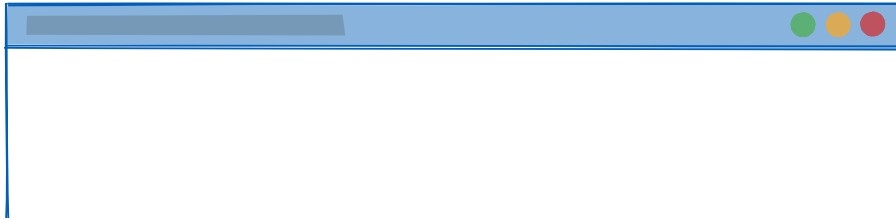


Figure 4.4. Window decorations – borders, title bar, buttons

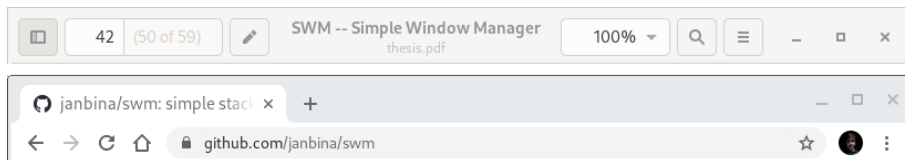


Figure 4.5. Custom window decorations of Evince and Google Chrome

Swm’s window decorations are pretty minimal, but configurable. There is no title bar with buttons, just a simple solid color border around the window and rectangle box indicating group membership – see Figure 4.3. All the colors, the width of the borders, as well as the font and color used for group names, are customizable using following `swmctl`’s commands:

```
swmctl config border 2 B0BEC5 00BCD4 F44336
swmctl config border-top 5 B0BEC5 00BCD4 F44336
// also border-bottom, border-left, border-right

swmctl config font "/usr/share/fonts/TTF/JetBrainsMono-Bold.ttf"

swmctl config info-bg-color 00BCD4
swmctl config info-text-color FFFFFFFF
```

It is possible to configure all the borders at once or each one separately using `config border-top` command and its variants. This means each border could have

different width and color. There are three colors for each border – they are used to indicate one of three possible window states:

- normal,
- active (focused),
- attention (see state `_NET_WM_STATE_DEMANDS_ATTENTION` in Section 2.7.9).

When it comes to window decorations, an important topic is *reparenting*. As mentioned in Section 2.5.5, X window hierarchy is tree-based, and all top-level application windows are created as direct children of so-called root window. Window manager might step into that and, before it maps the application window, create a new window (child of the root), and *reparent* application’s top-level window, using the newly created window as its parent. This is very important for advanced window decorations, like title bars, because they need to be created as separate windows. If we do not do reparenting, we will then have to manipulate all those decoration windows separately, which would be quite difficult. If we, on the other hand, reparent the application’s top-level window, as well as all the decoration windows, we can manipulate just their parent window.

Reparenting is quite complicated, though. As stated by Peter Hofmann, the author of window manager called *katriawm* [81]: “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 does not, 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.”

While developing *swm*, we faced those issues as well. If the window manager is non-reparenting, applications using the standard Java GUI toolkit are rendered as a plain gray boxes instead of rendering the GUI [82]. One solution to this might be to set the name of the window manager to one of those that are hard-coded in the Java GUI toolkit as non-reparenting [82]. This solution was used in early days of development of *swm*. The best solution, though, is to actually reparent the windows.

In the end, we chose to reparent all the windows in *swm*. Not only it did solve the problems with apps using the Java GUI toolkit, but it also made it possible to provide better window decorations that might be easily extended in the future.

4.7 Moving and Resizing

For moving and resizing windows, *swmctl* provides three commands, `move`, `resize` and `moversize`. Their syntax is as follows:

```
swmctl move [-id <WINDOW-ID>]
            [-n <NUM>] [-e <NUM>] [-s <NUM>] [-w <NUM>]

swmctl resize [-id <WINDOW-ID>]
             [-n <NUM>] [-e <NUM>] [-s <NUM>] [-w <NUM>]

swmctl moversize [-id <WINDOW-ID>]
                [-o <ORIGIN>]
                [-x <NUM>] [-y <NUM>] [-w <NUM>] [-h <NUM>]
                [-xr <NUM>] [-yr <NUM>] [-wr <NUM>] [-hr <NUM>]
```


4.9 Window Cycling

Window cycling is another common window manager feature. It is directly tied to keyboard shortcut, usually `alt + Tab` and it provides simple and fast way to switch between recently active windows. Most window managers comes with UI that lists all the windows user can cycle to, highlighting currently chosen window – see Figure 4.6. In minimalistic window managers, this UI is usually missing, though. This is the case for `cwm`, for example.

`Swm` provides very simple window cycling functionality as well. It is controlled via three `swmctl` commands – example `sxhkd` mapping could be seen in Listing 4.6. `Swm` does not come with any window cycling UI, it just temporarily raises particular window to the very top of the stack – even above fullscreen windows, so it can always be seen. User can cycle in both directions, using commands `swmctl cycle-win` and `swmctl cycle-win-rev`. After the cycling operation is done, that is, after `swmctl cycle-win-end` is called, window that was selected is raised permanently, this time respecting its layer though, and given the input focus (activated).

Window cycling functionality can be implemented by third party applications as well. There is, for example, open source application called *alttab*, that describes itself as “X11 window switcher designed for minimalistic window managers or standalone X11 session” [83]. It comes with decent UI and claims that “it’s lightweight and depends only on basic X11 libs, conforming to the usage of lightweight window manager” [83]. It works well with `swm`, the only problem is that it lists windows based on the `_NET_CURRENT_DESKTOP` EWMH property, and thus not showing windows from all visible groups in `swm`, that makes it only semi-usable.

```

1 alt + Tab
2   swmctl cycle-win
3 alt + shift + Tab
4   swmctl cycle-win-rev
5 @Alt_L
6   swmctl cycle-win-end

```

Listing 4.6. Cycling commands usage

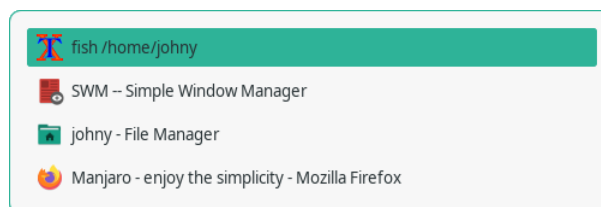


Figure 4.6. Window cycling UI in xfce

4.10 Scriptability

One of the requirements for `swm` was scriptability. Scriptability is possible thanks to `swm`’s compliance with EWMH and tools like `xdotool` and `wmctrl`, and also thanks to `swmctl`. Commands provided by `swmctl`, `xdotool`, and `wmctrl` were showed earlier in

this chapter, now we can have a look at a more complex example. In Listing 4.7, we can see a script that can organize four windows from the current desktop in a grid-like layout – the outcome is depicted in Figure 4.7. Firstly, the script makes use of two `xdotool` commands to get the id of the current desktop and to get a list of windows from that desktop. Then, it goes through those windows, taking maximally four of them, and moving them to different parts of the screen using `swmctl`'s `moveresize` command. The only thing that has to be changed for each window is the origin, which determines the corner to which the window will be moved.

With just a few more lines of code, we can create a script that will organize an arbitrary number of windows, not only four of them. We might also use different layout based on the number of windows

```

1  #!/bin/bash
2
3  # Get up to four windows from the current desktop
4  # and organize them on the screen
5
6  D=$(xdotool get_desktop)
7  I=0
8  ORIGIN=("nw" "ne" "sw" "se")
9
10 xdotool search -desktop $D -class "" | while read ID; do
11     if [[ $I -gt 3 ]]; then exit 0; fi
12     swmctl moveresize -id $ID \
13         -o ${ORIGIN[$I]} \
14         -xr .05 -yr .05 \
15         -wr .425 -hr .425
16     ((I++))
17 done

```

Listing 4.7. Script to organize windows on desktop

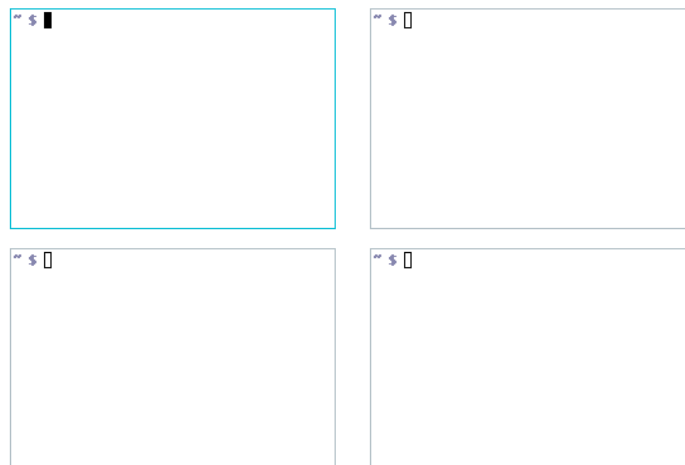


Figure 4.7. Windows in a grid layout produced by the script from Listing 4.7

4.11 ICCCM and EWMH Compliance

One of the goals of swm was to be more ICCCM and EWMH compliant than cwm.

As for ICCCM, swm tries to be fully compliant. It is not easy and straight-forward task though, because ICCCM is known for being ambiguous and difficult to correctly implement [84]. With swm, we tried hard and implemented even selection atoms that are not essential for the window manager to work and are usually left out (see Section 4.5). There are some ICCCM properties that are not used by swm, though. For example, swm does not make any use of `WM_ICON_NAME`, because it does not show anything when the window is in iconified state – that is left for third party task bars and pagers. It does not mean that `WM_ICON_NAME` is not supported, though – it just is not used because of the design and architecture of swm.

As for EWMH, swm tries to be fully compliant as well. Swm groups differ a bit from EWMH’s virtual desktop interpretation, which also affects some desktop related properties defined by EWMH – this was described in Section 4.4. Finally, following EWMH properties are not supported, because they do not make sense for swm for some reason:

- `_NET_DESKTOP_VIEWPORT` – swm desktops (groups) do not have viewports.
- `_NET_VIRTUAL_ROOTS` – according to EWMH specification [16], this property must be set by window managers using technique of virtual roots. Swm does not use this technique and so it does not set this property.
- `_NET_DESKTOP_LAYOUT` – this property is set by pagers according to EWMH specification [16]. Swm does not have any functionality that is dependant on the desktop layout chosen by the user, so it does not inspect this property.
- `_NET_SHOWING_DESKTOP` – desktop showing feature is not implemented in swm.
- `_NET_WM_NAME`, `_NET_WM_VISIBLE_NAME` – swm is not showing window name anywhere in its UI and thus it is not making use of `_NET_WM_NAME` property nor it is setting `_NET_WM_VISIBLE_NAME` property.
- `_NET_WM_ICON_NAME`, `_NET_WM_VISIBLE_ICON_NAME` – same as above.
- `_NET_WM_STATE_SHADED`, `_NET_WM_ACTION_SHADE` – swm does not draw title bars for windows and thus shaded state does not make sense for it.

4.12 Testing

Since nearly everything the window manager does is based on communication with the X server (receiving its events and responding to them), it is impossible to thoroughly test the window manager without the X server running. There are few parts that might be tested in isolation (Unit tested), though. For example, it would be possible to test `stack`, `focus` and `groupmanager` packages this way. They all maintain their internal state (e.g., stacking order), so we might call methods they provide and test that this internal state is updated accordingly. In the end though, we want to test that the window manager reacts to the respective X events and that the internal state is also applied to the X state anyway. Because of that, we went with integration testing to test swm.

Martin Flöser, former maintainer of KDE’s window manager KWin, said in his 2012 article [85] about window manager testing: “Given that we would have to basically

start the full-blown KWin to perform tests which interact with the X-Server, unit tests are out of scope and only integration tests seem feasible.” He also described how could the setup for window manager testing look like [85]: “We basically need a dedicated testing framework which starts a (nested) X Server, starts KWin, performs a test and shuts down both KWin and the X Server. A framework which is decoupled from the running system.”

In this chapter, we will describe how was the testing done for swm, as well as what is tested.

4.12.1 Testing Architecture

In Section 4.1, we described two tools that can run nested X Server, Xephyr (4.1.3), and Xvfb (4.1.4). Both of them would be usable for testing purposes, Xvfb is much better though – we do not need any graphical output for testing, and Xvfb does just that. This allows us to run tests even on machines with no display hardware, like test servers.

Tests are, as well as swm itself, written in Go, and same helper libraries were used – XGB and xgbutil (4.1.2). To run the tests, there is single shell script that could be seen in Listing 4.8. It compiles swm, swmctl and test app, starts Xvfb with swm and runs the test application on it.

The testing application runs various tests, outputting the test name and whether it finished correctly or with some errors, in which case those errors are printed to the output as well. Example output could be seen in Listing 4.9.

```

1 # build swm, swmctl and test app
2 go build github.com/janbina/swm/cmd/swm
3 go build github.com/janbina/swm/cmd/swmctl
4 go build github.com/janbina/swm/test
5
6 # start xvfb
7 Xvfb :111 -screen 0 1024x768x16 &
8 XVFB_PID=$!
9
10 # start swm, discard output logs
11 ./swm > /dev/null 2>&1 &
12
13 # run test and save the exit code
14 ./test; EXIT_CODE=$?
15
16 # shut down nested X server and cleanup
17 kill -15 $XVFB_PID
18 rm swm swmctl test
19
20 exit $EXIT_CODE

```

Listing 4.8. Testing script


```

1  Testing cycling ... OK
2  Testing desktop names ... OK
3  Testing group basics ... OK
4  Testing group window creation ... OK
5  Testing group window movement ... OK
6  Testing group visibility ...
7      error: groups.go:191: Window should be mapped
8      error: groups.go:198: Window should be mapped
9  Testing group membership ... OK
10 Testing moving command ... OK
11 Testing resizing command ... OK
12 Testing moveresize command ... OK
13 Testing window states ... OK
14 Total number of errors: 2

```

Listing 4.9. Test output

■ 4.12.2 Testing Process

One particular problem in testing is the asynchronicity of X. Because of it, it is not possible to send a request and check if it succeeded right after. For example, after sending a request to change a window state, we need to wait before checking that it was really changed. One option to do that would be to actually wait (i.e., sleep) before checking. This has shown to be a possible, but not an ideal solution – one has to choose the appropriate sleep time, long enough to be sure that the changes already took affect, but not very long at the same time. This leads to the state that the tests are both slow and can fail randomly at the same time. A better solution is to wait for an appropriate X event, which is not as easy but is failproof and much faster.

Example test case is shown in Listing 4.10. It tests whether window size changes correctly after sending a request to make it horizontally maximized. Firstly, a dummy window is created, and both the window geometry and the root window geometry are retrieved. After that, a request to add `_NET_WM_STATE_MAXIMIZED_HORZ` state to the window is sent. Then, the test application waits for `ConfigureNotify` event, which is issued upon changing window size. After that, window geometry (supposedly changed) is retrieved again and compared to what is believed to be the correct geometry in the horizontally maximized state – window height and y coordinate stay the same, while the x coordinate and width are those of root window.

■ 4.12.3 Test Coverage

Implemented tests cover the core functionality of swm, as well as the communication between swm and swmctl, and correct reaction to various X events, mainly those defined by EWMH. Tests are separated into eleven functions, their brief description follows:

- Cycling – tests that the window cycling works correctly. Issues swmctl commands to cycle windows back and forth and test that correct window gets activated.
- Desktop names – tests that desktop (group) names are set correctly (by EWMH conventions, see Section 2.7.8) by the swmctl command.
- Group basics – tests basic group manipulations using EWMH properties. That is, that the number of groups could be changed by requests to change root window

```

1 win := createWindow()
2 initGeom := win.Geometry()
3 screenGeom := X.RootWin().Geometry()
4
5 _ = ewmh.WmStateReqExtra(
6     X, win.Id, ewmh.StateAdd,
7     "_NET_WM_STATE_MAXIMIZED_HORZ", "", 2,
8 )
9 waitForConfigureNotify()
10 newGeom = win.Geometry()
11 assertGeomEquals(
12     xrect.New(
13         screenGeom.X(), initGeom.Y(),
14         screenGeom.Width(), initGeom.Height(),
15     ),
16     newGeom,
17     "invalid geometry",
18 )

```

Listing 4.10. Test case example

property `_NET_NUMBER_OF_DESKTOPS`, and that current group could be changed by requests to change `_NET_CURRENT_DESKTOP` root window property.

- Group window creation – test two swm’s group modes, sticky and auto. Changes current group mode using `swmctl` command, creates some windows, and tests that they were assigned the correct group.
- Group window movement – tests that windows could be moved to different group by requests to change their `_NET_WM_DESKTOP` property. Also tests that if some groups are removed, windows from those groups are moved to another group.
- Group visibility – makes different groups visible/hidden by issuing `swmctl` commands and tests that windows are appropriately mapped/unmapped based on their group membership.
- Group membership – tests that one window could be member of multiple groups. Adds/removes window from different groups and tests that correct groups are reported by `swmctl` command.
- Moving command – tests `swmctl` move command. Issues the command with different set of arguments and tests that window geometry changed appropriately.
- Resizing command – tests `swmctl` resize command. Issues the command with different set of arguments and tests that window geometry changed appropriately.
- Move-resize command – tests `swmctl` moveresize command. Issues the command with different set of arguments and tests that window geometry changed appropriately.
- Window states – tests that windows react properly to requests to change their `_NET_WM_STATE` EWMH property (see Section 2.7.10). For example, making the window maximized, fullscreen, hidden, focused, etc.

4.13 Code Management and Continuous Integration

The source code is managed using Git¹ version control system (VCS) and is published on GitHub² under the MIT license [86]. GitHub is the biggest and most important host of open-source code in the world and is used by companies like Google, Facebook, or Twitter [87].

4.13.1 Code Style

To maintain consistent formatting and code style across the project, we make use of some officially provided Go Tools [88], namely `gofmt`, `goimports`, and `govet`:

- `gofmt` is a tool that automatically formats Go source code. It takes care of indentation, brace positions, and more.
- `goimports` updates the imports, adding missing ones, and removing unreferenced ones. It also groups the imports into two groups – packages from the standard library and third-party packages – and sorts them alphabetically within those groups. In addition to fixing imports, `goimports` also formats code in the same style as `gofmt`.
- `govet` examines source code and reports suspicious constructs. For example, useless assignments, unreachable code, or unused results of calls to some functions.

A short example of a well-formed Go code could be seen in Listing 4.11. Notice how the imports are split into two groups and sorted alphabetically, and how type names are aligned inside the structure declaration. Note that `gofmt` uses tabs for indentation. While tabs are usually displayed as 8 characters wide, we opted for 4 characters wide tabs in listings in this thesis to save some horizontal space.

Although most IDEs that support Go already include those tools and format the code automatically upon file save, we want to be sure that no badly formatted code makes its way into our codebase. To ensure that, we can make use of mechanism provided by Git VCS – Git hooks. Git hooks are scripts that Git executes before or after events such as commit or push [89]. We defined a pre-commit Git hook that runs `goimports` on all files changed since the last commit and `govet` on the whole project (`govet` has to be run on the whole project, otherwise it would report, for example, an undeclared name that is declared in a different file). If they find any issues, they prevent the user from committing those changes until they are fixed. Since `goimports` is also able to automatically fix all the issues it finds, the user just needs to verify the changes and add them to the commit. Issues reported by `govet` have to be fixed manually.

4.13.2 Continuous Integration

“Continuous Integration is a software development practice where members of a team integrate their work frequently. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.” [90]

There are many solutions and tools for CI. For our project, we used GitHub Actions [91]. GitHub Actions are part of GitHub where the code is hosted, so it was very easy to set them up, and they are free for open-source projects. We can use them to run arbitrary scripts on Linux, macOS, and Windows machines. Actions can be configured to run

¹ <https://git-scm.com/>

² <https://github.com/janbina/swm>

```

1 package window
2
3 import (
4     "log"
5     "time"
6
7     "github.com/BurntSushi/xgb/xproto"
8     "github.com/BurntSushi/xgbutil"
9     "github.com/janbina/swm/internal/config"
10    "github.com/janbina/swm/internal/decoration"
11 )
12
13 type Window struct {
14     win          *xwindow.Window
15     infoTimer    *time.Timer
16     decorations  decoration.Decorations
17     moveState    *MoveState
18 }

```

Listing 4.11. Example of a well-formed Go code

when specific activity on GitHub happens, at a scheduled time, or when an event outside of GitHub occurs [91]. For example, the most typical use-case is to run tests for all pull requests.

In our project, we defined two actions that run on each push to the master branch, as well as for all pull requests to the master branch. The first action builds both swm and swmctl, and runs tests. The second action does the static analysis – it runs `goimports` and `govet` in the same fashion we described in the previous section. If anything fails, it is not possible to merge the pull request. Figure 4.8 shows an example of a pull request with failing checks, which prevents it from being merged. Figure 4.9 shows an example of a pull request which passed all the checks and was already merged.


2 failing and 1 successful checks [Hide all checks](#)

- ✗ **Static check / Format (pull_request)** Failing after 56s **Required** [Details](#)
- ✓ **Build and test / Build and test (pull_request)** Successful in 44s **Required** [Details](#)
- ✗ **Static check / Vet (pull_request)** Failing after 55s **Required** [Details](#)

Required statuses must pass before merging
All required [statuses](#) and check runs on this pull request must run successfully to enable automatic merging.

Merge pull request or view [command line instructions](#).

Figure 4.8. Failing CI checks preventing merge

 **janbina** merged commit **c67a558** into **master** 4 days ago [Hide details](#) [Revert](#)

3 checks passed	
✓ Build and test	Details
✓ Format	Details
✓ Vet	Details

Figure 4.9. Passed CI checks

Chapter 5

Conclusion

In this thesis, we described the working mechanisms of the X Window System Protocol and two standard protocols defined on top of it, ICCCM and EWMH. Those mechanisms and policies were described especially concerning window managers, special X Server clients that control the placement and appearance of windows on the screen. We also described what a window manager is, its different types, and features it usually offers, demonstrated on some existing implementations. Finally, a new stacking window manager called swm was designed in such a way that it is small, extensible, and scriptable. It was implemented in Go, a modern system programming language. It is ICCCM and EWMH compliant, and its core features include stacking window management, advanced window grouping mechanism, and basic window decorations. It is extensible and scriptable and comes with a command-line utility that makes it easy to manipulate windows in a simple but powerful way. We also described how a window manager could be tested and how tests for core features of swm were implemented.

Future Work

Swm is an open-source project, its source code is published on GitHub¹ under the MIT license. It was designed and implemented in a way to be built upon, to be extended with new functionalities, and adapted to various use cases and user preferences. Therefore, there is a potential for it to be extended by third parties, either in a way of contributions or by starting a new project using swm as a starting point.

¹ <https://github.com/janbina/swm>

References

- [1] Nathan Lineback. *Graphical User Interface Timeline*. [online]. 2012.
<http://toastytech.com/guis/guitimeline.html>. [cit. 2020-05-26].
- [2] Steven Mikes. *X Window System Program Design and Development*. Addison-Wesley Publishing Company, 1992. ISBN 0201550776.
- [3] Tom LaStrange. *twm – Tab Window Manager for the X Window System*. [online]. 1987.
<https://www.x.org/releases/X11R7.6/doc/man/man1/twm.1.xhtml>. [cit. 2020-05-26].
- [4] The Open Group. *Common Desktop Environment*. [online]. 1993.
<http://www.opengroup.org/cde/>. [cit. 2020-05-28].
- [5] Bryan Lunduke. *A visual history of OS desktop environments*. [online]. 2014.
<https://www.networkworld.com/article/2359355/156246-A-visual-history-of-OS-desktop-environments.html>. [cit. 2020-05-28].
- [6] Liam Proven. *Party like it's 1999: CDE Unix desktop REBORN*. [online]. 2012.
https://www.theregister.co.uk/2012/08/09/cde_goes_opensource/. [cit. 2020-05-28].
- [7] KDE. *KDE*. [online]. 1998.
<https://kde.org/plasma-desktop>. [cit. 2020-04-29].
- [8] Matthias Ettrich. *New Project: Kool Desktop Environment (KDE)*. [online]. 1996.
https://groups.google.com/forum/#!msg/de.comp.os.linux.misc/SDbiV3Iat_s/zv_D_2ctS8sJ. [cit. 2020-05-26].
- [9] Jim Hall. *How the Linux desktop has grown*. [online]. 2019.
<https://opensource.com/article/19/8/how-linux-desktop-grown>. [cit. 2020-05-28].
- [10] Xfce Development Team. *Xfce*. [online]. 1996.
<https://www.xfce.org/>. [cit. 2020-04-29].
- [11] Carsten Haitzler. *Enlightenment*. [online]. 1997.
<https://www.enlightenment.org/>. [cit. 2020-05-28].
- [12] The GNOME Project. *GNOME*. [online]. 1999.
<https://www.gnome.org/>. [cit. 2020-04-29].
- [13] Arch Wiki. *Desktop environment*. [online]. 2020.
https://wiki.archlinux.org/index.php/Desktop_environment. [cit. 2020-04-29].
- [14] suckless.org. *Dwm*. [online]. 2006.
<https://dwm.suckless.org/>. [cit. 2020-05-13].
- [15] David Rosenthal, and Stuart W. Marks. *Inter-Client Communication Conventions Manual*. [online]. 1994.
<https://www.x.org/releases/X11R7.6/doc/xorg-docs/specs/ICCCM/icccm.html>. [cit. 2020-05-13].

- [16] X Desktop Group. *Extended Window Manager Hints*. [online]. 2011.
<https://specifications.freedesktop.org/wm-spec/wm-spec-1.5.html>. [cit. 2020-05-13].
- [17] Freedesktop.org. *Freedesktop.org*. [online]. 2020.
<https://www.freedesktop.org/wiki/>. [cit. 2020-05-28].
- [18] The Linux Information Project. *Windowing System Definition*. [online]. 2005.
http://www.linfo.org/windowing_system.html. [cit. 2020-05-12].
- [19] STANDS4 LLC Definitions.net. *Windowing system*. [online]. 2020.
<https://www.definitions.net/definition/windowing+system>. [cit. 2020-05-12].
- [20] Arch Wiki. *Xorg*. [online]. 2020.
<https://wiki.archlinux.org/index.php/Xorg>. [cit. 2020-05-12].
- [21] The X.Org Foundation. *X.Org*. [online]. 2019.
<https://www.x.org/wiki/>. [cit. 2020-05-12].
- [22] X.Org Foundation. *XOrgFoundation – X.Org Foundation information*. [online].
<https://www.x.org/releases/current/doc/man/man7/XOrgFoundation.7.xhtml>. [cit. 2020-04-19].
- [23] freedesktop.org, and others. *Wayland*. [online]. 2008.
<https://wayland.freedesktop.org/>. [cit. 2020-05-12].
- [24] Jack Wallen. *Linux’s X.org server is vulnerable. Here’s how to stay safe*. [online]. 2017.
<https://www.techrepublic.com/article/a-lesser-known-reason-to-migrate-from-x-org/>. [cit. 2020-05-23].
- [25] freedesktop.org, and others. *Wayland Architecture*. [online]. 2008.
<https://wayland.freedesktop.org/architecture.html>. [cit. 2020-05-12].
- [26] Samuel Walladge. *Are we Wayland yet?* [online]. 2019.
<https://www.swalladge.net/archives/2019/10/14/are-we-wayland-yet/>. [cit. 2020-05-23].
- [27] Robot Guy. *X11 Sucks... So What’s Up With Wayland?* [online]. 2017.
<https://guyrobottv.wordpress.com/2017/04/05/x11-sucks-so-whats-up-with-wayland/>. [cit. 2020-05-23].
- [28] Shivam Singh Sengar. *Wayland v/s Xorg : How Are They Similar & How Are They Different*. [online]. 2018.
<https://www.secjuice.com/wayland-vs-xorg/>. [cit. 2020-05-23].
- [29] Wayne E. Carlson. *Computer Graphics and Computer Animation: A Retrospective Overview*. [online]. 2017.
<https://ohiostate.pressbooks.pub/graphicshistory/>. [cit. 2020-04-29].
- [30] David C. Smith, Charles Irby, Ralph Kimball, and Bill Verplank. Designing the Star User Interface. *Byte*. 1982, 242–282.
- [31] Arch Wiki. *Window manager*. [online]. 2020.
https://wiki.archlinux.org/index.php/window_manager. [cit. 2020-04-29].
- [32] Marius Aamodt Eriksen. *cwm*. [online]. 2004.
<https://man.openbsd.org/cwm.1>. [cit. 2020-05-5].
- [33] Rodolfo Gouveia. *Getting started with cwm*. [online]. 2009.
<https://undeadly.org/cgi?action=article;sid=20090502141551>. [cit. 2020-05-5].

-
- [34] ddc. *Introduction: calm window manager*. [online]. 2011.
<https://www.osnews.com/story/25359/introduction-calm-window-manager/>. [cit. 2020-05-5].
- [35] Dana Jansens, and Mikael Magnusson. *Openbox*. [online]. 2002.
http://openbox.org/wiki/Main_Page. [cit. 2020-05-11].
- [36] The LXDE Team. *LXDE*. [online]. 2006.
https://wiki.lxde.org/en/Main_Page. [cit. 2020-05-11].
- [37] LXQt. *LXQt*. [online]. 2013.
<https://lxqt.github.io/>. [cit. 2020-05-11].
- [38] Lubuntu Community. *Lubuntu*. [online]. 2011.
<https://lubuntu.me/>. [cit. 2020-05-11].
- [39] Manjaro GmbH & Co. KG. *Manjaro*. [online]. 2011.
<https://manjaro.org/>. [cit. 2020-05-11].
- [40] Arch Wiki. *Openbox*. [online]. 2020.
<https://wiki.archlinux.org/index.php/openbox>. [cit. 2020-05-12].
- [41] suckless.org. *Philosophy*. [online]. 2015.
<https://suckless.org/philosophy/>. [cit. 2020-05-13].
- [42] suckless.org. *Dwm patches*. [online]. 2006.
<https://dwm.suckless.org/patches/>. [cit. 2020-05-13].
- [43] Bastien Dejean. *Bspwm*. [online]. 2012.
<https://github.com/baskerville/bspwm>. [cit. 2020-05-12].
- [44] Arch Wiki. *Bspwm*. [online]. 2020.
<https://wiki.archlinux.org/index.php/Bspwm>. [cit. 2020-05-12].
- [45] Bastien Dejean. *Simple X hotkey daemon*. [online]. 2013.
<https://github.com/baskerville/sxhkd>. [cit. 2020-04-29].
- [46] Michael Stapelberg. *i3*. [online]. 2009.
<https://i3wm.org/>. [cit. 2020-05-6].
- [47] Julien Danjou. *awesome*. [online]. 2007.
<https://awesomewm.org/>. [cit. 2020-05-27].
- [48] Jason Creighton Spencer Janssen, Don Stewart. *xmonad*. [online]. 2007.
<https://xmonad.org/>. [cit. 2020-05-27].
- [49] Thorsten Wissmann. *herbstluftwm*. [online]. 2011.
<https://herbstluftwm.org/>. [cit. 2020-05-27].
- [50] Aldo Cortesi. *Qtile*. [online]. 2008.
<http://www.qtile.org/>. [cit. 2020-05-27].
- [51] X.Org Foundation. *X - a portable, network-transparent window system*. [online].
<https://www.x.org/releases/current/doc/man/man7/X.7.xhtml>. [cit. 2020-04-19].
- [52] X.Org Foundation. *Releases*. [online].
<https://www.x.org/wiki/Releases/>. [cit. 2020-04-19].
- [53] Adrian Nye. *X Protocol reference manual for version 11 of the X window system, Vol. 0 (Definitive Guides to the X Window System)*. Third edition. O'Reilly Media, 1992. ISBN 0937175501.
- [54] The Open Group. *X Window System Protocol*. [online]. 2004.
<https://www.x.org/releases/X11R7.7/doc/xproto/x11protocol.html>. [cit. 2020-05-28].

- [55] Mark Lillibridge. *xprop*. [online]. 2019.
<https://www.x.org/releases/X11R7.5/doc/man/man1/xprop.1.html>. [cit. 2020-05-22].
- [56] Adrian Nye. *Xlib Programming Manual for Version 11, Rel. 5, Vol. 1 (Definitive Guides to the X Window System)*. Third edition. O'Reilly Media, 1994. ISBN 1565920023.
- [57] The Open Group. *Xlib - C Language X Interface*. [online]. 2002.
<https://www.x.org/releases/current/doc/libX11/libX11/libX11.html>. [cit. 2020-05-28].
- [58] X.Org Foundation. *The X New Developer's Guide: Xlib and XCB*. [online]. 2013.
<https://www.x.org/wiki/guide/xlib-and-xcb/>. [cit. 2020-04-20].
- [59] Andrew Gallant. *X Go Binding*. [online]. 2012.
<https://github.com/BurntSushi/xgb>. [cit. 2020-05-13].
- [60] Jasper St. Pierre, and others. *X Window System Basics*. [online].
<https://magcius.github.io/xplain/article/x-basics.html>. [cit. 2020-05-13].
- [61] B. A. Tague M. D. McIlroy, E. N. Pinson. *UNIX Time-Sharing System: Forward*. 1978.
- [62] Michael Carlberg. *Polybar*. [online]. 2016.
<https://github.com/polybar/polybar>. [cit. 2020-05-13].
- [63] The Lemon Man. *Lemonbar*. [online]. 2012.
<https://github.com/LemonBoy/bar>. [cit. 2020-05-13].
- [64] Jon Gelo. *PyPanel*. [online]. 2003.
<https://github.com/jgelo/pypanel>. [cit. 2020-05-13].
- [65] suckless.org. *Dmenu*. [online]. 2006.
<https://tools.suckless.org/dmenu/>. [cit. 2020-05-10].
- [66] Sean Pringle. *Rofi*. [online]. 2012.
<https://github.com/davatorium/rofi>. [cit. 2020-05-10].
- [67] Jordan Sissel. *xdotool*. [online]. 2007.
<https://github.com/jordansissel/xdotool>. [cit. 2020-04-27].
- [68] Tomáš Stýblo. *wmctrl*. [online]. 2003.
<http://tripie.sweb.cz/utis/wmctrl/>. [cit. 2020-04-27].
- [69] The Go Authors. *Go*. [online]. 2009.
<https://golang.org/>. [cit. 2020-04-27].
- [70] Andrew Gallant. *XGBUtil*. [online]. 2012.
<https://github.com/BurntSushi/xgbutil>. [cit. 2020-05-13].
- [71] Matthew Allum. *Xephyr*. [online]. 2004.
<https://www.freedesktop.org/wiki/Software/Xephyr/>. [cit. 2020-04-27].
- [72] David P. Wiggins. *Xvfb*. [online].
<https://www.x.org/releases/X11R7.7/doc/man/man1/Xvfb.1.xhtml>. [cit. 2020-04-27].
- [73] Tyler Bui-Palsulich, and Eno Compton. *Using Go Modules*. [online]. 2019.
<https://blog.golang.org/using-go-modules>. [cit. 2020-05-4].
- [74] Kyle Quest. *Standard Go Project Layout*. [online]. 2017.
<https://github.com/golang-standards/project-layout>. [cit. 2020-05-4].
- [75] Docker Inc. *Docker*. [online]. 2013.
<https://github.com/docker/cli>. [cit. 2020-05-4].

-
- [76] The Kubernetes Authors. *Kubernetes*. [online]. 2014.
<https://github.com/kubernetes/kubernetes>. [cit. 2020-05-4].
- [77] The Go Authors. *Go 1.4 Release Notes*. [online]. 2020.
<https://golang.org/doc/go1.4>. [cit. 2020-05-4].
- [78] The Linux man-pages project. *Linux Programmer's Manual (unix-sockets for local interprocess communication)*. [online]. 2020.
<http://man7.org/linux/man-pages/man7/unix.7.html>. [cit. 2020-05-6].
- [79] Michael Carlberg. *Module: xworkspaces*. [online]. 2016.
<https://github.com/polybar/polybar/wiki/Module:-xworkspaces>. [cit. 2020-05-6].
- [80] Chuan Ji. *How X Window Managers Work, And How To Write One (Part I)*. [online]. 2014.
<https://jichu4n.com/posts/how-x-window-managers-work-and-how-to-write-one-part-i/>. [cit. 2020-05-7].
- [81] Peter Hofmann. *katriawm: The adventure of writing your own window manager*. [online]. 2016.
<https://www.uninformativ.de/blog/postings/2016-01-05/0/POSTING-en.html>. [cit. 2020-05-9].
- [82] Arch Wiki. *Java*. [online]. 2020.
<https://wiki.archlinux.org/index.php/Java>. [cit. 2020-05-10].
- [83] Alexander Kulak. *Alttab*. [online]. 2016.
<https://github.com/sagb/alttab>. [cit. 2020-05-11].
- [84] Conrad Parker. *[chat] Re: [SLUG] Ximian / Gnome and Xalf*. [online]. 2001.
<https://raw.githubusercontent.com/kfish/xsel/1a1c5edf0dc129055f7764c666da2dd468df6016/rant.txt>. [cit. 2020-05-11].
- [85] Martin Flöser. *Unit Testing a Window Manager*. [online]. 2012.
<https://blog.martin-graesslin.com/blog/2012/06/unit-testing-a-window-manager/>. [cit. 2020-05-10].
- [86] Open Source Initiative. *The MIT License*. [online]. 1987.
<https://opensource.org/licenses/MIT>. [cit. 2020-05-24].
- [87] Klint Finley. *For Open Source, It's All About GitHub Now*. [online]. 2019.
<https://www.wired.com/story/open-source-all-about-github-now/>. [cit. 2020-05-25].
- [88] The Go Authors. *Go Tools*. [online]. 2012.
<https://github.com/golang/tools>. [cit. 2020-05-24].
- [89] Matthew Hudson. *Git Hooks*. [online]. 2012.
<https://githooks.com/>. [cit. 2020-05-24].
- [90] Martin Fowler. *Continuous Integration*. [online]. 2006.
<https://martinfowler.com/articles/continuousIntegration.html>. [cit. 2020-05-24].
- [91] GitHub Inc. *GitHub Actions*. [online]. 2019.
<https://github.com/features/actions>. [cit. 2020-05-24].



Appendix A

Acronyms

- API ■ Application programming interface
- CI ■ Continuous Integration
- EWMH ■ Extended Window Manager Hints
- GUI ■ Graphical User Interface
- ICCCM ■ Inter-Client Communication Conventions Manual
- VCS ■ Version Control System
- XGB ■ X Go Binding

Appendix B

Contents of enclosed SD card

```
/
├── src ..... the directory of source codes
│   ├── swm ..... the directory of Go source codes of the implementation
│   └── thesis ..... the directory of plainTeX source codes of the thesis
├── DP_Bina_Jan_2020.pdf ..... the thesis text in PDF format
├── readme.txt ..... the file with SD card contents description
└── showcase.mp4 ..... the video with a showcase of swm
```