OxideWM

Release 0.1

OxideWM contributors

Sep 13, 2023

CONTENTS

1 Contents **3**

1.1 Introducing *Oxide* 3

1.2 Project procedure 5

1.3 Diagrams 25

1.4 Using *Oxide* 45

Oxide is a tiling window manager for X11. It is written in Rust and uses X11rb. This project idea is inspired by DWM, leftWM and i3WM.

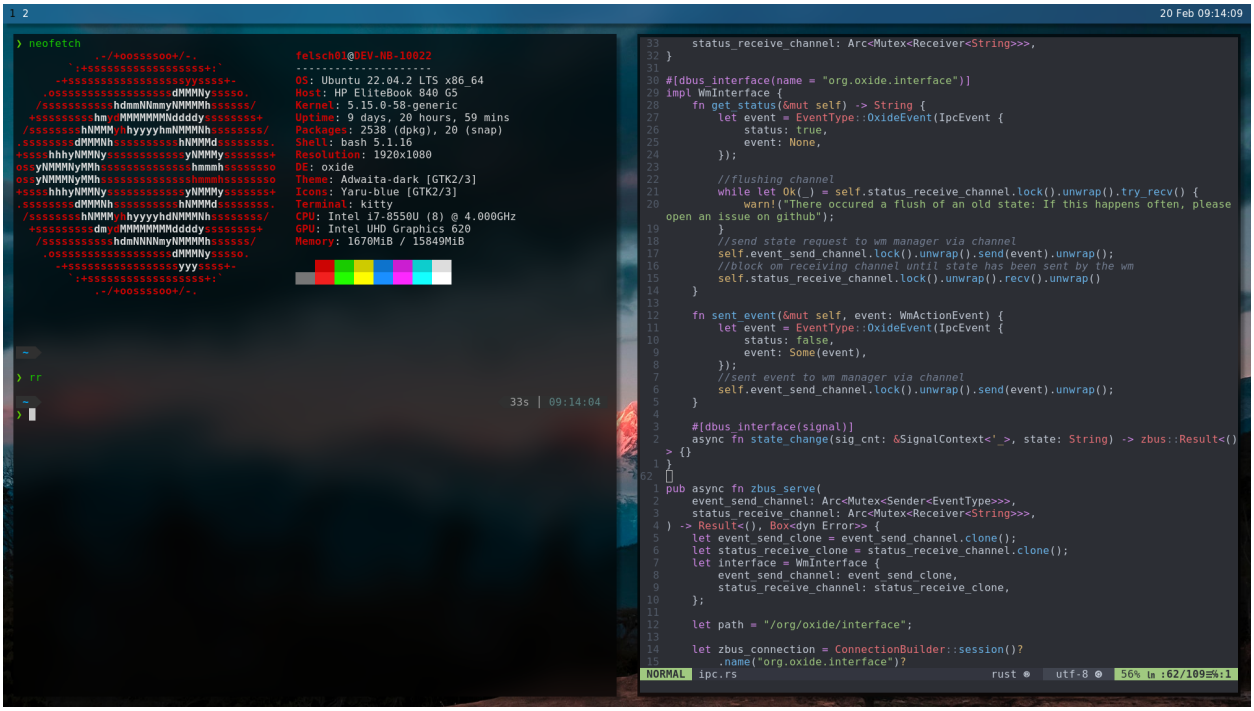

Fig. 1: *Oxide* screenshot

A short overview about the topics on this page:

The site “Introducing *Oxide*” contains basic knowledge for getting in touch with *Oxide*.

Furthermore the category “Project procedure” contains every relevant documents and informations about the development process, e.g. specifications or critical technologies and their justifications.

The section “Diagrams” is for the different diagrams from component and hmi behavior over UML-diagrams to automatically generated class-diagrams.

Last but not least there is “Using *Oxide*”, which describes the process of installation as well as the different configuration options, containing the man pages among other things, too.

CHAPTER

ONE

CONTENTS

1.1 Introducing *Oxide*

1.1.1 Introduction

Oxide is a dynamic tiling windowmanager for X11.
Windows are automatically arranged in a grid-like fashion. The user can then move and resize windows by using keyboard shortcuts. Custom defining of those shortcuts to launch applications is also possible.
Oxide tries to maximize the screensize by removing unnecessary borders and decorations as well as to be as keyboard friendly as possible. Everything can be done via the keyboard.
Oxide is done via a configuration file. The configuration file is written in YAML and can be reloaded at runtime. This makes the user able to change the behavior of *Oxide* without having to restart it.

1.1.2 Terminology

Window
: An X11 application window such as a browser or terminal.

Workspace
: A workspace contains multiple windows. The user can switch between several workspaces.

Layout
: Layouts are different algorithms placing windows.

1.1.3 Target group

The target group contains power users with advanced Linux knowledge.

1.1.4 Product functions

The *Oxide* window manager gives the user the ability to start and quit applications through its interface. The software itself is supposed to support dynamic tiling, allowing the user to arrange multiple applications in a grid-like arrangement optimizing screen space utilization. Along with this it supports both floating and static applications, giving the user flexibility in his window management.

Therefore applications are expected to be moved around the screen by the user to different tiled positions or to float as a separate window.

Keyboard inputs are handled effectively, allowing the user to control all aspects of the applications by using keyboard shortcuts. The software supports focusing on different windows, making the user able to switch between applications.

Oxide supports multiple workspaces as well as multiple monitors, allowing the user to create and switch between different virtual desktops and to extend their workspace across multiple screens. It also provides an interface for configuring various settings and options, such as the number of workspaces, monitor arrangement, and more.

Allowing the user to specify which applications should start automatically when the software is launched is another feature.

The window manager integrates a taskbar providing an intuitive and streamlined way to switch between open applications and workspaces. For this it is necessary to support popular utilities like Drun or Rofi.

Inter process communication (IPC) is used for interacting between different applications and services, allowing for a seamless integration with the users workflow. The window manager uses a config file in which the user can manage his preferences and settings. Also power management features are included, such as screen locking after a specified timeout to help conserve energy and improve security. For improving the overall user experience the software includes visually appealing animations.

1.1.5 Config file

Oxide can be configured via its config file. This includes keybindings, appearance and more. Before editing, the global config file located under

```
/etc/oxide/config.yml
```

should be copied into the users home directory under

```
~/.config/oxide/config.yml
```

For a more detailed description of the config see *configuration of Oxide*.

1.1.6 Logging

Oxide log messages are written to

```
/var/log/syslog
```

1.1.7 Files

Per-user config file:

```
~/.config/oxide/config.yml
```

Global config file:

```
/etc/oxide/config.yml
```

Oxide desktop file:

```
/usr/share/xsessions/oxide.desktop
```

1.1.8 Bugs

Please open an issue on https://github.com/DHBW-FN/OxideWM/issues .

1.2 Project procedure

1.2.1 Customer specification (Lastenheft)

Product goal

Getting to know how window managers and Xorg work. Development of a working window manager.

Target group

Target group contains power users with advanced Linux knowledge.

Product functions

Fundamental

The *Oxide* window manager should give the user the ability to start and quit applications through its interface. The software itself is supposed to support dynamic tiling, allowing the user to arrange multiple applications in a grid-like arrangement optimizing screen space utilization. Along with this it should support both floating and static applications, giving the user flexibility in his window management.

Therefore applications are expected to be moved around the screen by the user to different tiled positions or to float as a separate window.

Keyboard inputs are to be handled effectively, allowing the user to control all aspects of the applications by using keyboard shortcuts. The software should support focusing on different windows, allowing the user to switch between applications.

Basic

The software should support multiple workspaces as well as multiple monitors, allowing the user to create and switch between different virtual desktops and to extend their workspace across multiple screens. It is also supposed to provide an interface for configuring various settings and options, such as the number of workspaces, monitor arrangement, and more.
Autostarting of applications is supposed to be another feature, allowing the user to specify which applications should start automatically when the software is launched.
The window manager should integrate a taskbar providing an intuitive and streamlined way to switch between open applications and workspaces. For this it is necessary to support popular utilities like Drun or Rofi.

Desired

Inter process communication (IPC) should be used for interacting between different applications and services, allowing for a seamless integration with the users workflow.
The window manager is supposed to use a config file in which the user can manage his preferences and settings. Also power management features should be included, such as screen locking after a specified timeout to help conserve energy and improve security.
For improving the overall user experience the software is to include visually appealing animations.

Documentation

Keeping track of tickets with timestamps.

1.2.2 Technical specification (Pflichtenheft)

Product functions

1. Fundamental

1.1 starting and quitting apps

The *Oxide* window manager should give the user the ability to start and quit applications through its interface.

1.2 tiling functionality

The software itself must support dynamic tiling, allowing the user to arrange applications in a grid-like arrangement optimizing screen space utilization. Along with this it shouis supposed to support both floating and static applications, giving the user flexibility in his window management.

1.3 moving windows

Therefore applications are expected to be moved around the screen by the user to different tiled positions or to float as a separate window.

1.4 controllable via keyboard

The user must be able to control all aspects of the applications by using keyboard shortcuts.

1.5 controllable via IPC

The user must be able to control all aspects of the applications by using the IPC interface.

1.6 focusing windows

The software must support focusing on different windows, allowing the user to switch between applications.

1.7 key-forwarding

When a window is stated to be focused the keybord inputs must be directed to the focused application.

2. Basic

2.1 multiple workspaces

The software must support at least ten workspaces, allowing the user to create, quit and switch between different virtual desktops.

2.1.1 move window to workspace

The software must support moving a window to another workspace. When this functionality is executed, the window-manager must: - remove the window from the old workspace - add the window to the new workspace

2.1.2 switching between workspaces

When this functionality of the window manager is executed, the window manager must: - display all windows that were opened or moved to this screen (if fullscreen is not active).

2.1.3 closing workspaces

When this functionality of the window manager is executed, the window manager must: - close all windows that are currently in this workspace - switch to another open one, so that the user is never on "no" workspace When the last workspace is closed, a new workspace must be created. The windowmanager must then switch to the newly created workspace.

2.2 config

The window manager must provide an interface for configuring various settings and options. This configuration must be human readable or must provide another interface so that an linux averse person can change the settings. There must be default values for the configuration elements, so that when a users configuration is incorrect, the windowmanager still starts. Furthermore, the configuration must be applied to the windowmanager, every time it is started.

2.2.1 keybindings

For every command, that the window manager provides, the user must be able to configure a keybinding specified as below. A keybinding must contain exactly one none modifier key such as *1*, *2*, *A*, *B*, ... It can contain any combination of the following modifiers: *Alt*, *Meta*, *Command*, *Shift*. To enhance the configurability, the user must be able to assign multiple commands to a single keybinding.

2.2.2 autostart

Autostarting of applications must be supported, allowing the user to specify which applications should start automatically.

2.3 utilities

The window manager should integrate a taskbar providing astreamlined way to switch between open applications and workspaces. For this it is necessary to support popular utilities like Drun or Rofi.

3. Desired

3.1 multiple screens

The windowmanager must empower the user to use multiple screens connected to his computer.

3.1.1 multiple screens workspaces

To take full advantage of the multiple screens, the windowmanager must allow workspaces on every stream.

3.1.1 multiple screens moving windows

The windowmanager must provide a way, to move windows between workspaces across screens.

3.2 screen locking

Also power management features should be included, such as screen locking after a specified timeout to help conserve energy and improve security.

3.3 statusbar

The windowmanager should provide a statusbar of some sort, to keep track of which workspaces exist, and on which workspace the user currently operates.

4. Documentation

Keeping track of tickets with timestamps.

5. Data relevant for the user

The application will be running locally so it needs to be downloaded and installed by the user before using it for the first time. Files needed for configuration will be stored locally.

6. Product performance - requirements

Claim is having no delay between key inputs and the following action. If possible, visible tasks should be performed in under a 24th of a second. This is not possible for opening application windows.

7. Quality requirements

Randomly crashing must not happen. If configurations are invalid they should be overwritten by default values. The config file should be formatted as JSON.

8. User Interface

Controlling the window manager will only be possible by using the keyboard. A mouse can be used to focus on individal frames and interact with application interfaces like webbrowsers.

9. Non-functional requirements

An installer with package manager cargo is required.

10. Project enviroment

10.1. Software

The product is supposed to be used on Unix based operating systems with an X11 instance running. Furthermore there is no other running window manager accepted.

10.2. Hardware

Required hardware is at least one monitor as well as a keyboard working with the operating system. There are no hardware limitations.

10.3 Organizational framework

Since the code is licensed with GPL v3 there are no conflicts with GPL licensed libraries.

10.4 Product interface

The behavior of the window manager can be customized by changing the config files. Program actions will be stored in log files located under TODO .

11. Special requirements

11.1 Software

- x11rb
- buildin crate **log** for logging
- **Zbus** for IPC
- **Serde** for parsing

11.2 Development interfaces

- X11 API
- D-Bus

1.2.3 Project handbook

General project schedule

Researching technologies

All research results and discussions will be stored in the `concepts` folder. All documents will be written in markdown - there are no other formal restrictions.

Ticketing

Every task will be documented with a Git issue. The current status will be kept updated for the following states:

- TODO
- in progress
- open for review
- done

Branching

Every issue will get its own branch. A feature branch will be named after the following guideline:

```
feature/ISSUE<ISSUENUMBER>-<Featurename>
```

A bugfix branch will be named after the following guideline:

```
bug/ISSUE<ISSUENUMBER>-<Featurename>
```

The feature branches can freely be branched for testing purposes. These sub-branches can be merged back into to top-branch without any pull requests.

Crossbranching between feature branches is prohibited.

Every merge into the `main` branch has to be accepted and reviewed through a pull request. There should not be any rebase onto `main`. Working methods on the feature branches are open to developer.

Testing

All test logs are to be stored in the subdirectory `test_logs`. Those will not be published on GitHub. Upcoming issues should be documented with Git issues with the following format:

```
Titel: error-code

error description

\```
stackstrace
\```
```

Unittest

Logs from unittests are to be stored in `test_logs/unittests`. At the end of the project all logs should be pushed to GitHub with one commit. Unittests can be documented with their source code. The output has to be logged and saved.

Manual tests

Manual tests are stored in `test_logs/manual`. At the end the logs should be commited like the unittest logs. These logs are formatted like the following:

```
# Testname_Testdate

## Content

... What was tested? ...

## Test results

... Which errors occured, which functions worked? ...
```

Logging

Logging should work with the following levels:

- info
- trace
- warn

Scrum

Sprint duration: *1* week Sprint-Meeting: *weekly while the lecture*

1.2.4 Work package plan

Title	Duration [weeks]
Planning	3
Project setup	2
Concepts	3
Fundamental features	6
Basic features	5
Testing	6
Desired features	3
Feature freeze / bug fixing	2
Presentation planning	2

Hint: If the work package plan is not shown big enough to read, please click on it.

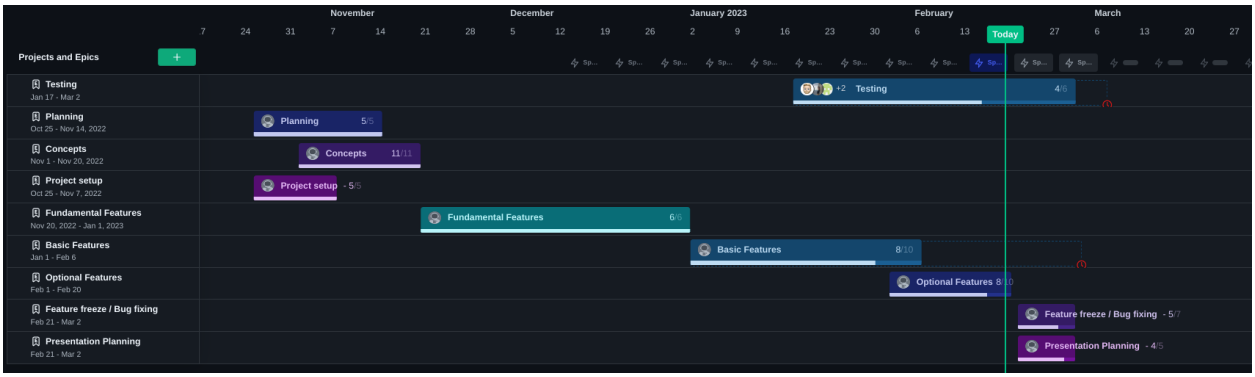

Fig. 1: work package plan

1.2.5 Technologies

Critical technologies

Rust as implementation language

This document outlines why rust has been chosen as the language of implementation for this project. But it needs to be mentioned that this choice was not purely made out of technical reasoning, because we think rust is fun and we enjoy to learn this language.

Technical arguments for Rust

Rust is a good choice for a system level language as it allows direct memory acces but also object orientation and offers a memory safe environment. Further this is achieved without a resource intensive garbage collection and high speeds in the realm of C are possible. Even though rust is still “relatively” new, it already has a stable ecosystem and a lot of libraries supporting it.

A first search for rust libraries allowing connections to the X-Server shows multiple results. The same is true for dbus- and IPC-libraries.

Therefore Rust seems to be a good choice for our project.

Inter Process Communication (IPC)

During a first discussion it was decided that *Oxide* should be controllable via an IPC mechanism. This functionality will be inspired by i3-msg.

Description

An IPC mechanism for the window manager is required. This is neccessary for:

- Taskbar
- External libraries
- Command line utility

Feature list

The following list includes currently proposed features

- everything that is achievable via the keyboard (kill, move, launch…)
- current state of the window manager including e.g. layout or windows

IPC integration solution

Since there are two types of events that have to be handled, there needs to be some separation between them.

One type are `xevents`, received from the `X11` instance, and the other one is custom events created by the user, received over `zbus`.

For this reason, each type of event will get its own loop on its own thread, which will await them and push them into a list shared between them. The events in this list will be taken care of by the window manager, who will execute the correct action based on event type and content.

Technical solution

The following sections describe the argument for the different IPC-mechanisms and libraries.

Requirements

As for the aforementioned use cases it will not be required to send large amounts of data. Only short messages will be exchanged between the clients. Also it is not expected that the IPC performance will have a significant impact on the usability of the system. Therefore some IPC options such as shared memory and semaphores will not be regarded as these options are not as easy to use and do not offer any significant advantages.

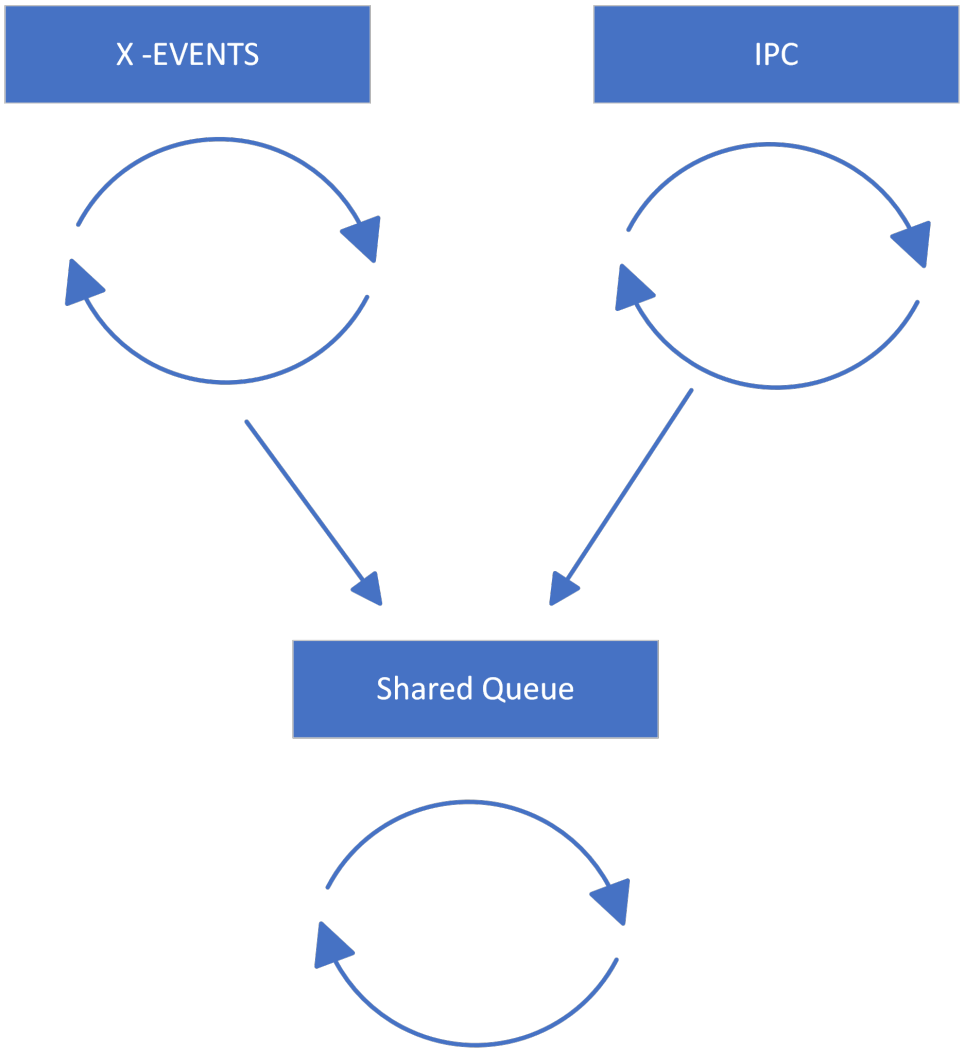

Fig. 2: IPC queue

Possible IPC mechanisms

There are multiple different ways of implementing IPC on posix systems.

FIFO

Named Pipes Wikipedia

- work like normal pipes, but are a permanent file on the system
- fasted regarded option
- good library support

Unix Sockets

Unix Domain Sockets Wikipedia

- work like TCP sockets
- very fast IPC mechanism
- easy to use and inbuilt library support

D-Bus

D-Bus Wikipedia
D-Bus documentation Rust
D-Bus create from freedesktop.org
D-Bus interface for Rust

- high level IPC mechanism
- based on unix sockets
- widely used in projects such as Gnome and KDE
- offers message queuing, tow way communication and is supposed to offer a easy to use interface
- comparetively slow compared to FIFO or UNIX sockets

Key Takeaways

Discussion about IPC on Stackoverflow
Stackoverflow Comparison D-Bus vs Unix Sockets
Practical uses of D-Bus

- TCP Sockets are only about 16% slower compared to FIFO
- IPC performance is in most cases not the bottleneck
- Sockets allow for two way communication
- Sockets are more widely supported

- IPC interface should be abstracted, so that the IPC mechanism can be changed in a later stage
- D-Bus should offer a high level, simple to use IPC mechanism

Conclusion

After a technical discussion with the team the conclusion came to that **D-Bus is most suitable**. The **performance is deemed non critical** in our use case and the ease of use will be benefitial for the project. None the less, the IPC interface should be **created in an abstract manner** allowing for a possible replacement of the underlying IPC mechanism.

Implementation

Available libraries

There seem to be two main projects striving to provide D-Bus support for rust.

Zbus project repository
Zbus crate
Zbus documentation

- official D-Bus rust implementation by the freedesktop.org foundation
- pure rust implementation
- extensive documentation
- examples

dbus-rs repository
dbus crate

- wrapper library for libdbus -> libdbus dependency - examples

Conclusion

Zbus seems to have some advantages over D-Bus-rs, mainly: - official freedesktop.org library - pure rust -> no libdbus dependency - Extensive documentation - Due to being an official library, maintenance is most likely certain

Therefore we came to the conclusion **to use zbus** as our IPC library.

Conclusion

As IPC-mechanism **dbus** was chosen as most suitable. The rust library **zbus** has been chosen as implementation.

Reactiveness

The window manager should at all stages be reactive and only use a minimal amount of resources. In order to assure this polling of threads should never be implemented.

Event Sources

Events for the window manager can be created by multiple sources.

1. `X-Events` Generated by the X-Server and regarding userinput, clients requests and more.
2. `IPC-Events` Generated via the IPC-mechanism by the user or programs running on the system.

In order to be able to handle these two event sources two possible solutions are available:

1. Polling of both event-sources

poll for X-Event
poll for IPC-event
execute received event

This layout allows for a very simple single threaded implementation of the window manager but requires a frequent polling loop in order to allow for a reactive behavior.

Therefore this layout is not suitable for our requirements as this would be very inefficient.

2. Multithreaded queues

This layout requires three different queues. 1. X-Event queue -> waits until an x-event is received 2. IPC-Event queue -> waits until an IPC-event is reveived 3. mainloop -> waits for an event forwarded to the mainevent-queue by the above mentioned threads

If implemented in the above shown way, no polling is required. This therefore allows to sleep the threads until an event is received and the kernel wakes up the thread. It therefore will be very reactive and use only minimal resources.

=> Therefore we will implement this layout.

Technical solution

Waiting for event

The following documents functions of the chosen libraries that support waiting for an event:

1. X-Event The `x11rb` library supports waiting on an event: wait_for_event method
2. IPC-Event Zbus is asynchronous in nature. The handling functions are registered to the server and executed when an event is received. Zbus Example
3. mainloop as shared queue the channel implementation of the rust standard library was chosen. This allows to wait for an event: recv method

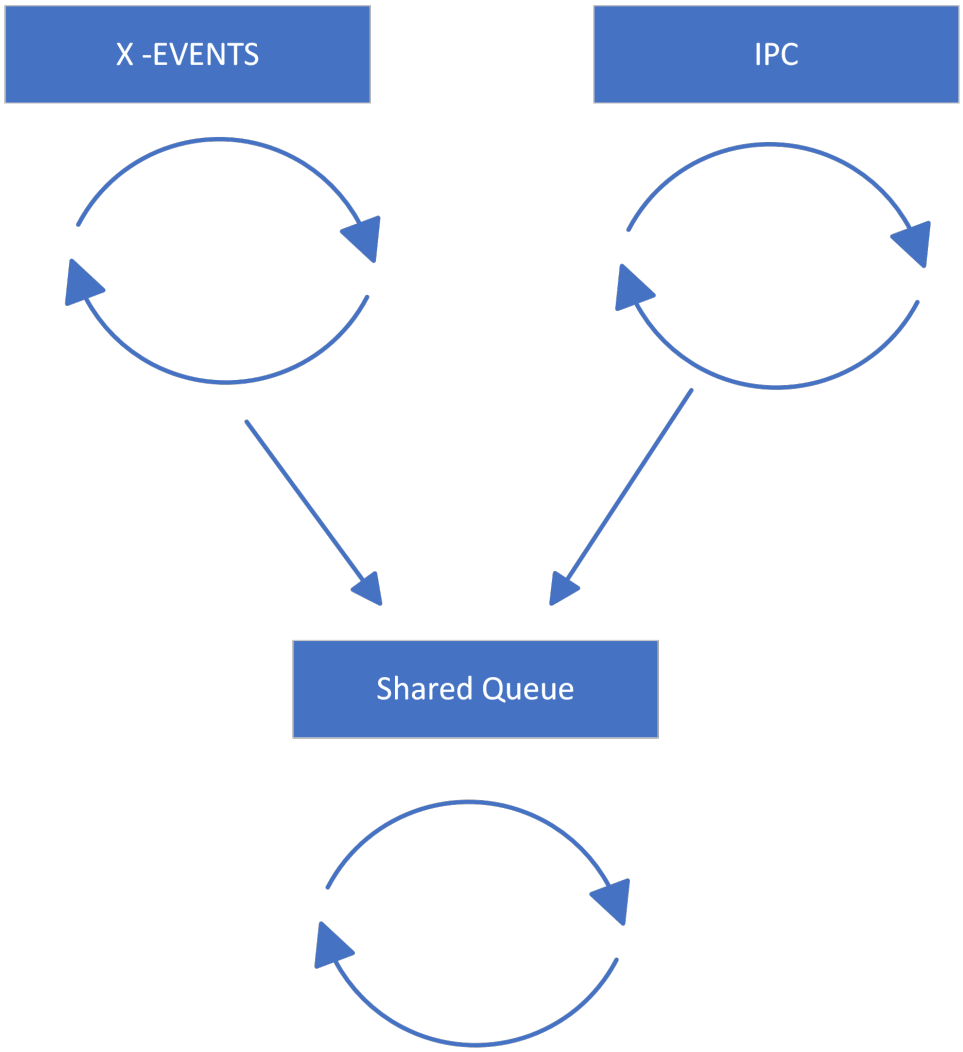

Fig. 3: IPC-queue

XCB

This document covers some basics of XCB. This contains strengths and weaknesses and other aspects worth mentioning when trying to decide between it and Xlib.

All information was pulled from the XCB tutorial.

What is XCB

It is a alternative to the X-server interface Xlib. Both offer the ability to communicate with a systems x-server, which is a crucially important aspect of a window manager.

XCB eliminates the need for programs to implement the X protocol layer by offering low-level access to X-servers. Since the protocol is standardized, it is possible to talk to any X-server with XCB.

What does XCB provide over Xlib

- Toolkit implementation
- Direct protocol programming (see section *Why not to use XCB*)
- Leightweight emulation of commonly used portions of the Xlib API
- XCB does not lock itself while waiting for a response it send to a x-server instance like Xlib. This avoids needless stalling while a request is processed. Instead, XCB binds a cookie to a request which can then be used to ask for a pointer to the corresponding reply. This not only enables reading of the reply only when it is required, but also is ~5 times faster then locking while waiting for a reply.

Latency comparison

- Request: `W`
- Reply: `R`
- No action: `-`
- Amount of send requests: `N`

Xlib

Due to how Xlib works, a request-reply cycle works like this:

```
W-----RW-----RW-----RW-----R
```

The total time is `N * (T_write + T_round_trip + T_read)`.

XCB

XCBs request-reply cycle looks like this:

```
WWWW--RRRR
```

The total time is `N * T_write + max(0, T_round_trip - (N-1) * T_write) + N * T_read`.

Conclusion

XCB offers considerably faster event handling. The tutorial linked at the top of this document also proiveds the source code and results of a benchmarking which leads to the same results.

Why to not use Xlib

Xlib is quite big: Xlib is bigger then XCB and can therefore not be used with minimalistic systems. However, the target groups of this project are (semi-)well equiped computers, which makes this not as much of an advantage.

Latency: Xlib manages events synchronously, with the principle of *first in, first out (fifo)*. This can cause delays when dealing with a bigger amount of events within a short notice.

Multithreading: XCB appears to support this feature. Xlib can to some degree work with multiple threads too but its API was not designed for this purpose which makes it difficult to work with as well as error-prone.

Why not to use XCB

Direct protocol access: This can be good or not depending on the system an application will run on. Xlib performs chaching, layering and other optimizations by itself. XCB does not provide this feature.

Summary

XCB seems to be a lighter version of Xlib that also solves issues like multithreading while losing some optimizations such as caching in the process. However, XCB offers a considerably quicker request-reply method, which also allows reading of replies when necessary and does not force an aplication to do so when the response becomes available.

Window manager configuration

At the beginning of the project, it was decided to have an external config file which can be personalised freely. For this to happen a suitable file format had to be chosen.

File format

Typical file formats used for config files are `JSON`, `YAML` or `XML`. Due to poor readability, `XML` has been ruled out from the start. Unfortunately does `JSON` not support any comments inside the file, which was decided to be an important feature. Therefore it was decided to use `YAML` as the proper file format for *Oxide* config files.

Technical implementation

The following sections describe the argument for the chosen parsing library.

Parsing the config file

The config file needs to be parsed before we can accesses the stored data. This should be as easy and effortless as possible. The preferred solution for this is to have a parser that outputs a single struct which contains all config values.

Library

The serde crate is the obvious choice for serialization and deserialization in the rust eco-system. It is widely supported and has subcrates such as serde_yaml for specific file formats.

Additionally features such as giving fiels default values when not part of the config are possible.

```
#[serde(default='default_value')]
```

Using this it is possible to use default values for not present or wrongly assigned variables.

Conclusion

After evaluating all aspects the team came to the conclusion to use YAML as file format and the serde_yaml crate as parser.

Programming paradigms

- Client-Server model
- event driven, functional, imperative, procedural, structured programming

Programming languages

- Rust
- bash/shell
- Make

Development Environment

- Posix
- Xorg
- Xephyr
- CLion / Vim / Visual Studio Code

Hardware

- Personal Computer

D-Bus interprocess communication (IPC)

D-Bus interface description

OxideWM has a D-Bus interface for IPC communication. This is primarily used in the `Oxide-IPC` library. This interface mainly gives access to the current state of *Oxide*. This state includes the loaded config, current windows, layouts, workspaces… It also allows to execute oxide commands.

Interface

```
org.oxide.interface
```

D-Bus Method Calls

Returns the current *OxideState* as a JSON object:

```
get_state() -> String
```

Executes the given command:

```
sent_event(WmActionEvent) -> void
```

D-Bus Signal

Returns the current oxide state when change occurs to the subscribers:

```
state_change -> String
```

1.2.6 Testing *Oxide*

Running tests

Automated and integration tests can be run using the main makefile:

```
make test
```

Where to find test results

Automated test logs are exluded from version control to avoid cluttering. Manual test results and findings can be found in:

```
test/results
```

Unittests

Unittests are not used in this project due to the significant amount of human input required to complete them. Instead, integration and automated tests are used to test and validate features.

Integration tests

Since the project requires some very restrictive setup, like a connection to the X11 server, which can only be granted once at a time, intergration tests are very limited as well, due to them running in parallel. They are currently used to validate that the projects config parser works correctly, which includes checking for wrong datatypes or missing fields in the config file. Additionally, the creation- and switching-process of workspaces is tested.

Automated tests

In this project, an automated tests is defined as a test that is performed on the full build, but does not require any human input. This is useful for testing much of the basic functionality that the project should support after each new update while removing the significantly higher test duration a human reviewer would require.

Unfortunately it is not possible to test *everything* using this method, and issues found by this kind of tests have to be manually traced back to their origin as well, as the only information the testing framework has access to is a JSON-dump of the entire windowmanager.

Automated tests for this project work by using `Xephyr` in combination with `oxide-msg` as well as a custom testing framwork tailored to make writing new tests as simple as possible. The files relevant for automated testing are located here:

```
test/resources
```

Functionality being tested automatically:
- opening windows (xterm and kitty, both are terminals)
- closing windows (xterm)
- moving focus between windows (xterm and kitty, 5 windows total)
- moving windows / switching window position (10 movements per layout)
- switching layout (vertical stripes, horizontal stripes, tiled layout)
- closing the windowmanager

Manual tests

Manual tests are used to cover all other areas ignored by the previous testing methods.

Manually tested features are:
- installation of the windowmanager
- running the fully installed version of the project as a real windowmanager
- keyboard inputs
- mouse inputs
- autostarting applications
- interaction with `dmenu`

In addition, this type of test is used to narrow down issues after they are discovered by automated tests.

1.3 Diagrams

1.3.1 Components and behavior diagrams

The individual components and their behavior in certain situations are shown below.

Hint: If the diagrams are not shown big enough to read, please click on them.

Components

Change request

New application

New application user perspective

Switch workspace user perspective

Switch workspace user perspective

1.3.2 HMI behavior

The individual components and their behavior in certain situations are shown below.

Hint: If the diagrams are not shown big enough to read, please click on them.

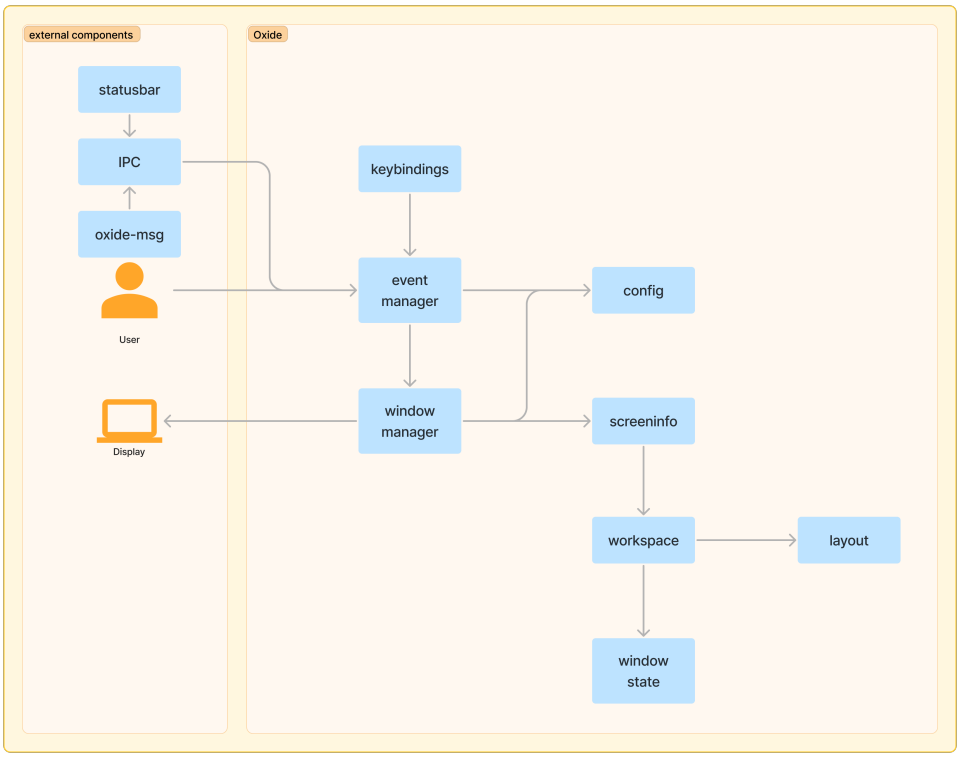

Fig. 4: components

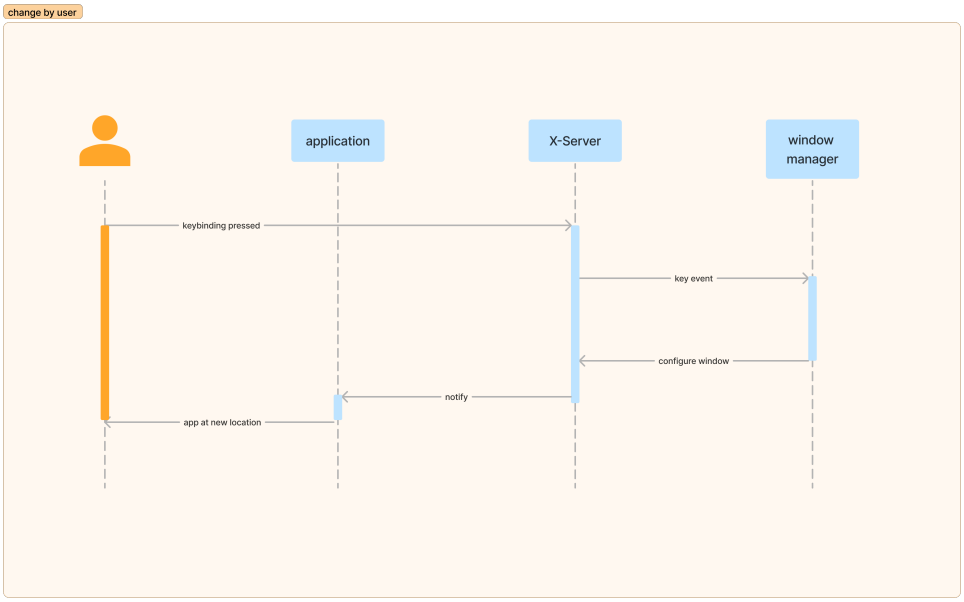

Fig. 5: behaviour while incoming change request

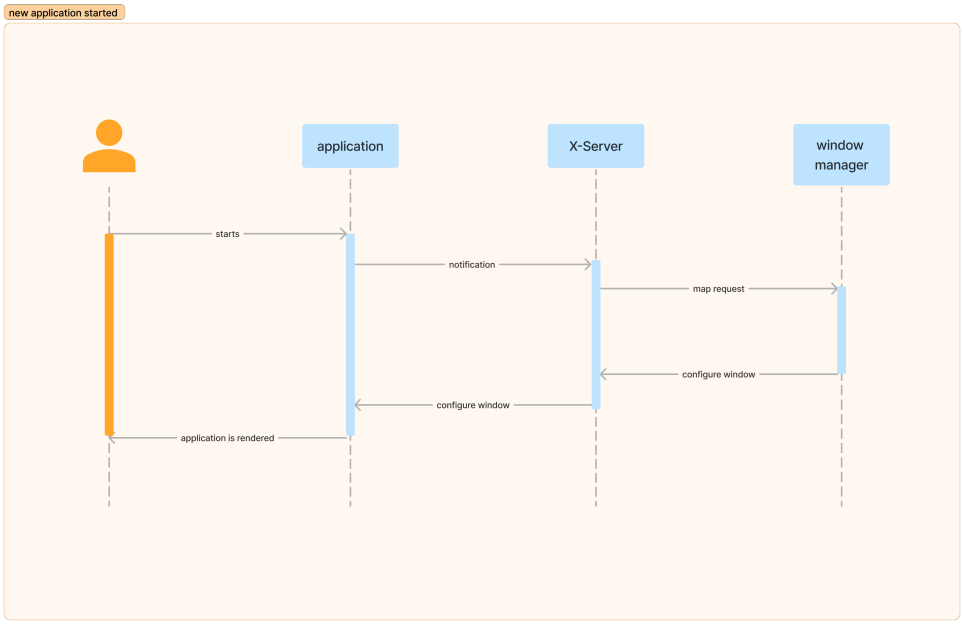

Fig. 6: behaviour when new application is demanded

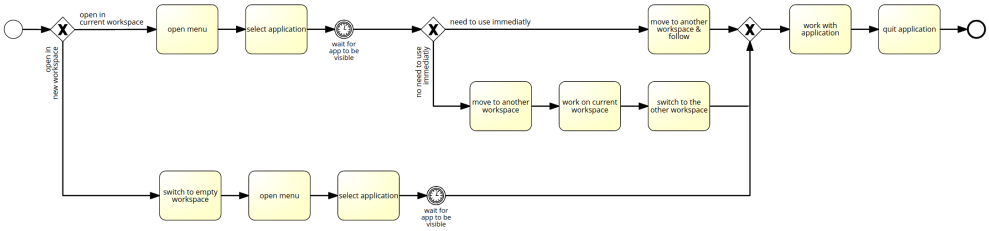

Fig. 7: behaviour when new application is demanded from the user perspective

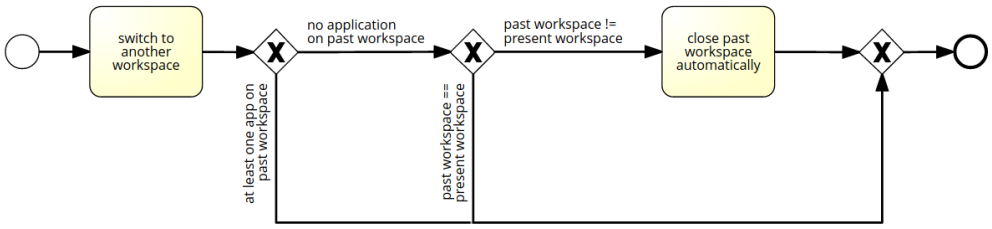

Fig. 8: behaviour when a new workspace is selected from the user perspective

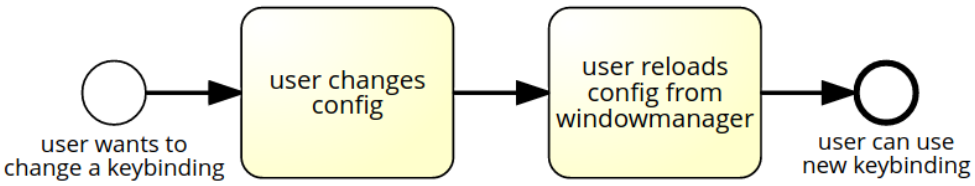

Fig. 9: behaviour what happens during a config change from the user perspective

Horizontal layout

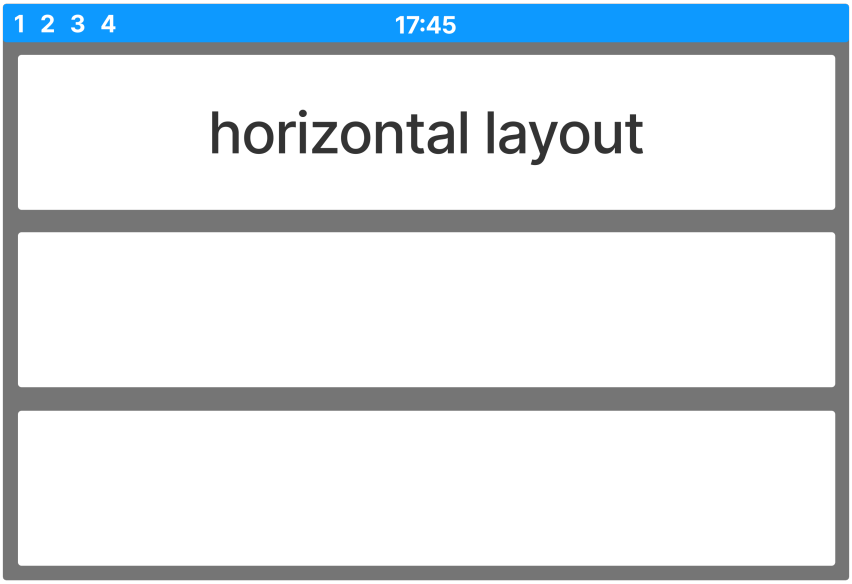

Fig. 10: horizontal layout

Vertical layout

Tiled layout

Workspaces in statusbar

1.3.3 Flowcharts

The following flowcharts show the technical structure and sequence of the product.

Hint: If the diagrams are not shown big enough to read, please click on them.

Main event loop

Instantiation

Eventhandling

Communication window manager and statusbar

Register keybinds

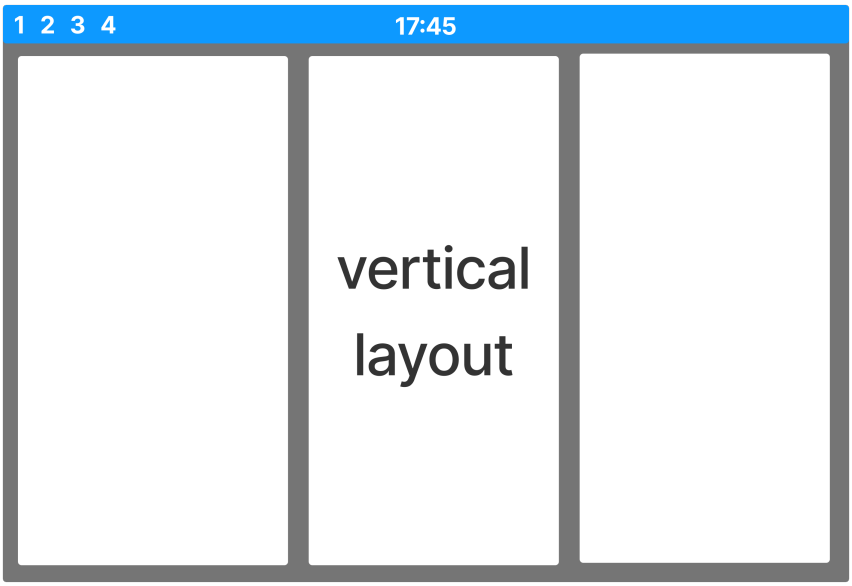

Fig. 11: vertical layout

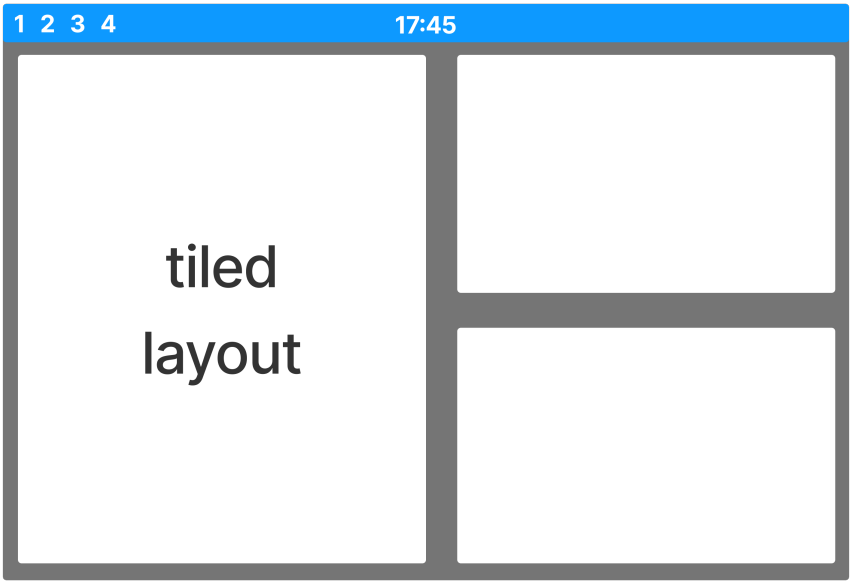

Fig. 12: tiled layout

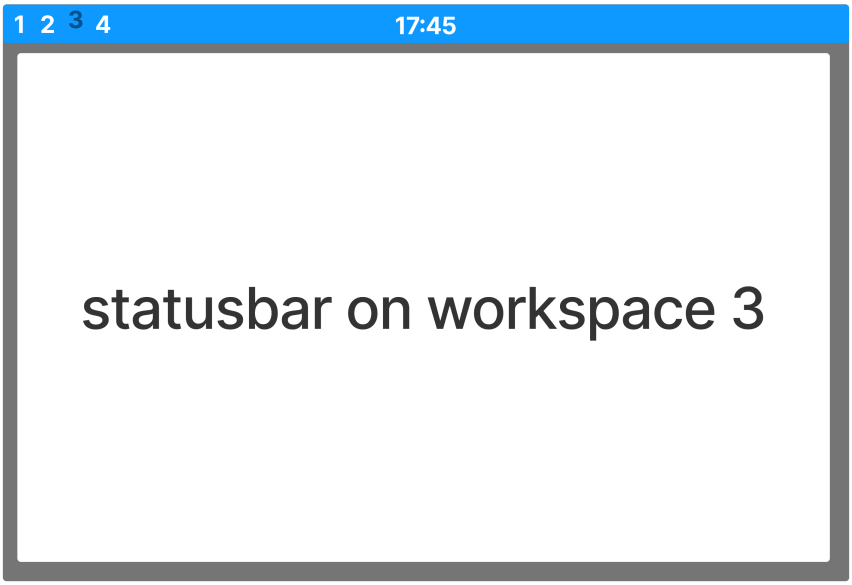

Fig. 13: workspace 3 is active

Getting associated keybind when key pressed

1.3.4 Class diagrams

All shown class diagrams are **automatically generated**.

extensions

All shown class diagrams are **automatically generated**.

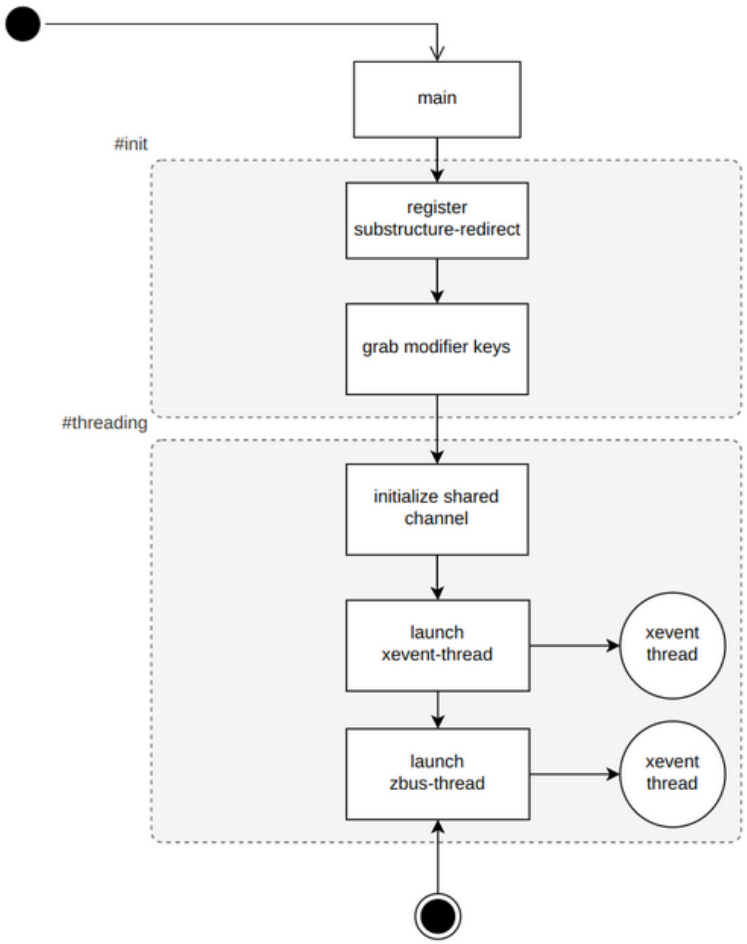

Fig. 14: main event loop

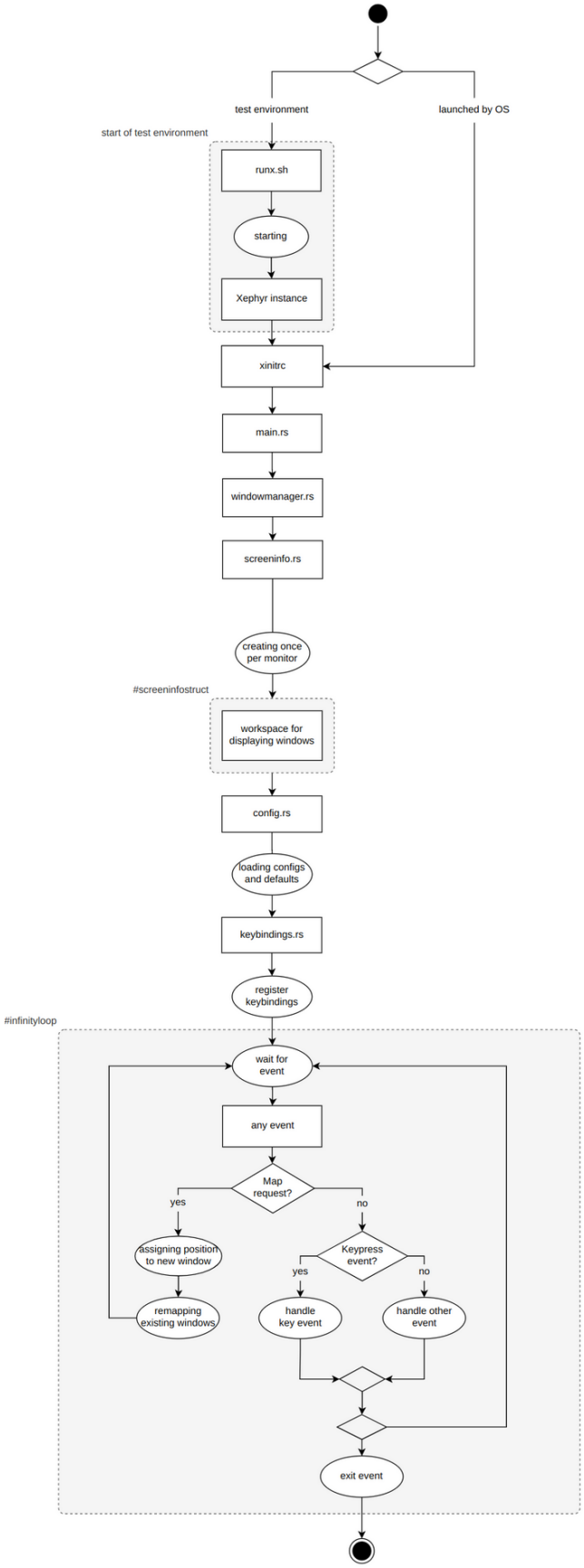

Fig. 15: instantiation

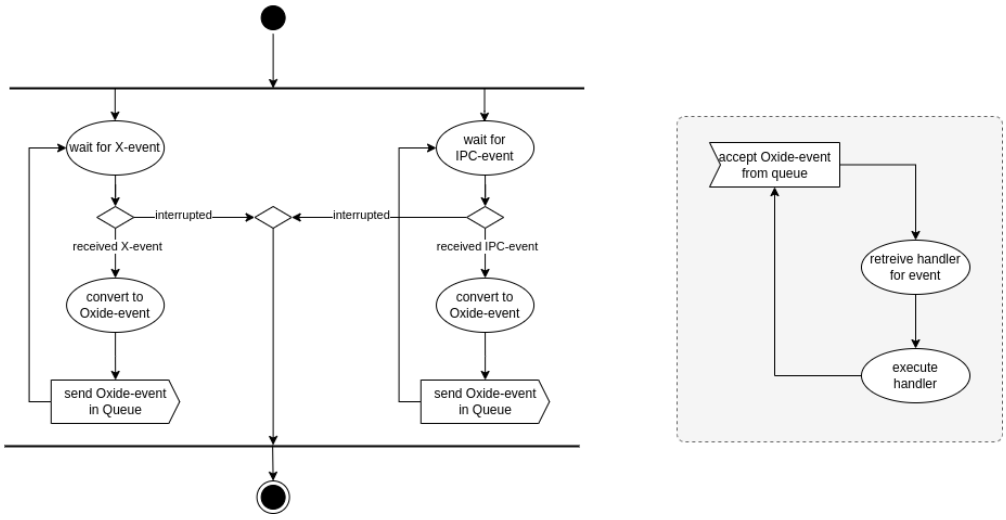

Fig. 16: eventhandling

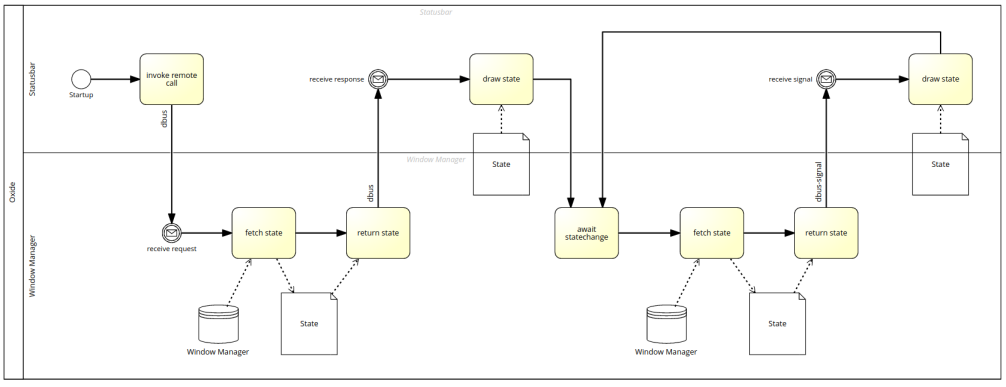

Fig. 17: window manager and statusbar communicating

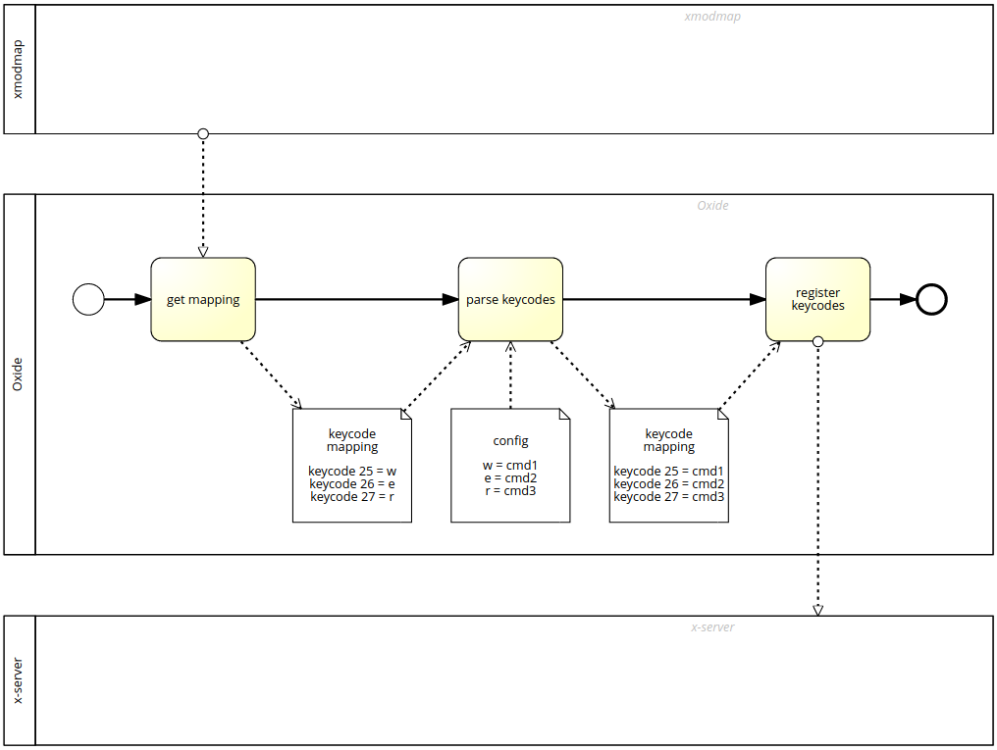

Fig. 18: register keybinds

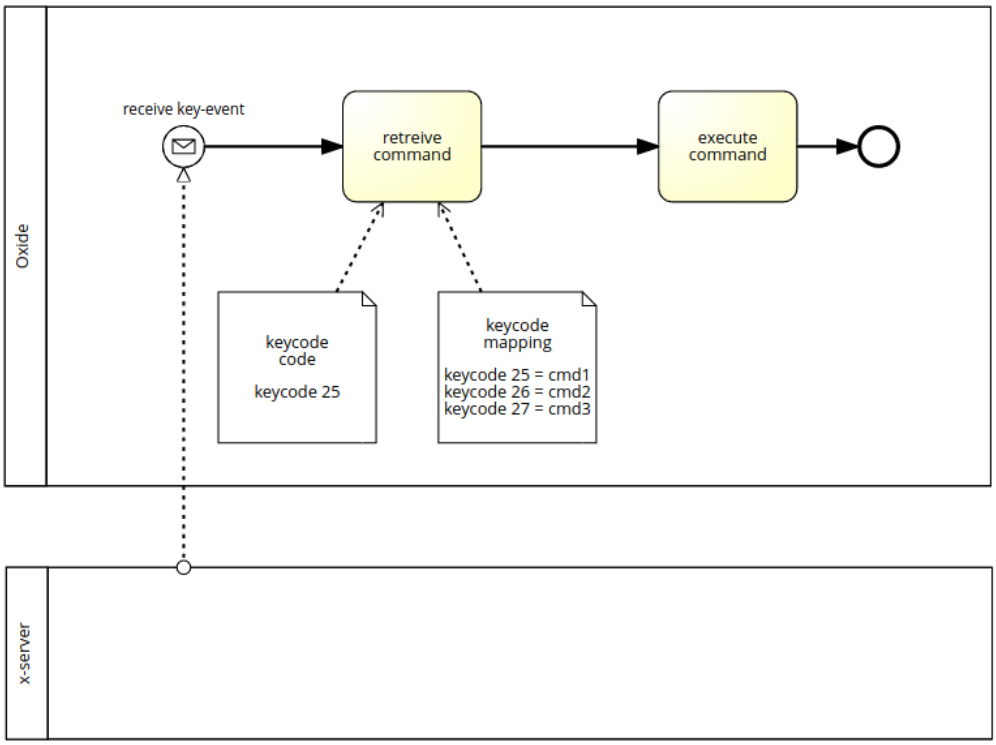

Fig. 19: associated keybind for pressed key

oxide-bar

Hint: If the diagrams are not shown big enough to read, please click on them.

config

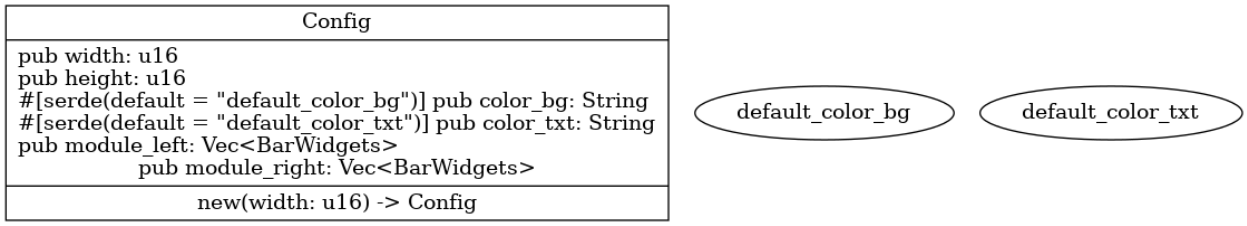

Fig. 20: config.png

main

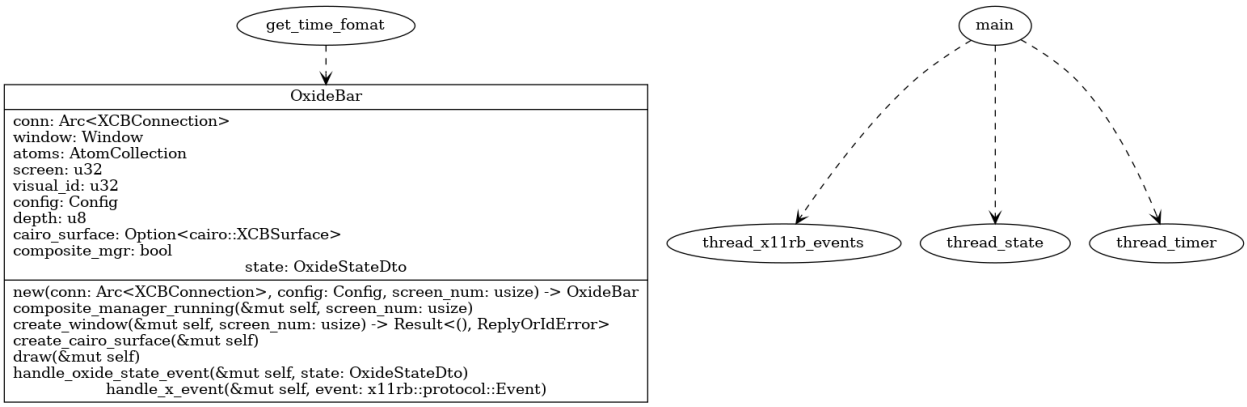

Fig. 21: main.png

xcb visualtype

oxide-ipc

Hint: If the diagrams are not shown big enough to read, please click on them.

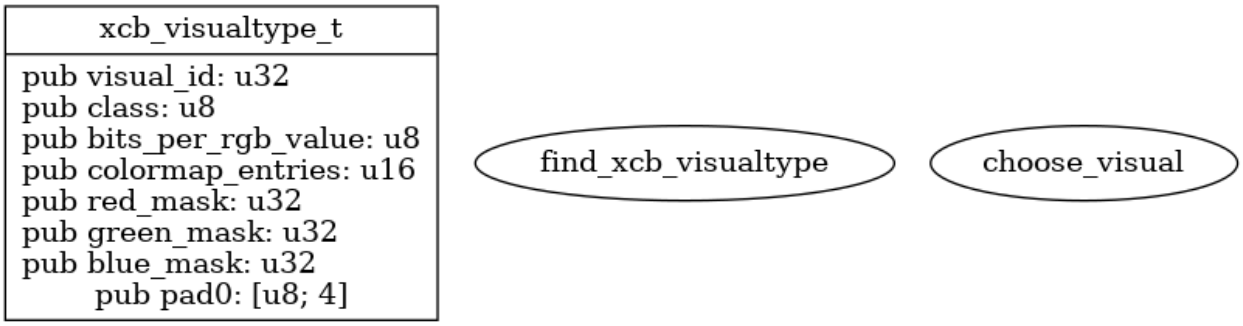

Fig. 22: xcb_visualtype.png

ipc

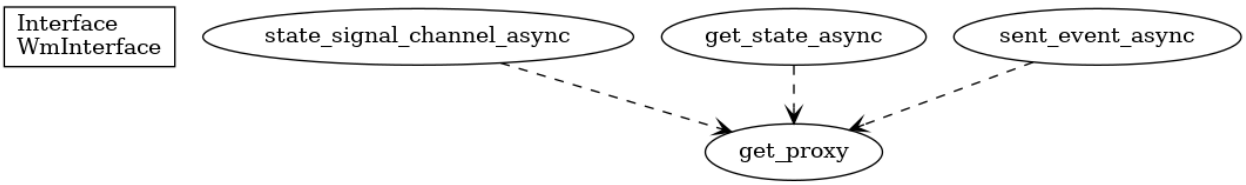

Fig. 23: ipc.png

lib

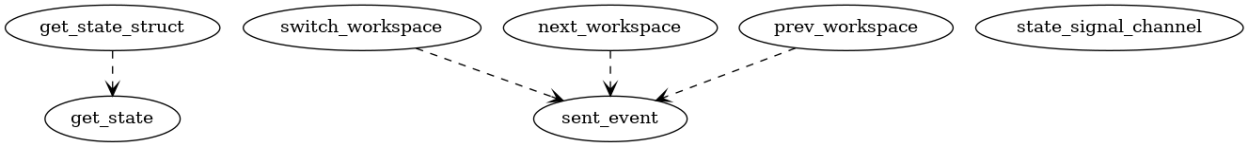

Fig. 24: lib.png

oxide-msg

Hint: If the diagrams are not shown big enough to read, please click on them.

main

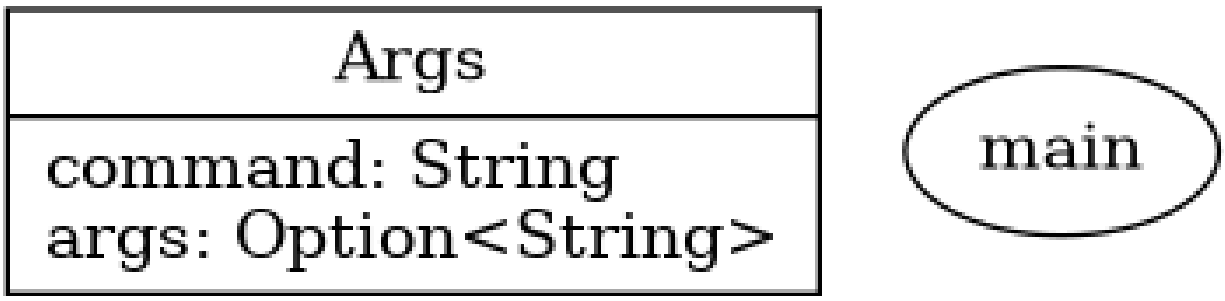

Fig. 25: main.png

windowmanager

All shown class diagrams are **automatically generated**.

config

Hint: If the diagrams are not shown big enough to read, please click on them.

commands

config

eventhandler

Hint: If the diagrams are not shown big enough to read, please click on them.

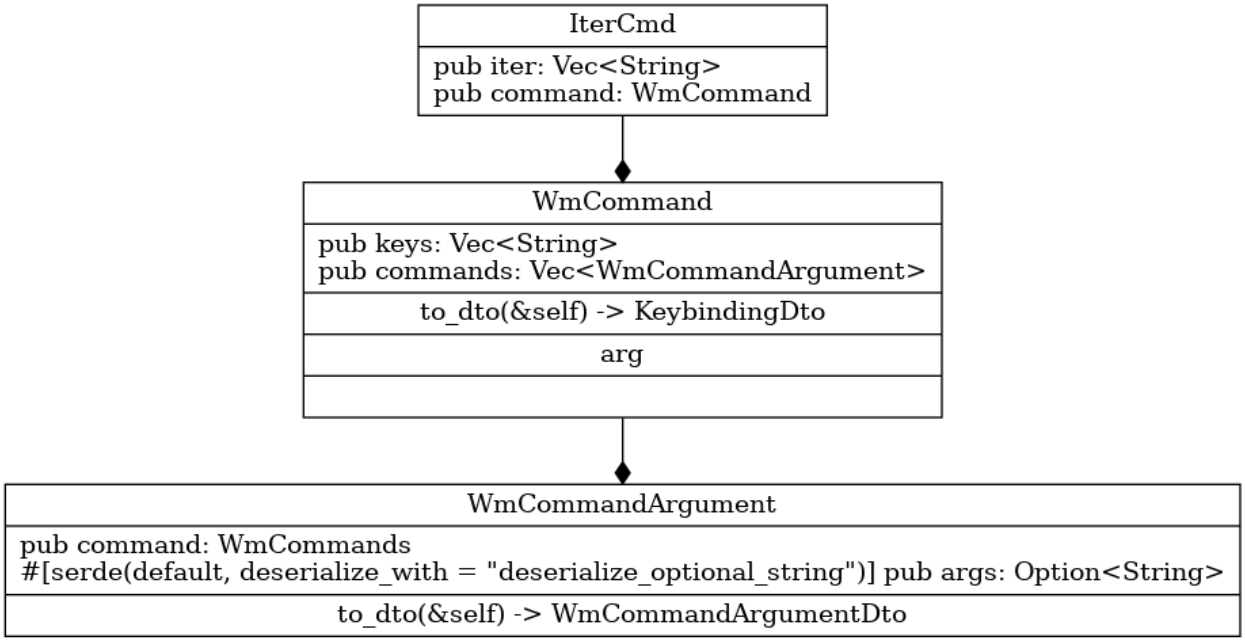

Fig. 26: commands.png

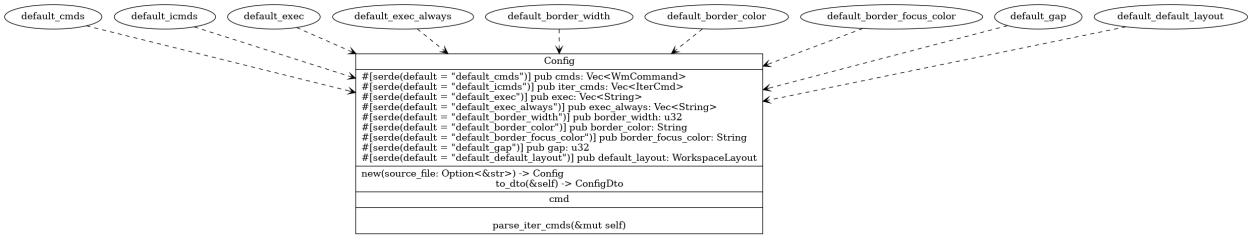

Fig. 27: mod.png

events

Fig. 28: events.png

eventhandler

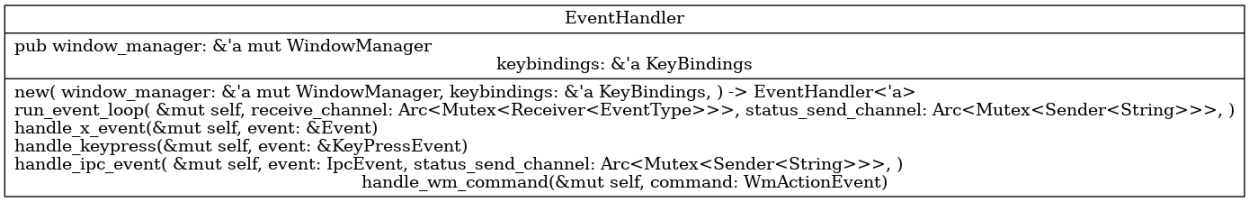

Fig. 29: mod.png

screeninfo

Hint: If the diagrams are not shown big enough to read, please click on them.

error

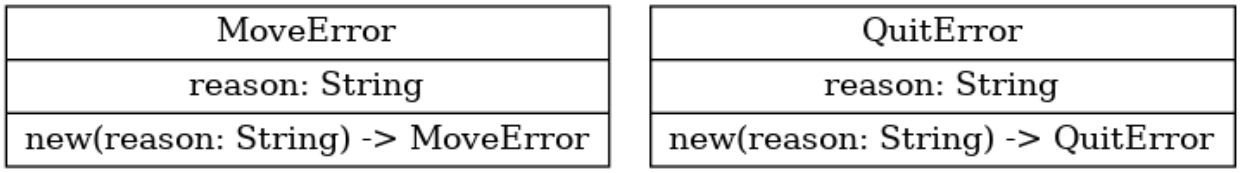

Fig. 30: error.png

screeninfo

Fig. 31: mod.png

setup

Hint: If the diagrams are not shown big enough to read, please click on them.

connection

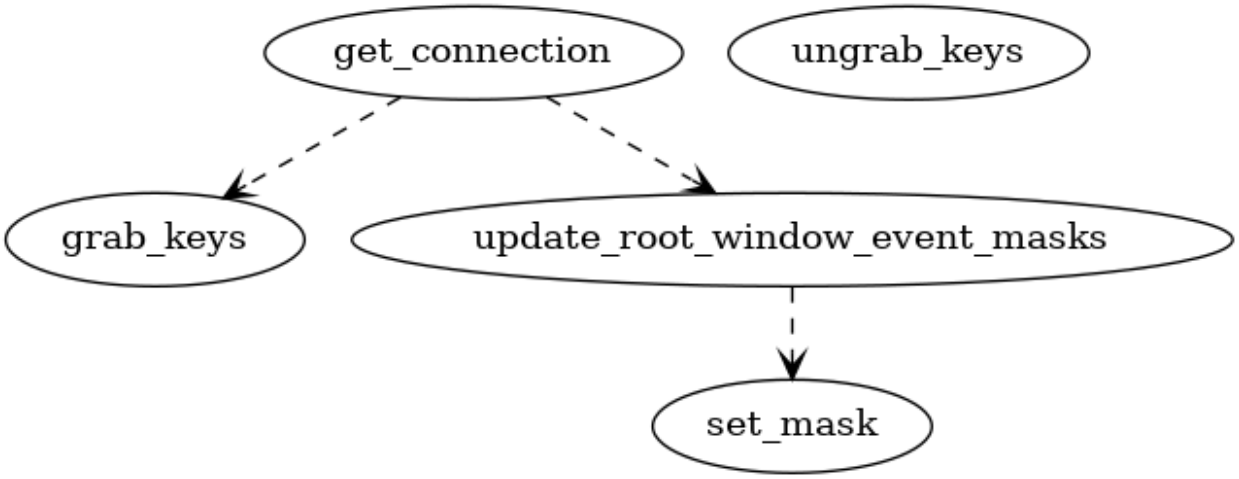

Fig. 32: connection.png

windowmanager

Hint: If the diagrams are not shown big enough to read, please click on them.

windowmanager

workspace

Hint: If the diagrams are not shown big enough to read, please click on them.

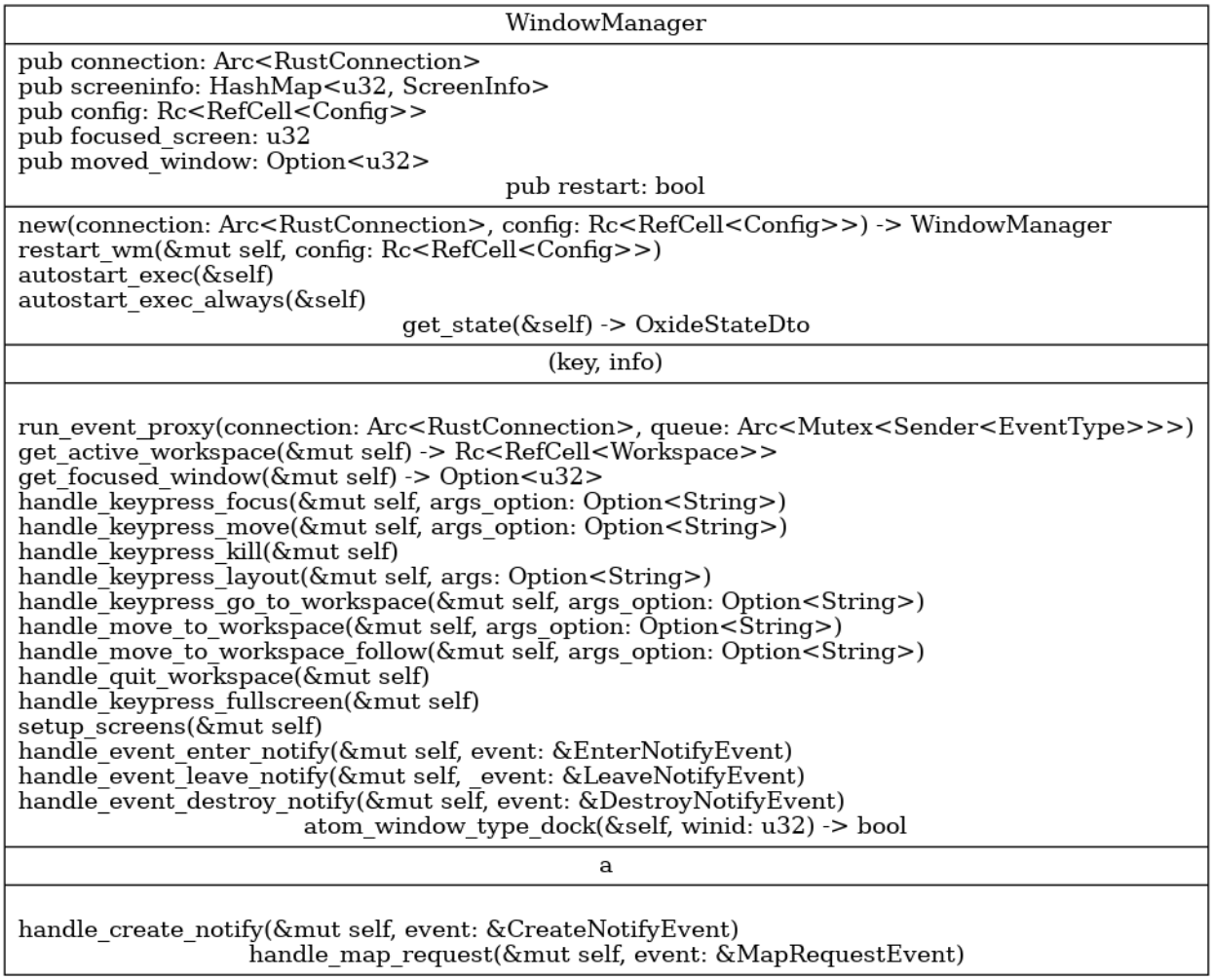

Fig. 33: mod.png

workspace

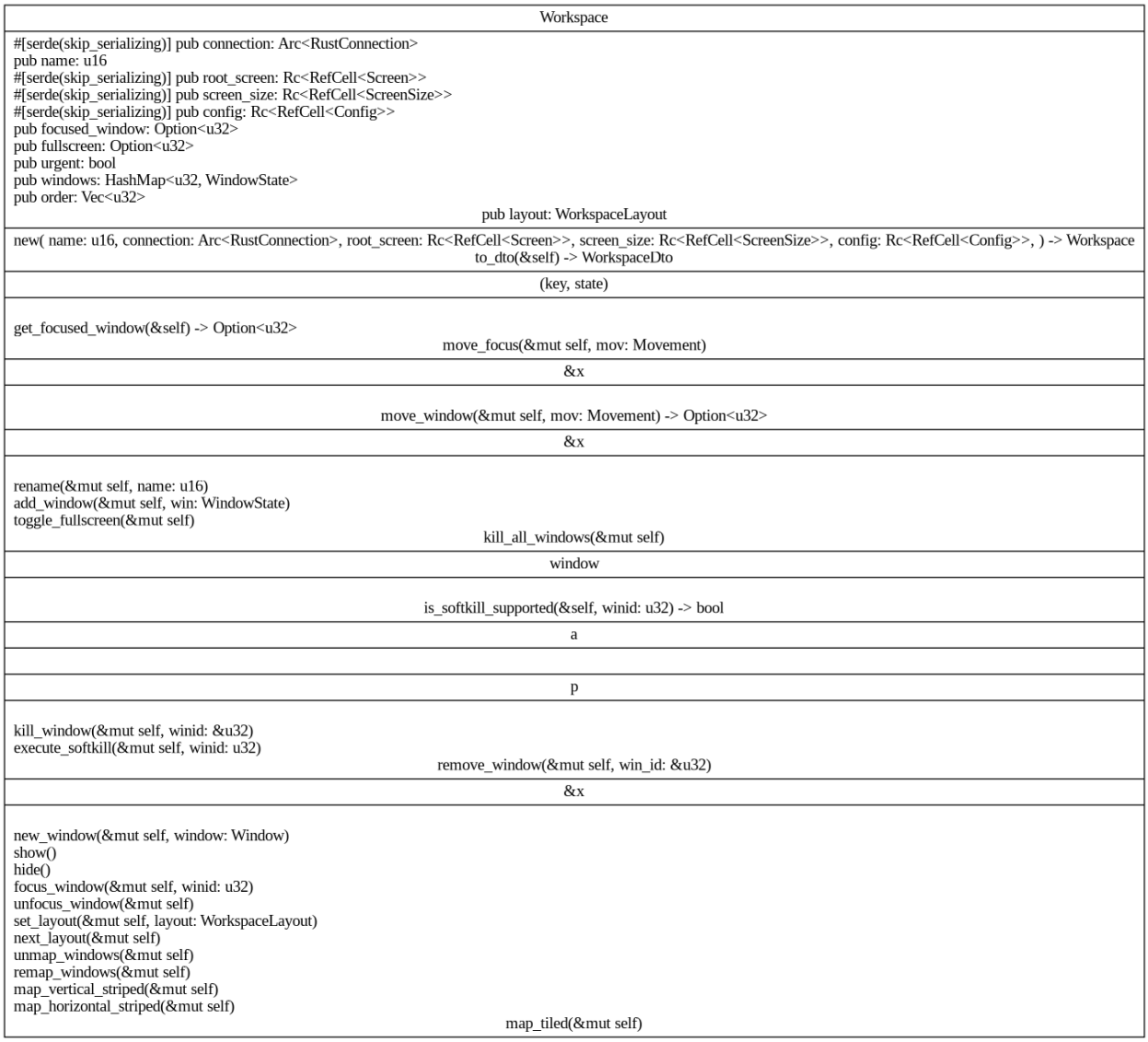

Fig. 34: mod.png

parse error

workspace layout

workspace navigation

Hint: If the diagrams are not shown big enough to read, please click on them.

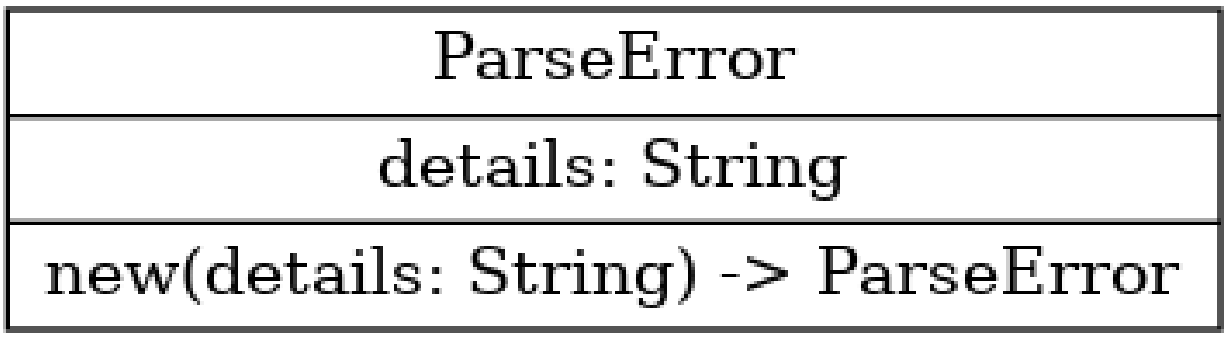

Fig. 35: parse_error.png

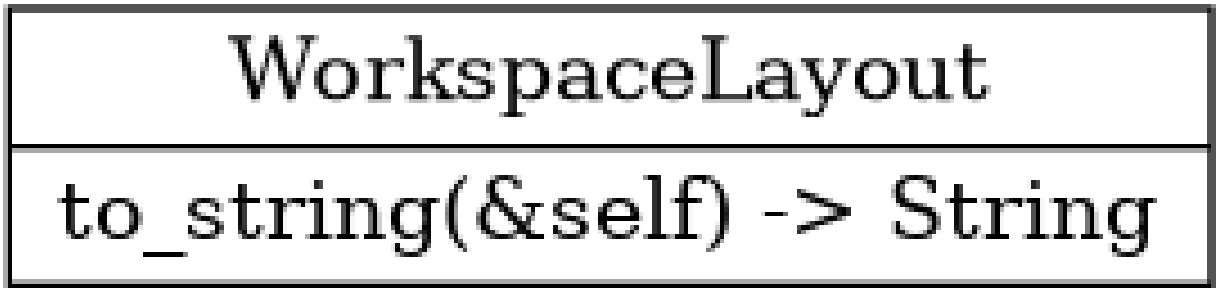

Fig. 36: workspace_layout.png

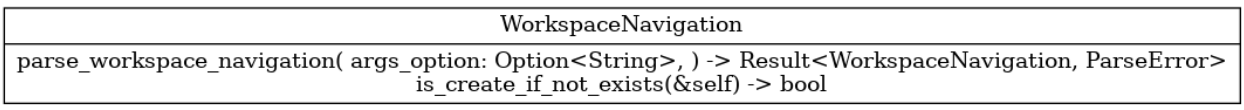

Fig. 37: workspace_navigation.png

auxiliary

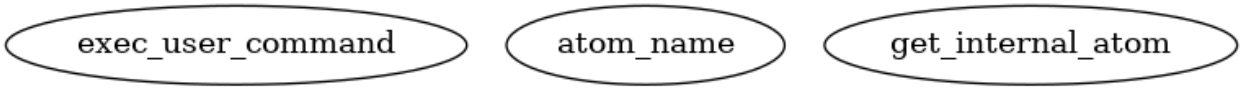

Fig. 38: auxiliary.png

ipc

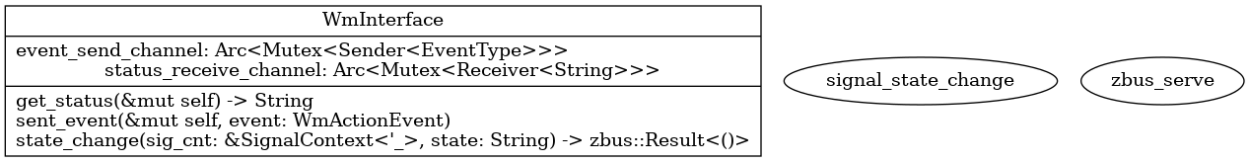

Fig. 39: ipc.png

keybindings

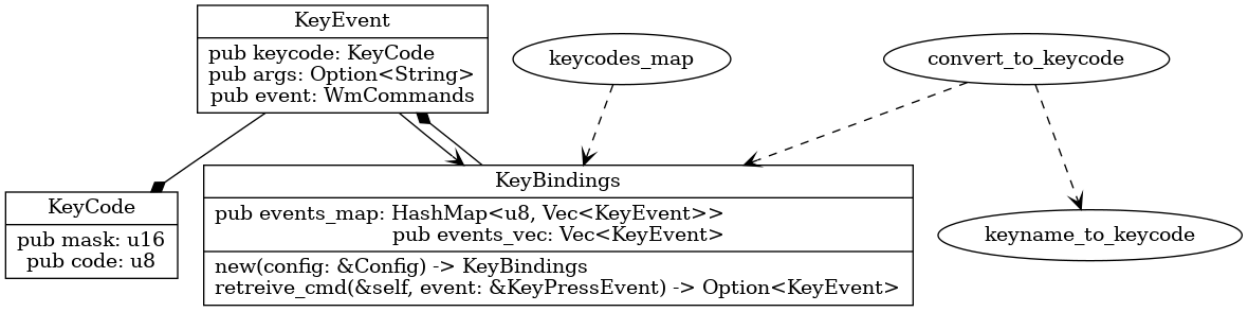

Fig. 40: keybindings.png

main

windowstate

1.4 Using *Oxide*

1.4.1 Installation

Prerequisits

Rust needs to be installed. After it has been installed, restart the terminal session, so that any new environment variables are loaded.

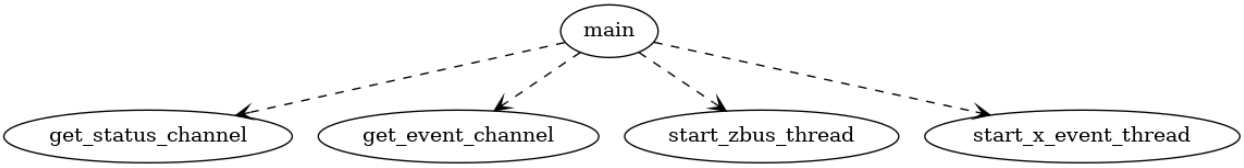

Fig. 41: main.png

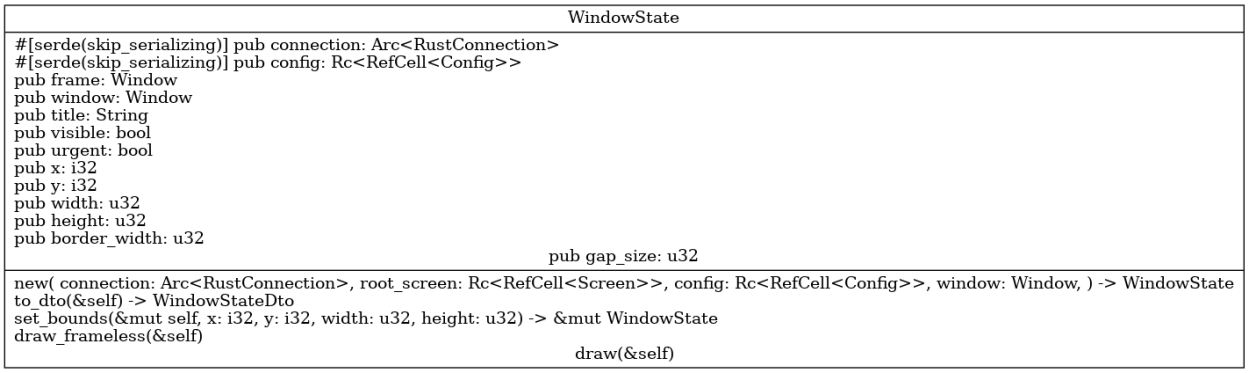

Fig. 42: windowstate.png

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Build tools need to be install:

```
sudo apt install git make build-essential libglib2.0-dev libcairo2-dev libpango1.0-dev␣
↪kitty xterm
```

Installation

1. Clone the *Oxide* git repository:

```
git clone https://github.com/DHBW-FN/OxideWM.git
```

2. Install *Oxide* via make:

```
cd OxideWM
make install
```

Sudo privileges are required to install *Oxide*.

After installation you can quit your current X session and log out. Subsequently *Oxide* should be selectable as window manager in your login screen.

1.4.2 Configuration

Description

Define the behavior of Oxide. The config file provides the possibility to customize e. g. keybindings, layout, style. If the home config file is not existing, default values will be used but commands like *exec* and *exec_always* will not be working. The config file is written in YAML.

Files

During launch, Oxide searches for a config file in the following locations:

Home config file:

```
~/.config/Oxide/config.yml
```

System config file:

```
/etc/Oxide/config.yml
```

Keybindings

Keys

A keybinding has to consist of at least one or more MODIFIERS and exactly one normal key such as 't' for example.

Modifier

M
: Meta key

A
: ALT key

C
: CONTROL key

S
: SHIFT key

Commands

Commands consist of a command and optional arguments.

Commands (COMMAND)

Move [MOVEMENT]
: move window

Focus [MOVEMENT]
: move focus

Quit
: quit the window manager

Kill
: kill the currently focused window

Restart
: reloads the config and restarts components

Layout [LAYOUT]
: change the current layout

GoToWorkspace [WORKSPACE_ARGS]
: change the current workspace

MoveToWorkspace [WORKSPACE_ARGS]
: move the used window to a different workspace

MoveToWorkspaceAndFollow [WORKSPACE_ARGS]
: move the focused window to and select a different workspace

Exec COMMAND
: execute a given command

Fullscreen
: toggle fullscreen mode for the focused window

Arguments (ARGS)

Command arguments are necessary for the movement, the layout or to control workspaces.

Movement (MOVEMENT)

Left
: moves to the left

Right
: moves to the right

Layout (LAYOUT)

VerticalStriped
: windows vertically next to each other

HorizontalStriped
: windows horizontally underneath each other

None
: if no argument is provided, the next layout is chosen

Workspace arguments (WORKSPACE_ARGS)

Next
: Next initialized workspace with a higher index than the current workspace. If the workspace with the highest index is selected, the index with the lowest index will be selected.

Previous
: Next initialized workspace with a lower index than the current workspace. If the workspace with the lowest index is selected, the index with the highest index will be selected.

Next_free
: Next available workspace with which is not initialized. Gaps in the workspace indices are filled first.

Index
: workspace with the given index

Iterations

The iteration commands provide the possibility to change between workspaces when given an iteration number as shown in the example down below.

iter
: iterates over given number in order to change

Default keybindings

Here is a short overview of the default keybindings.

Meta+Shift+e
: quits the window manager

Meta+Shift+r
: restarts the window manager

Meta+Shift+q
: kills the current window

h/l
: direction keys (left/right)

Meta+[DIRECTION]
: changes the focus to the direction window

Meta+Shift+[DIRECTION]
: moves the window to the direction

Meta+f
: changes the current window to fullscreen

Meta+u
: switches to the next layout

Meta+i
: changes the layout to vertical

Meta+Shift+i

changes to layout to horizontal

Right/Left
workspace navigation keys (next/previous)

Meta+[WORKSPACE_DIRECTION]
changes to the workspace direction

Meta+n
opens a new workspace

Control+Meta+[WORKSPACE_DIRECTION]
moves a window to the workspace direction

Control+Meta+n
opens a new workspace and moves the window to it

Meta+Shift+[WORKSPACE_DIRECTION]
moves the window to the workspace direction and follows it

Meta+Shift+n
creates a new workspace, moves the window to it and follows

Control+Meta+Down
quits the workspace

Meta+t
opens dmenu

1/2/3/4/5/6/7/8/9
workspace numbers

Meta+[WORKSPACE_NUMBER]
switches to workspace number

Control+Meta+[WORKSPACE_NUMBER]
moves window to workspace number

Meta+Shift+[WORKSPACE_NUMBER]

moves window to workspace number and follows it

Borders

border_width
sets the border width of windows in pixels

border_color
sets the border color and has to be entered in hexadecimal

border_focus_color
sets the border color for focused windows and has to be entered in hexadecimal

gap
gap between windows in pixels

Execute

exec
one time execution when the window manager starts

exec_always
is executed during start of the window manager and also at each restart

Examples

Keybindings

```
cmds:
 - keys: ["M", "t"]
 commands:
     - command: Exec
       args: "dmenu"
```

In this example pressing the meta key and 't', a new dmenu window is opened.

Iterations

```
iter_cmds:
 - iter: [1, 2, 3, 4, 5, 6, 7, 8, 9]
 command:
   keys: ["M", "C", "$VAR"]
   commands:
        - command: GoToWorkspace
          args: "$VAR"
```

In this example using the **ALT** and **CONTROL** key paired with a number from one to nine, the user can go to the desired workspace. `$VAR` is a reference for the entered iterator.

Bugs

Please open an issue https://github.com/DHBW-FN/OxideWM/issues .

1.4.3 Configuration of statusbar

Description

Define the behavior of the statusbar for *Oxide*. The config file provides the possibility to customize the text and background color of the *Oxide* statusbar. The config file is written in YAML.

Files

During launch, *Oxide* bar searches for a statusbar config file in the following two locations.

Home config file:

```
~/.config/Oxide/bar_config.yml
```

System config file:

```
/etc/Oxide/bar_config.yml
```

Color

In order to configure the colors, they have to be entered in hexadecimal. If the colors are not defined, default values will be used.

Examples

```
color_bg: "0x008000" # green
color_txt: "0xFFFF00" # black
```

Bugs

Please open an issue https://github.com/DHBW-FN/OxideWM/issues .

1.4.4 Configuration of *Oxide* msg

Synopsis

oxide-msg [**-h**] | [**-v**] | [**-c** command] [**-a** argument]

Description

The *oxide-msg* is an IPC command tool allowing querying and messaging to Oxide via the commandline.

Options

-a, **–argument** [ARGUMENT]
: arguments to specify command behavior

-c, **–command** [WM_COMMAND]
: window manager commands

-h, **–help**
: output help message and exit

-v, **–version**
: output version information and exit

WM commands (WM_COMMAND)

Move -a [MOVEMENT]
: move window

Focus -a [MOVEMENT]
: move focus

Quit
: quit the window manager

Kill
kill the currently focused window

Restart
reloads the config and restarts components

Layout -a [LAYOUT]
change the current layout

GoToWorkspace -a [WORKSPACE_ARGS]
change the current workspace

MoveToWorkspace -a [WORKSPACE_ARGS]
move the focused window to a different workspace

MoveToWorkspaceAndFollow -a [WORKSPACE_ARGS]
move the focused window to and select a different workspace

Exec -a [COMMAND]
execute a given command

Fullscreen
toggle fullscreen mode for the focused window

Movement (MOVEMENT)

Left
moves to the left

Right
moves to the right

Layout (LAYOUT)

Vertical
windows vertically next to each other

Horizontal
windows horizontally underneath each other

None
: if no argument is provided, the next layout is chosen

Workspace arguments (WORKSPACE_ARGS)

Next
: Next initialized workspace with a higher index than the current workspace. If the workspace with the highest index is selected, the index with the lowest index will be selected.

Previous
: Next initialized workspace with a lower index than the current workspace. If the workspace with the lowest index is selected, the index with the highest index will be selected.

Next_free
: Next available workspace with which is not initialized. Gaps in the workspace indices are filled first.

Index
: workspace with the given index

Examples

```
cargo run -p oxide-msg -- -c "exec" -a "kitty"
cargo run -p oxide-msg -- --command "kill"
```

Bugs

Please open an issue https://github.com/DHBW-FN/OxideWM/issues .