ADC23

ILIAS BERGSTRÖM

THE ARCHITECTURE OF DIGITAL AUDIO WORKSTATIONS (AND OTHER TIME-BASED MEDIA SOFTWARE)

Presenter

Ilias Bergström

Work at Elk Audio

Background in Research

Elk Audio

Embedded Linux OS for audio

- Off-the-shelf SOCs (ARM and x86)
- Less than 1ms roundtrip latency
- With all the benefits of Linux

Clients I cannot mention.

The Nina Synthesizer uses the open-source license of the OS, and is an awesome synth!

Elk LIVE

Enables playing together live with low latency

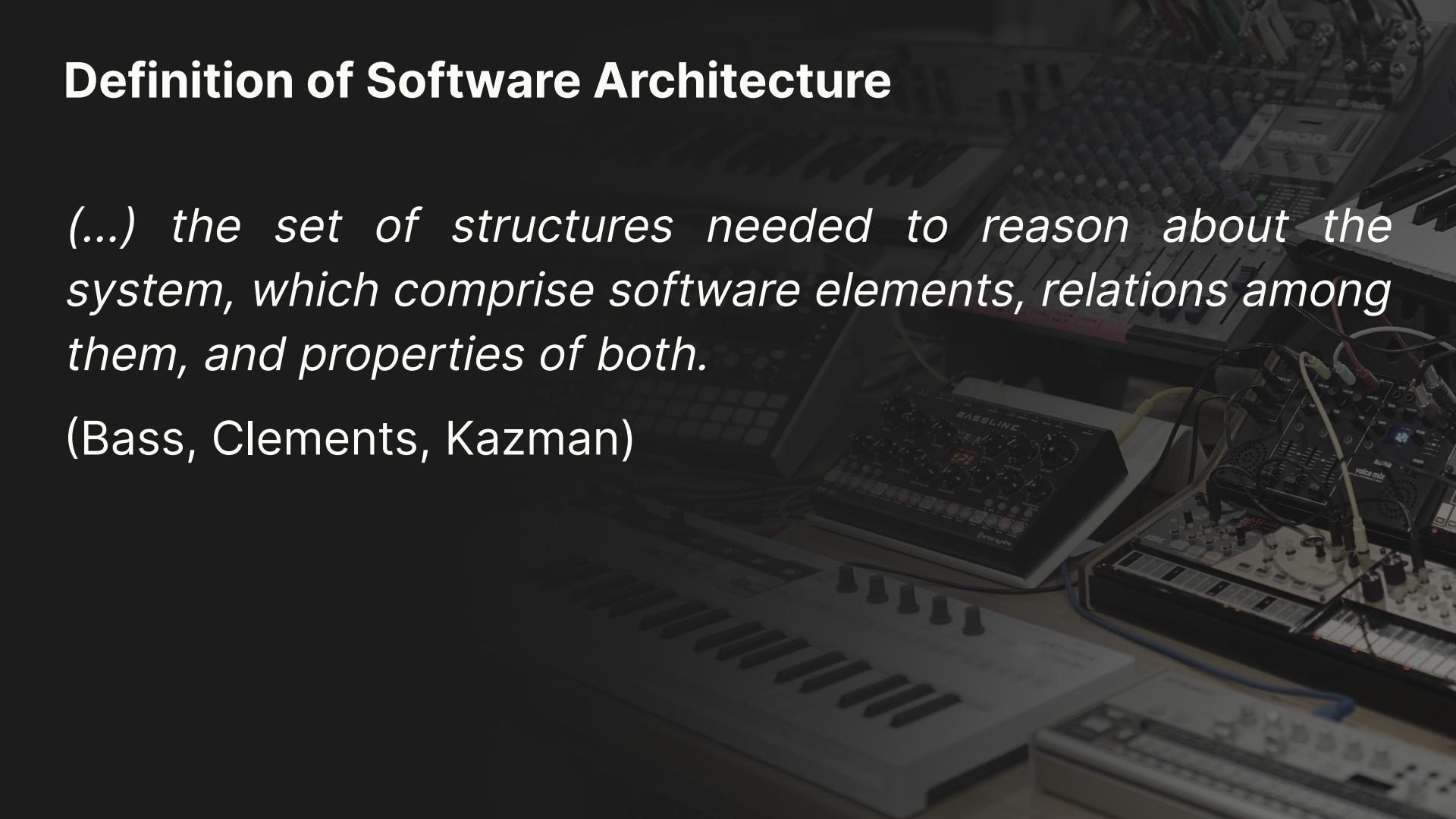
Mac OS app available - Windows coming

Software Architecture

Definition of Software Architecture

(...) the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.

(Bass, Clements, Kazman)



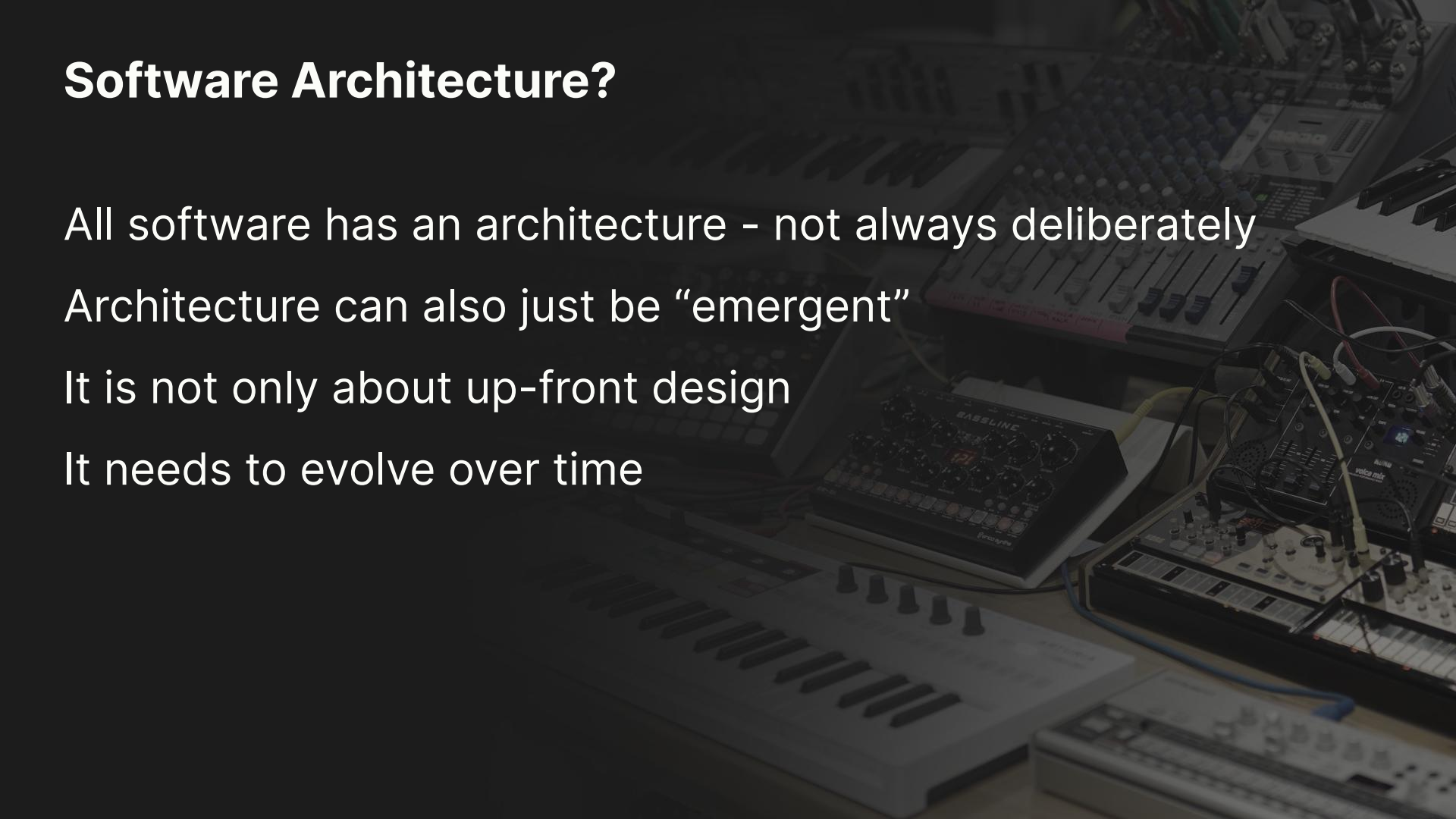
Software Architecture?

All software has an architecture - not always deliberately

Architecture can also just be “emergent”

It is not only about up-front design

It needs to evolve over time



Overview

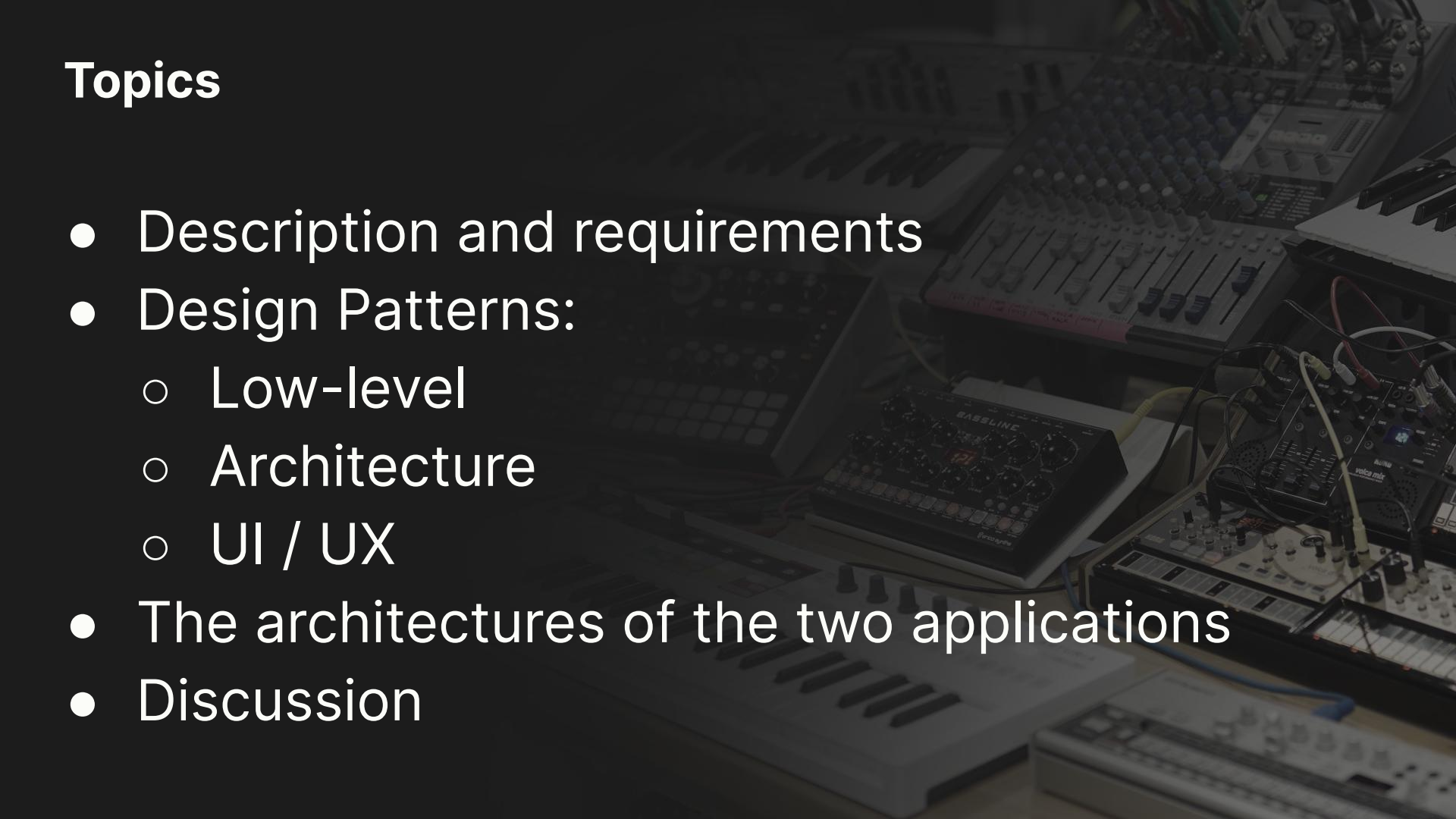
- Architecture has not been covered frequently at ADC.

- I chose a broad scope:
  - There should be something both for juniors and veterans.

- Please don’t try to read details in diagrams during the talk.
  - There won’t be time.

Topics

- Description and requirements
- Design Patterns:
  - Low-level
  - Architecture
  - UI / UX
- The architectures of the two applications
- Discussion



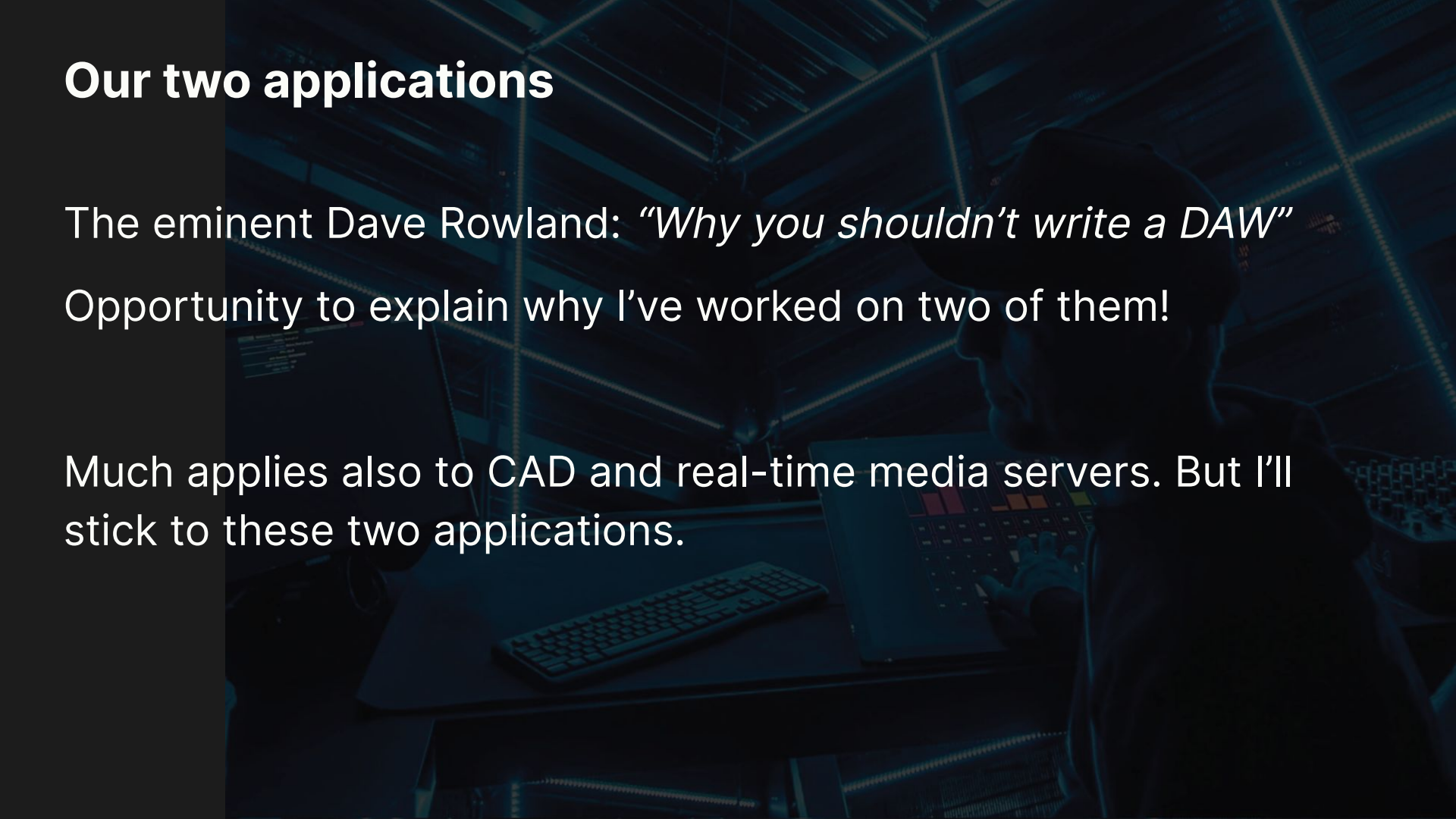
The two applications

Our two applications

The eminent Dave Rowland: *“Why you shouldn’t write a DAW”*

Opportunity to explain why I’ve worked on two of them!

Much applies also to CAD and real-time media servers. But I’ll stick to these two applications.



Sushi

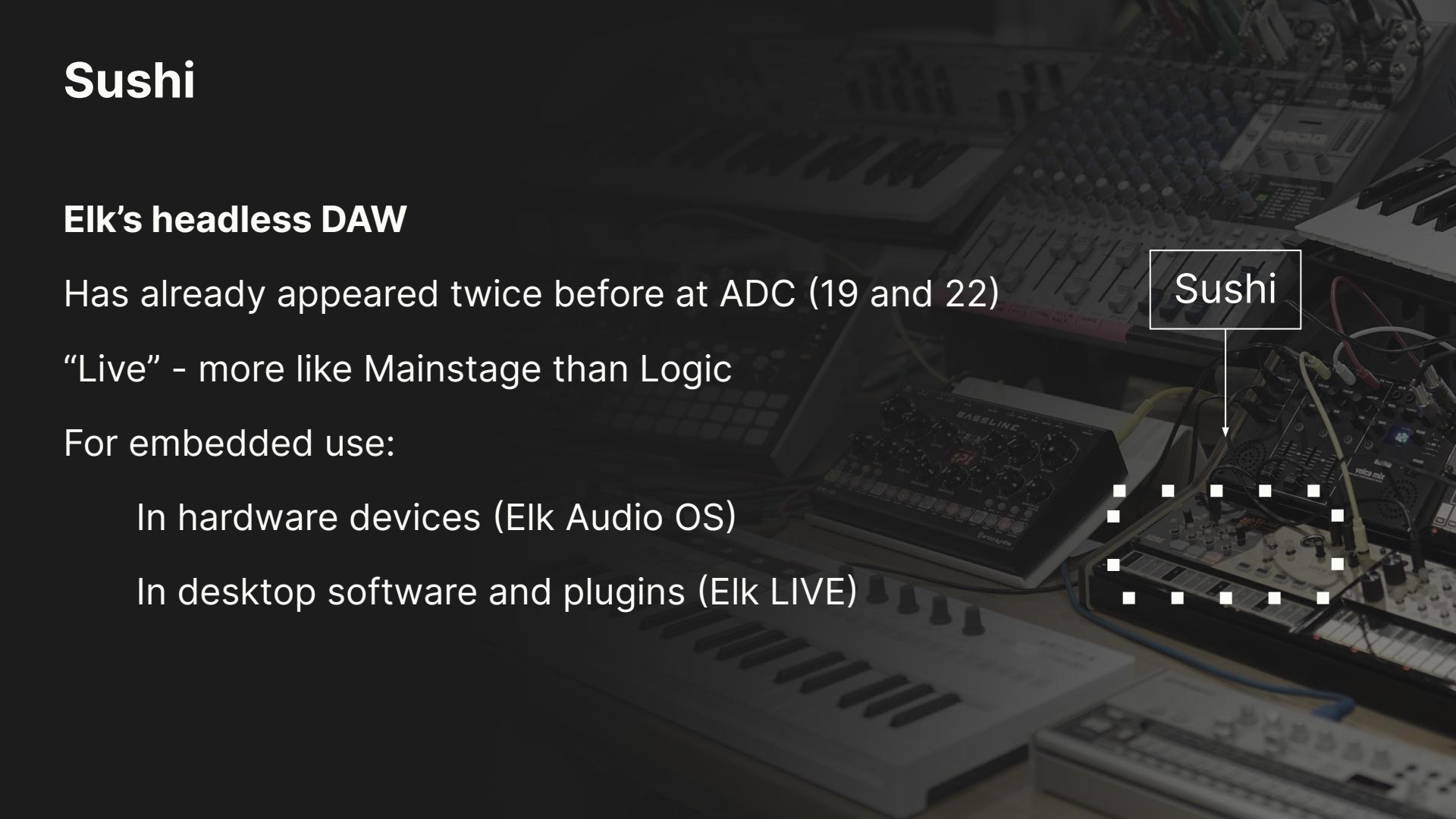
Elk's headless DAW

Has already appeared twice before at ADC (19 and 22)

"Live" - more like Mainstage than Logic

For embedded use:

- In hardware devices (Elk Audio OS)
- In desktop software and plugins (Elk LIVE)



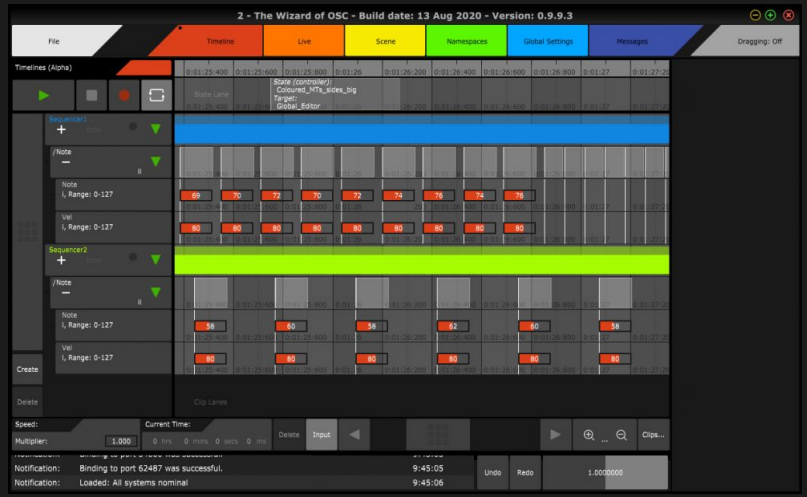
Not detailed before is TWO, a “Media Control Workstation”

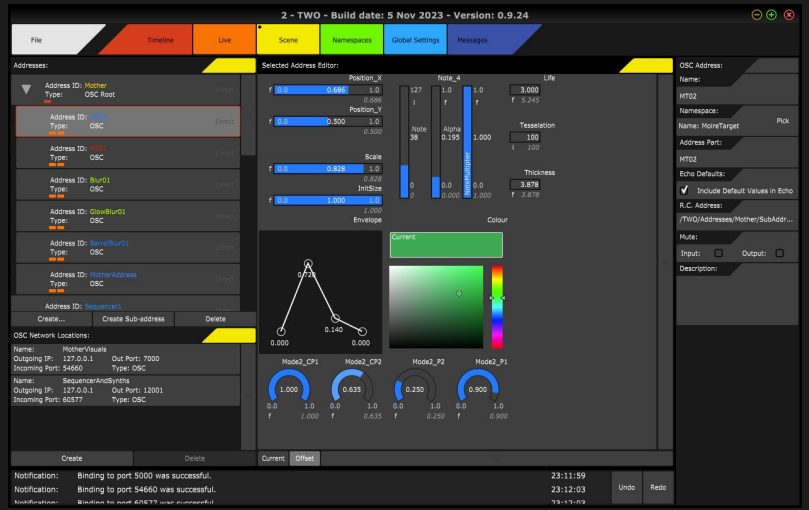
My “passion project”

Like a DAW - but for all types of media control signals.

MIDI, and for show control, etc.

Uses JUCE



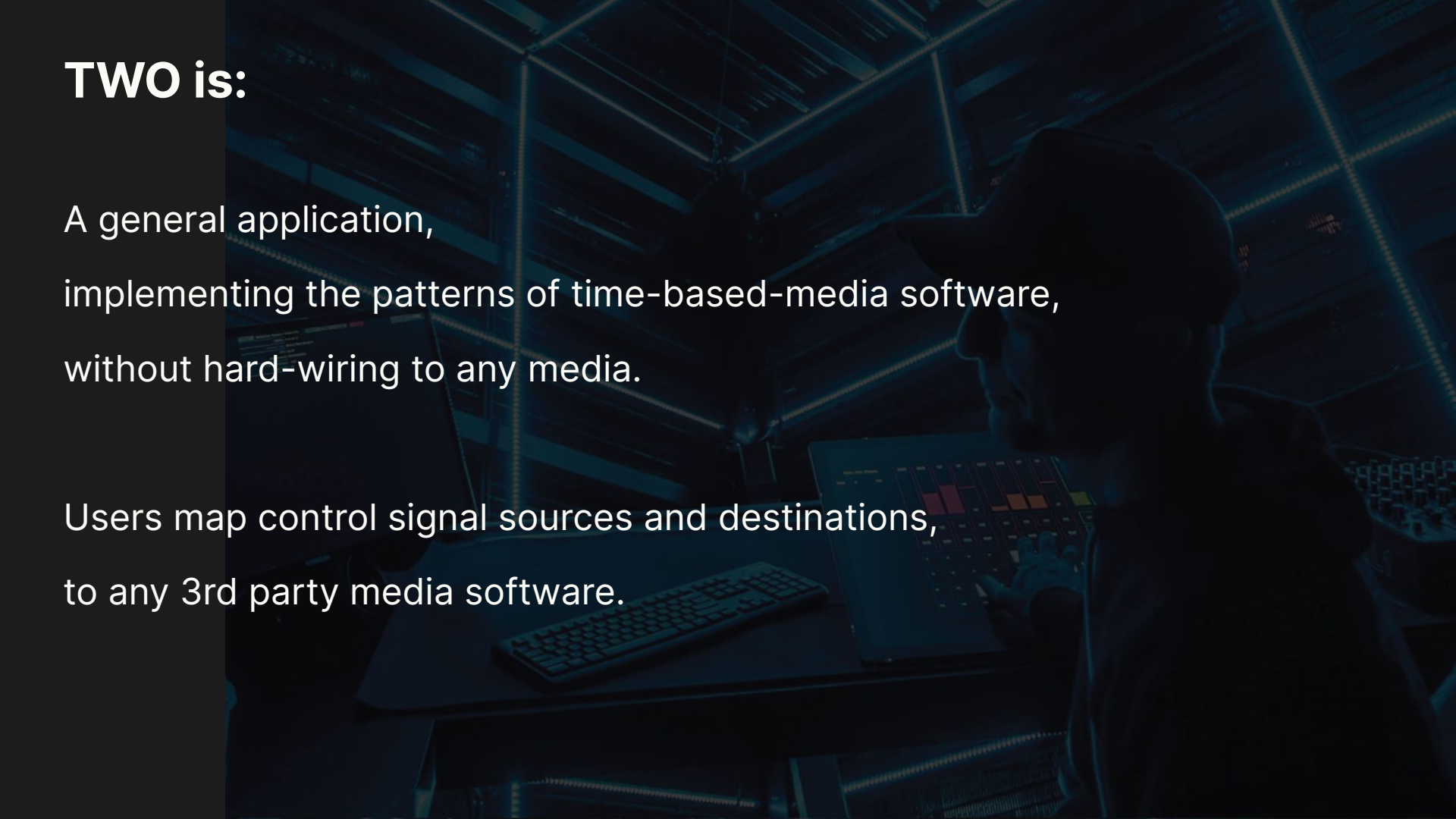


Time-based-media software?

They share common design patterns, forming a “genre”.

Main difference:

The media and algorithms that control signals affect.



TWO is:

A general application,

implementing the patterns of time-based-media software,

without hard-wiring to any media.

Users map control signal sources and destinations,

to any 3rd party media software.

Creatives already use:

Sequencer applications

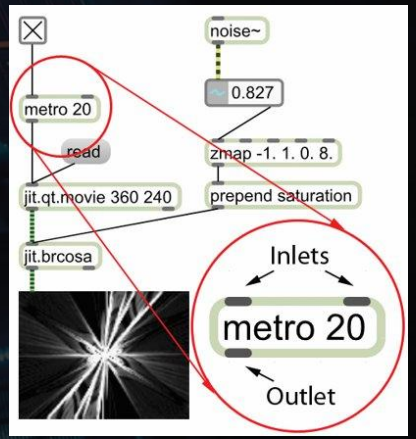
to record, play, and interpolate.

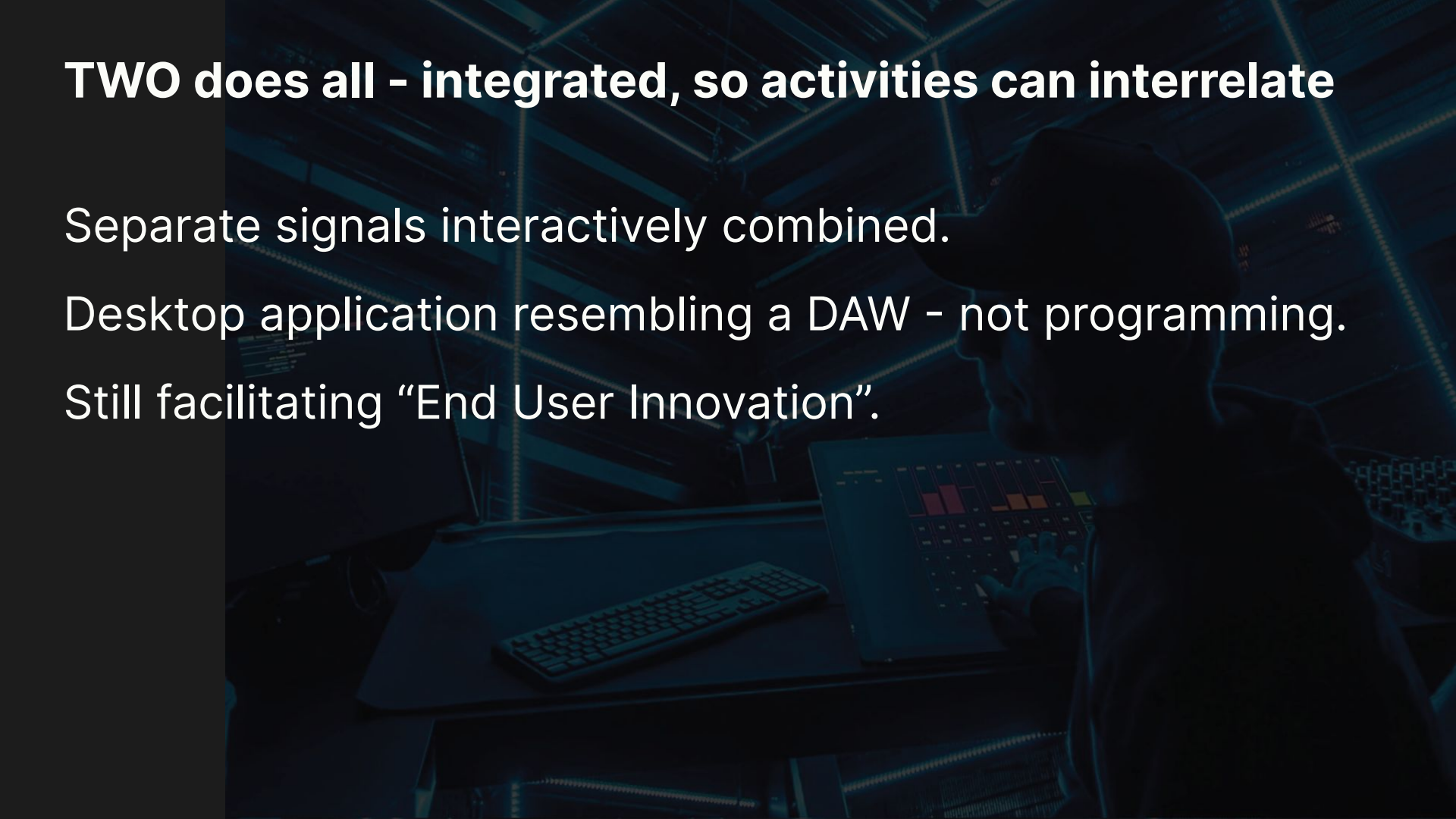
Control apps

to build UI's for sending signals.

Data-flow programs

to map signals from sources to destinations.





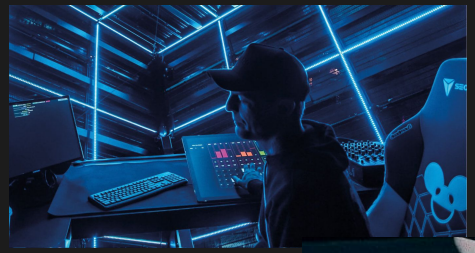
TWO does all - integrated, so activities can interrelate

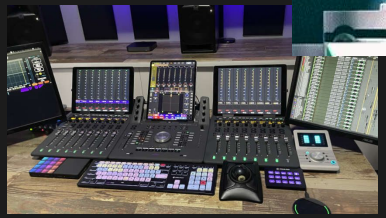
Separate signals interactively combined.

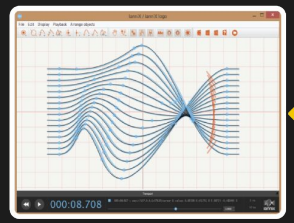
Desktop application resembling a DAW - not programming.

Still facilitating “End User Innovation”.

End-User Innovation



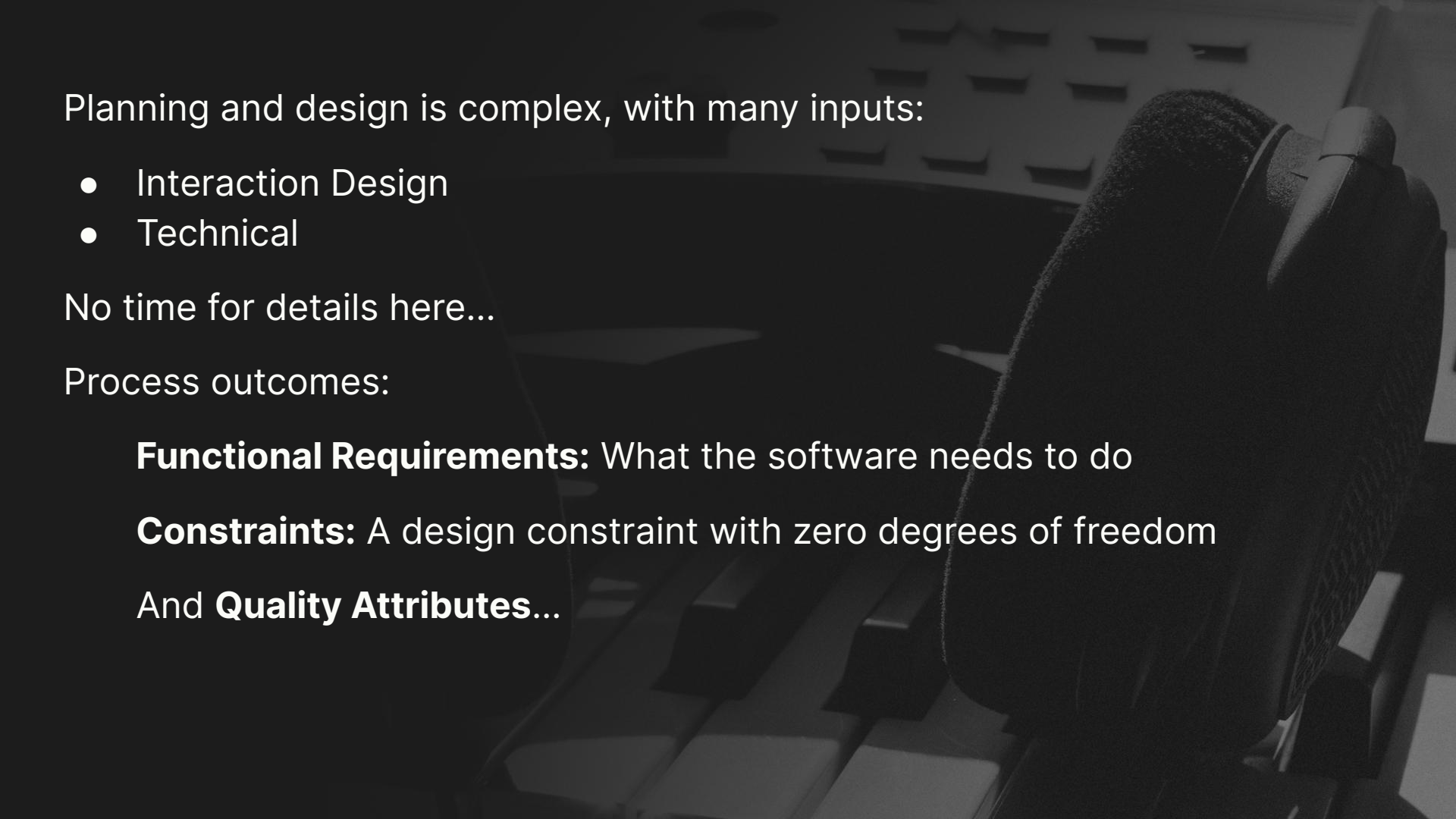




AUDIO
MIDI
AUDIO
AUDIO
OSC
MIDI
OSC
OSC
VIDEO

Users adapt existing tools for their unique needs

Software Requirements and Challenges



Planning and design is complex, with many inputs:

- Interaction Design
- Technical

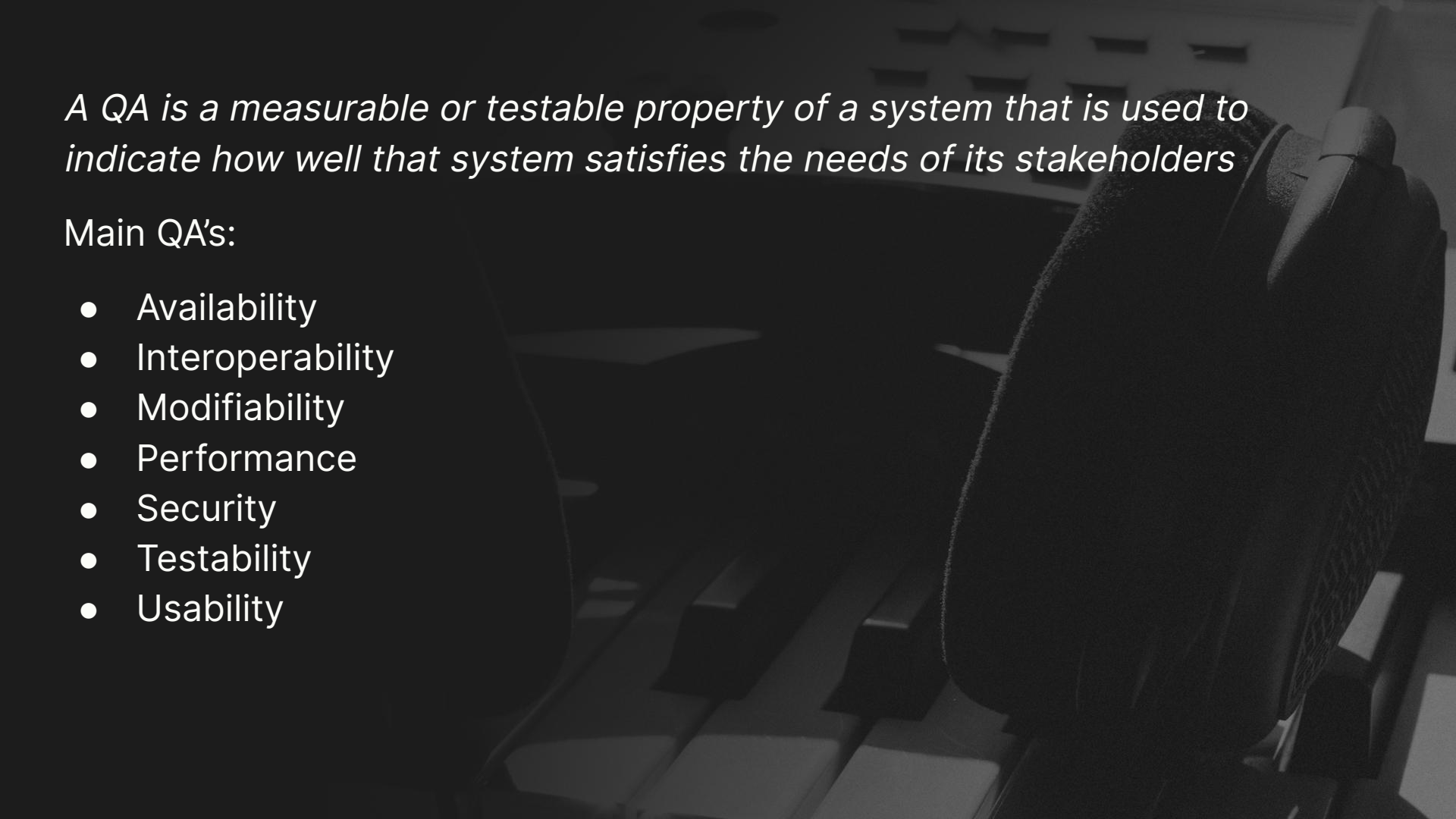
No time for details here…

Process outcomes:

Functional Requirements: What the software needs to do

Constraints: A design constraint with zero degrees of freedom

And **Quality Attributes**…



A QA is a measurable or testable property of a system that is used to indicate how well that system satisfies the needs of its stakeholders

Main QA's:

- Availability
- Interoperability
- Modifiability
- Performance
- Security
- Testability
- Usability

Sushi Requirements

Remote Procedure Call (RPC) API's

Dynamic audio graph

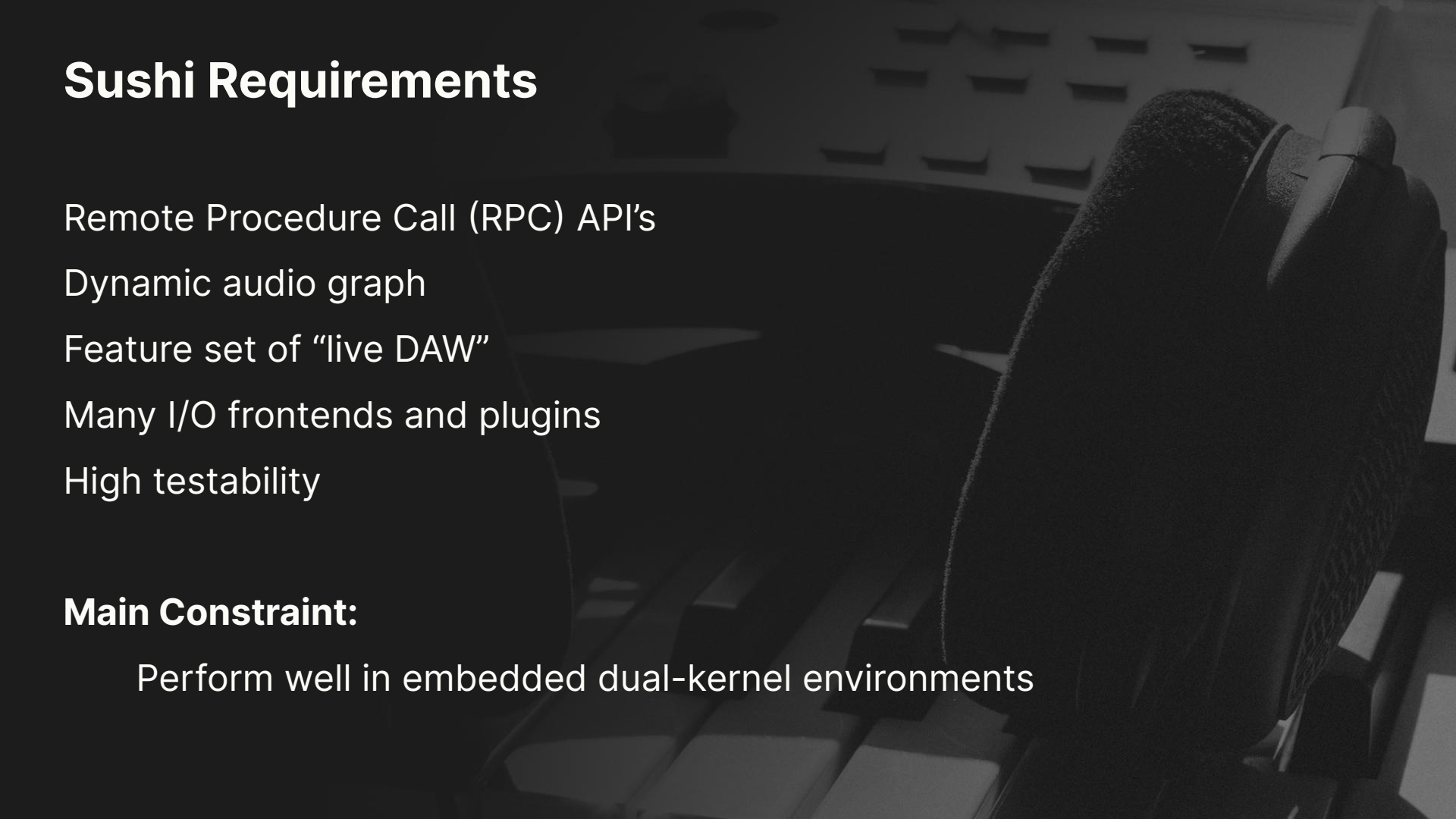
Feature set of "live DAW"

Many I/O frontends and plugins

High testability

Main Constraint:

Perform well in embedded dual-kernel environments



TWO Requirements

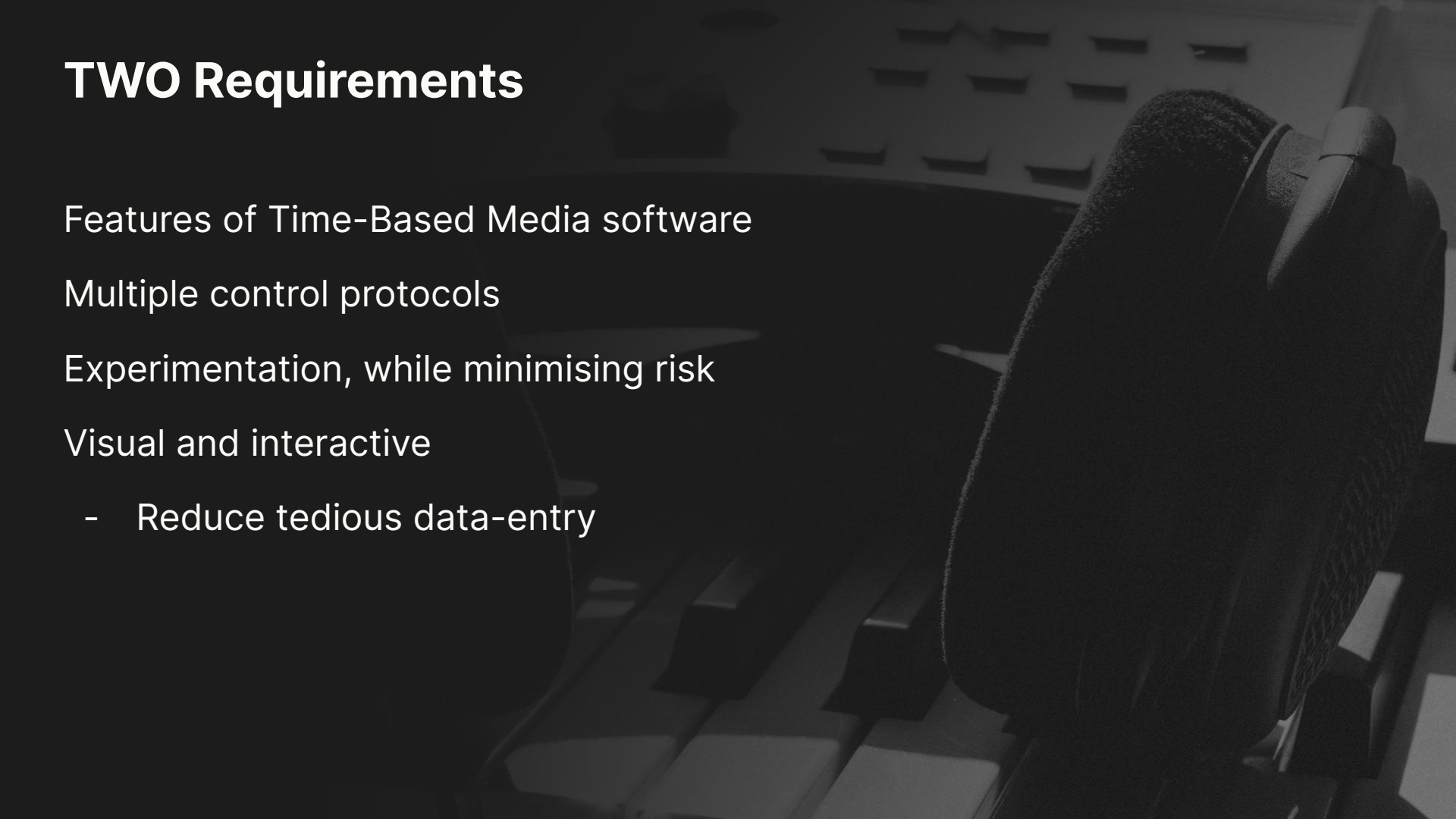
Features of Time-Based Media software

Multiple control protocols

Experimentation, while minimising risk

Visual and interactive

- Reduce tedious data-entry



Software Design Patterns

Software Design Patterns

Low-level building blocks:

“General, reusable solutions to commonly occurring problems”

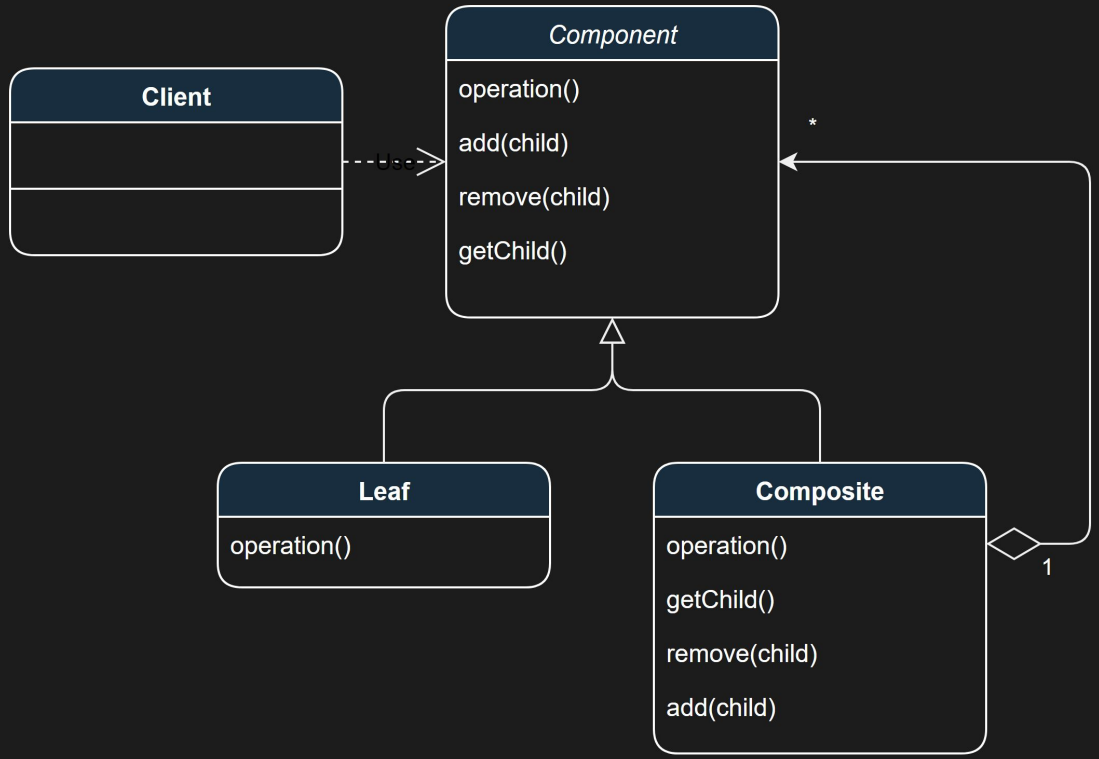
“Design Patterns” book (Gamma et al - the Gang of Four).

Ubiquitous today:

std library and JUCE probably implement every single one.

Next: A quick glance over fundamental patterns used in TWO and Sushi.

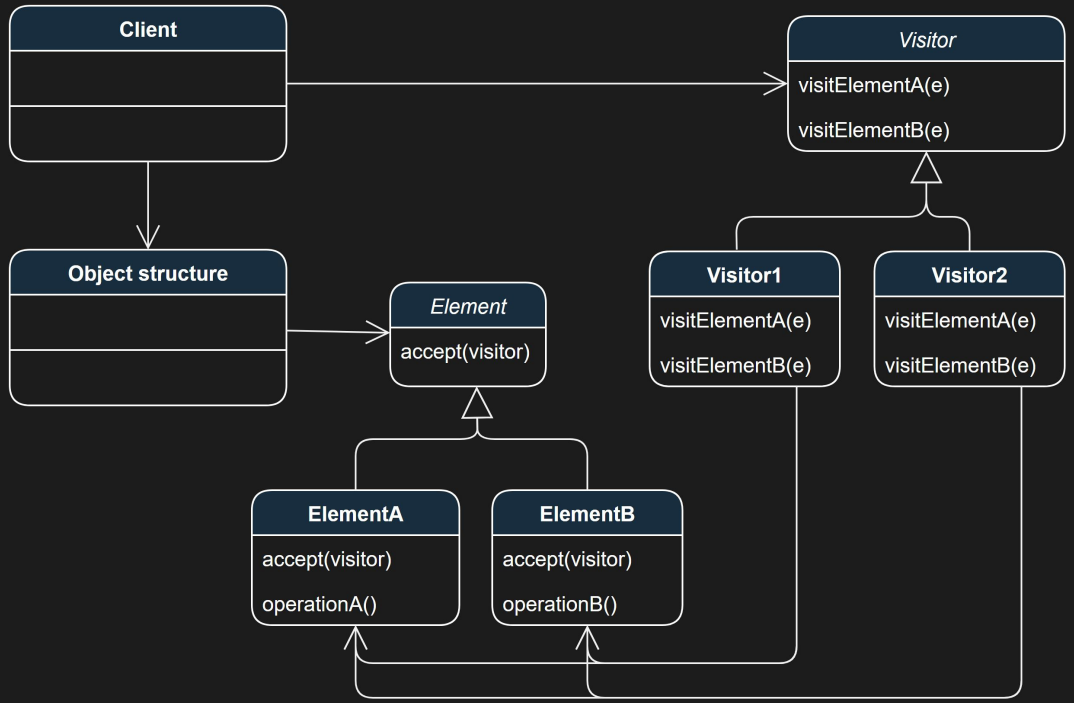
Composite Pattern



Composes objects into tree structures, representing part-whole hierarchies.

Lets clients treat individual objects and compositions uniformly.

Visitor Pattern

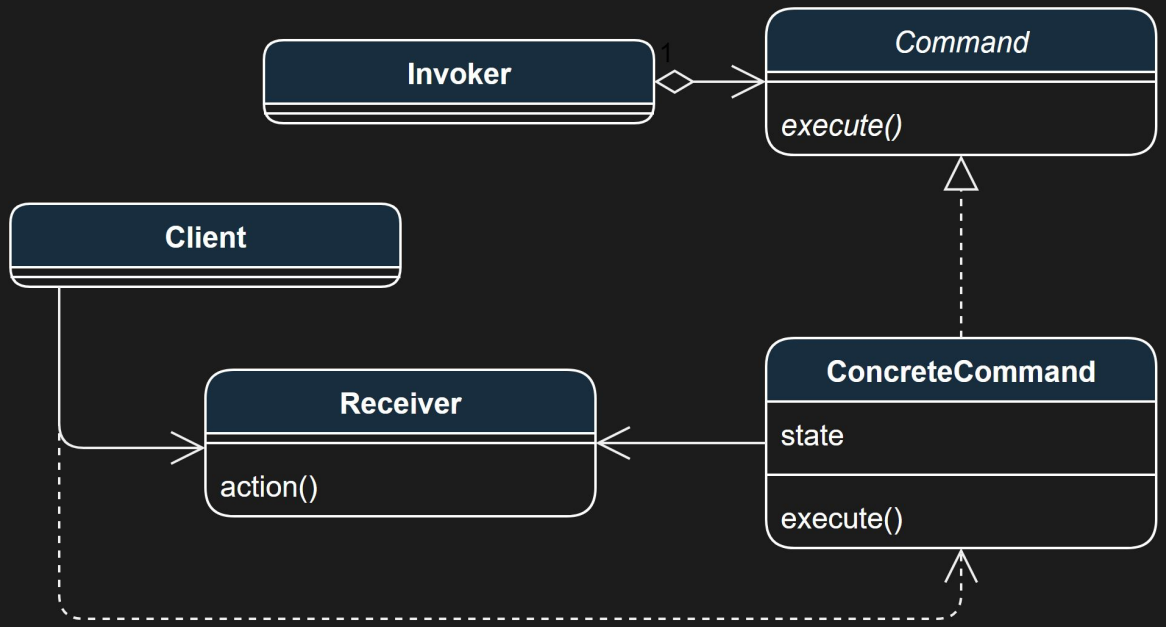


Represents an operation to be performed on the elements of e.g. a Composite structure.

Original GoF is verbose and coupled.

E.g. Klaus Iglberger proposed a modernisation at CppCon 22

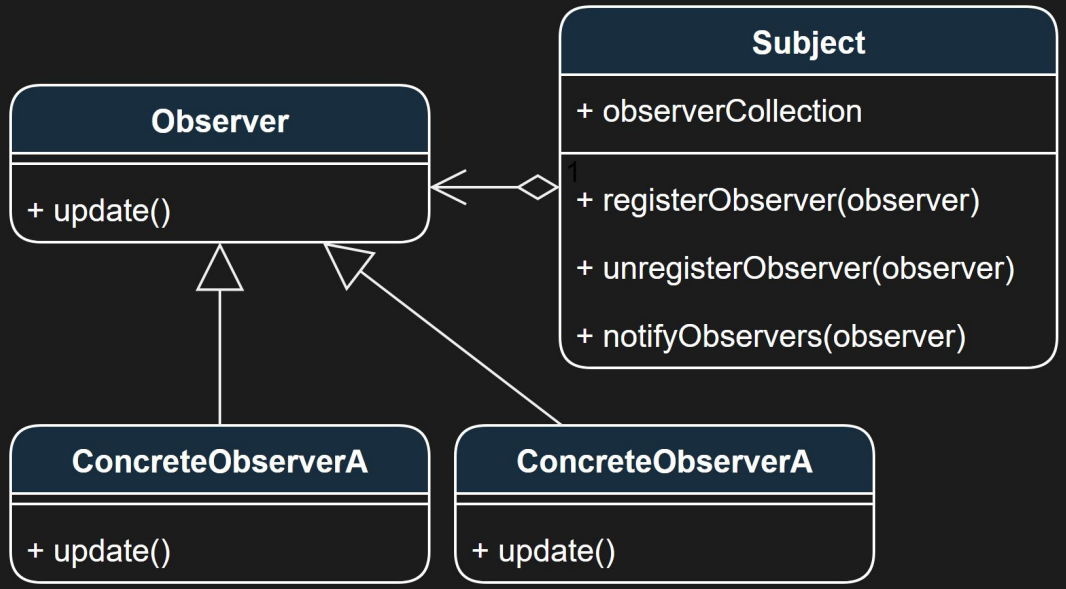
Command Pattern



Encapsulates a request or operation as an object.

Allows implementing full Undo-Redo.

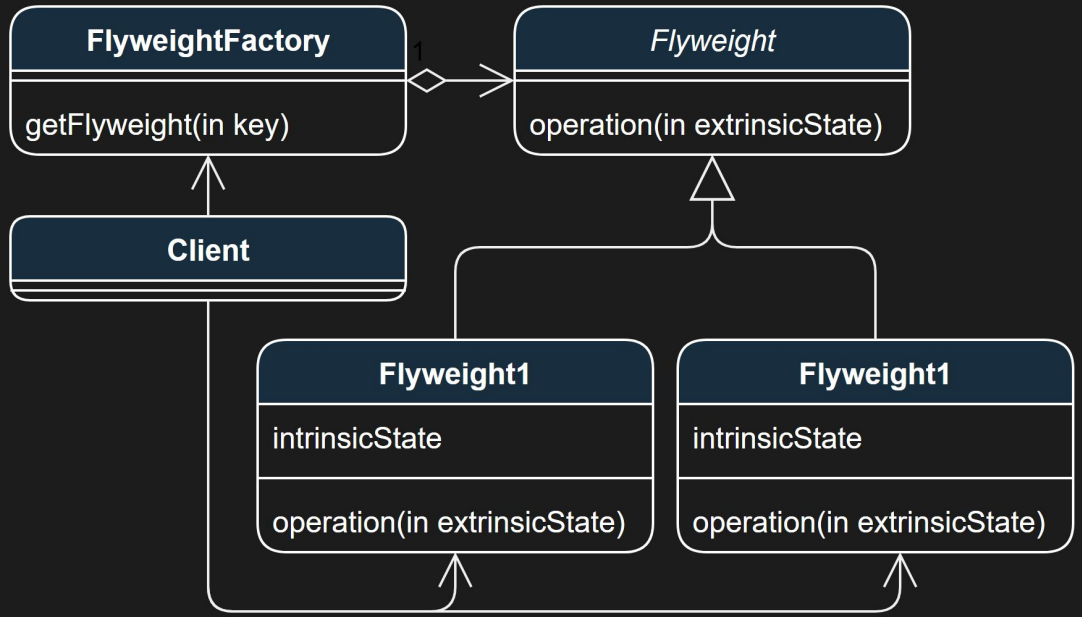
Observer Pattern



Defines a one-to-many dependency between objects:

so when one changes state, all dependents are notified.

Flyweight Pattern



Allows repetitions of similar objects efficiently - through sharing.

Design Patterns work best together!

A composite structure, altered, serialised, and displayed by visitors, modified with commands, with instance support through flyweight, and kept in sync using observers. Behind a Facade :)

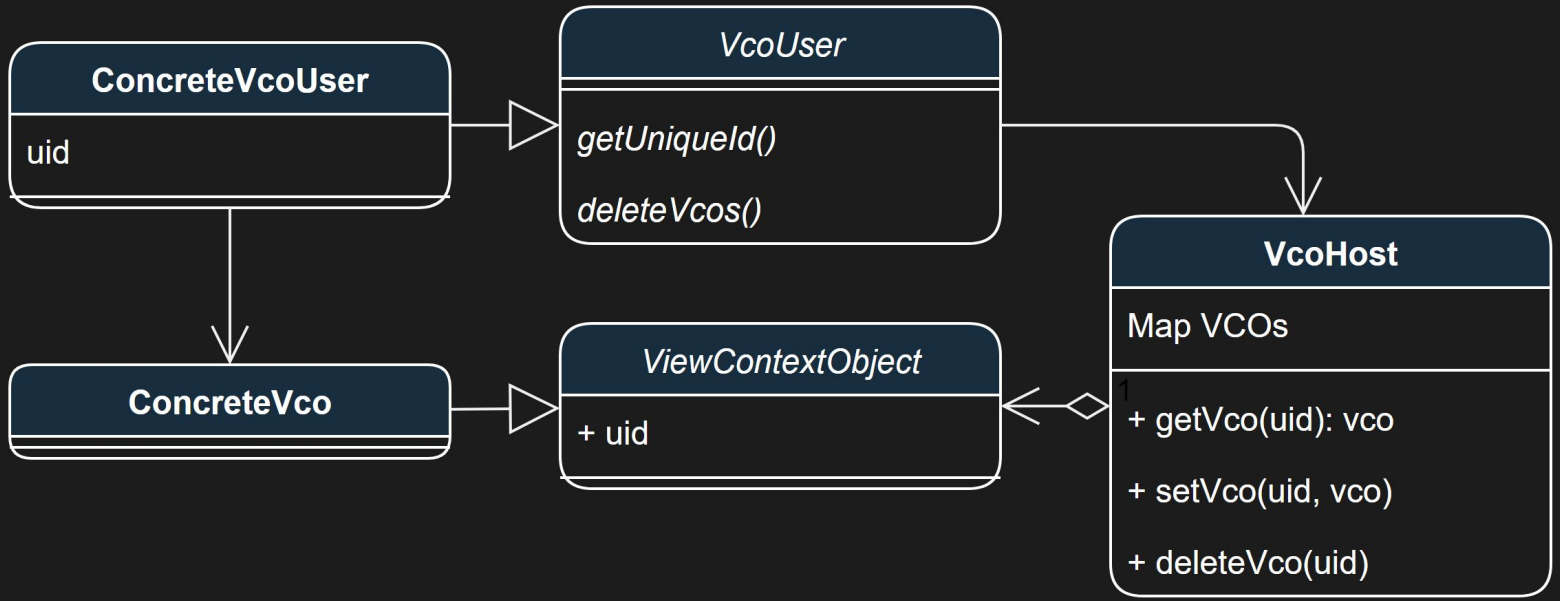
Juce ValueTrees combine these patterns into a bigger abstraction.

Dave Rowland gave a fantastic talk at ADC ‘17.

View Context Object (my own design)

A Visitor can store state inside an existing structure,

instead of duplicating the structure.



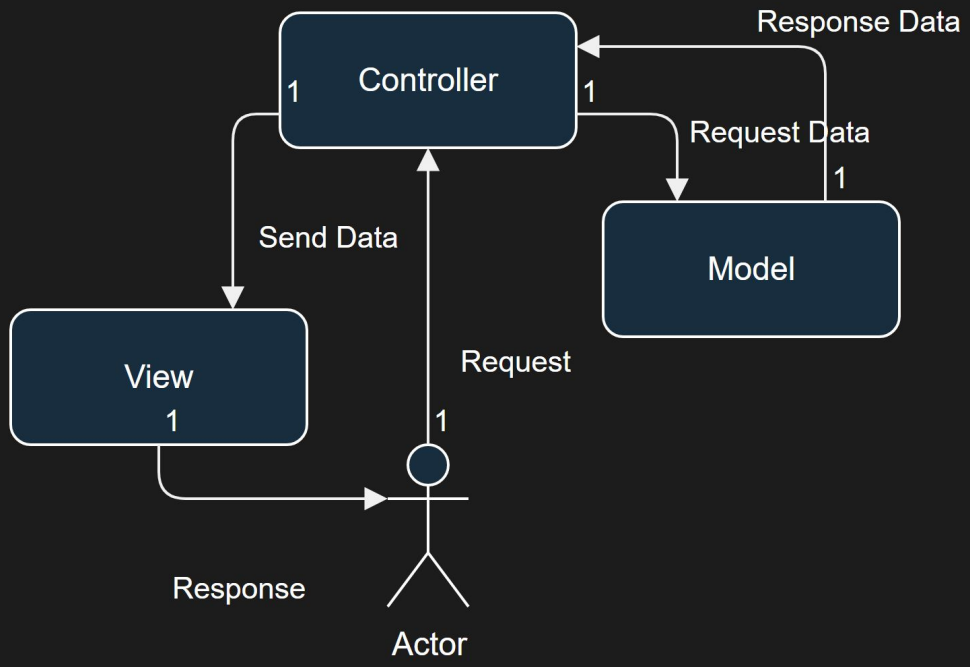
Sushi and TWO use many more Design Patterns

- Abstract Factory
- Factory Method
- Dependency Injection
- Adapter
- Facade
- Iterator
- …

But there’s no time to cover more here!

Architectural Design Patterns

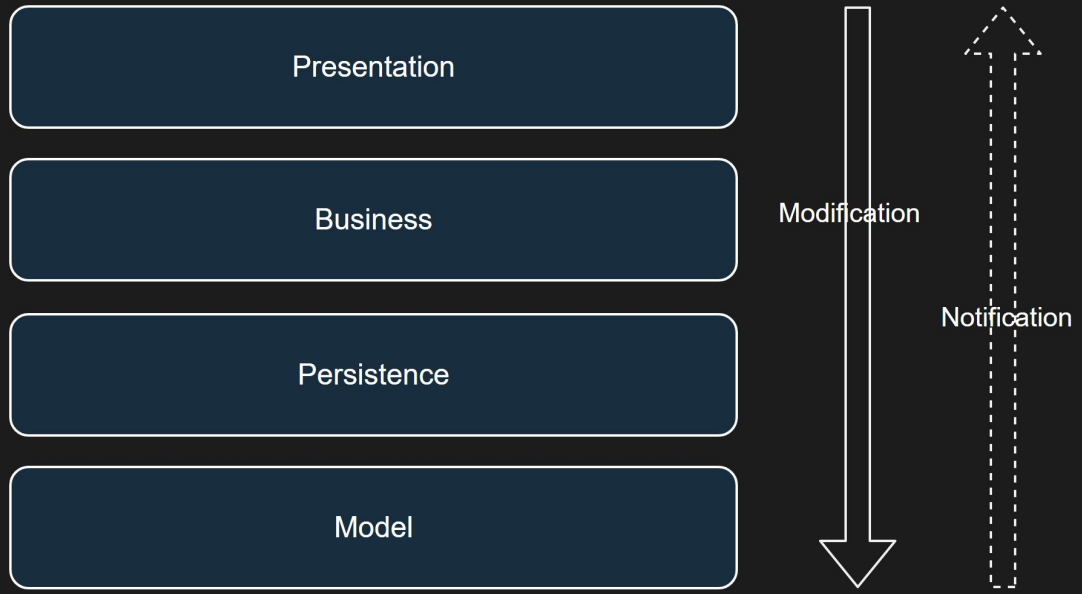
Model View Controller



MVC allows separating internal representations of information, from how it’s presented to and accepted from the user.

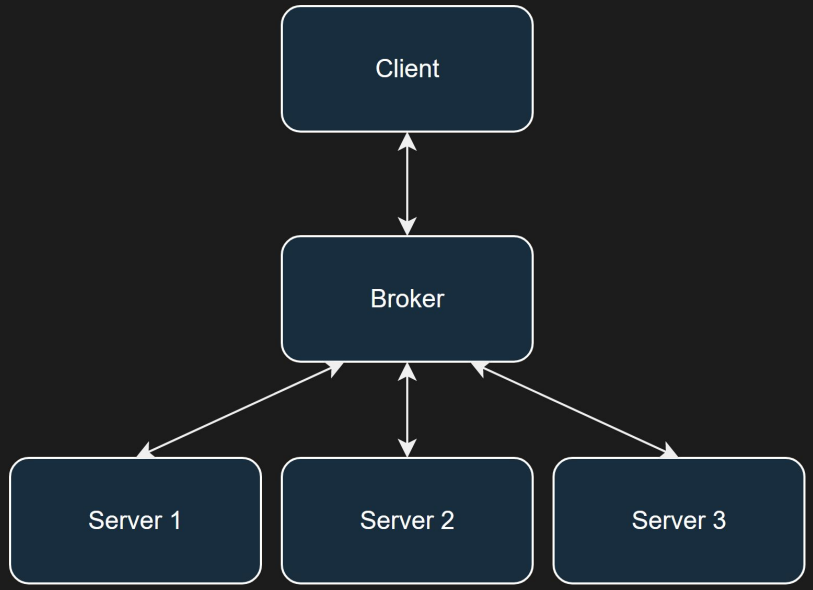
* Variations: HMVC, MVA, MVP, MVVM, HMR&C

Layered Architecture



Allows e.g. presentation, application processing, and data management functions, to be logically separated.

Server-Client, and Broker



For decoupled components in distributed systems, interacting over RPC.

Server-Client alone is for 1-to-1 mappings.

A Broker is needed when there are several different Servers involved.

UI and Experience Design Patterns

UI and Experience Design Patterns

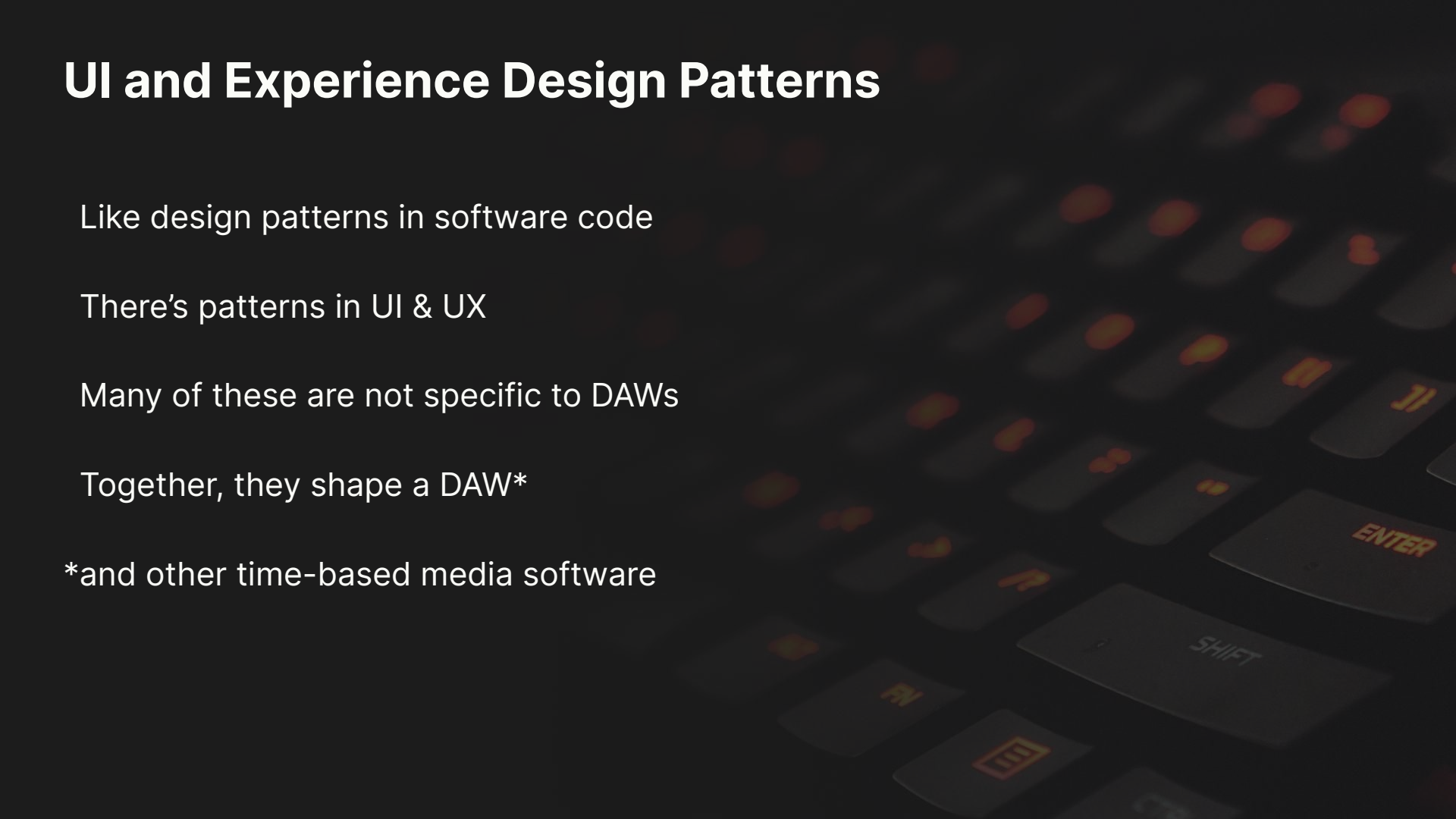
Like design patterns in software code

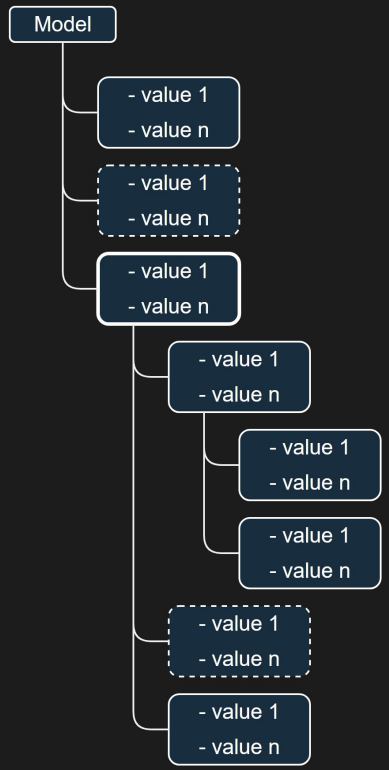
There’s patterns in UI & UX

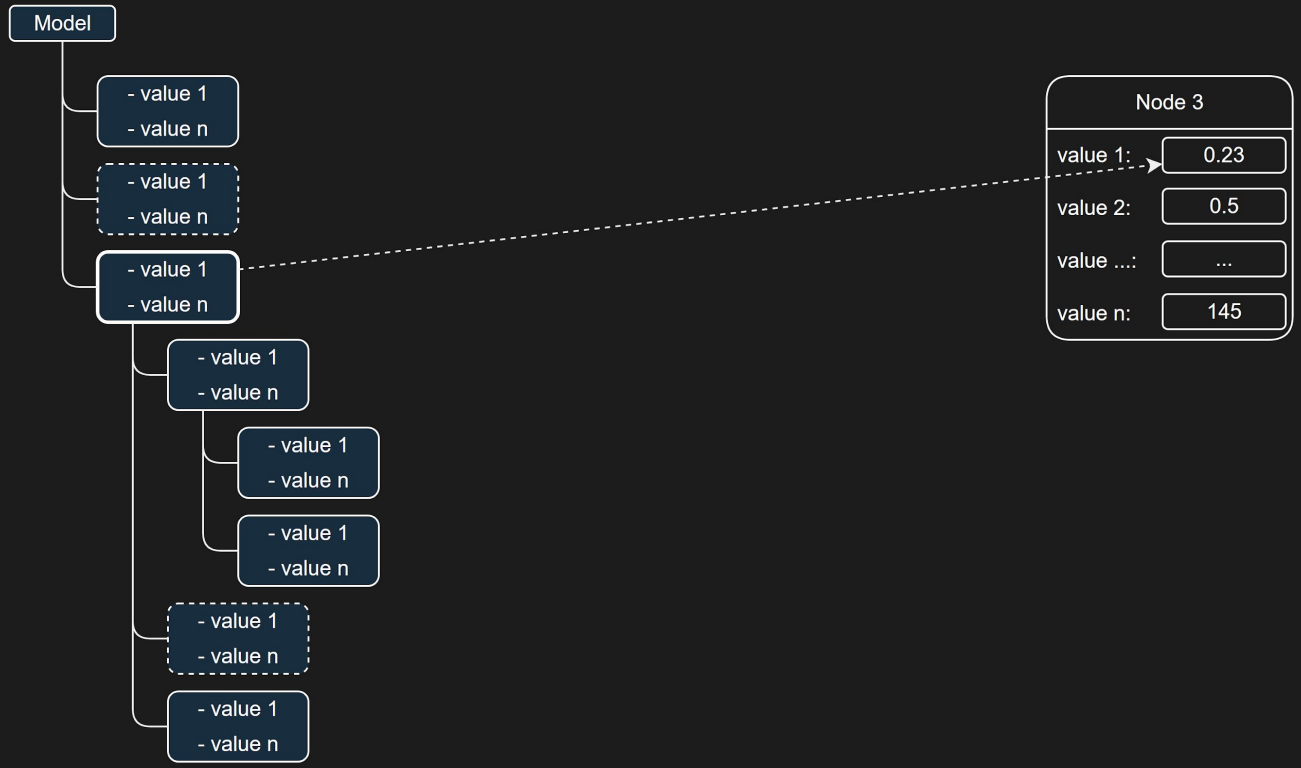
Many of these are not specific to DAWs

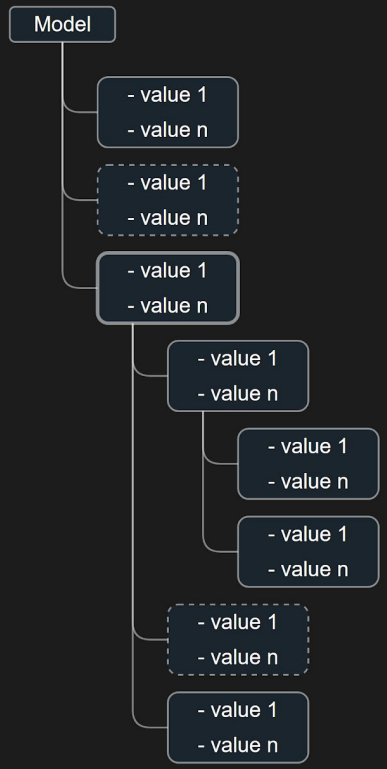
Together, they shape a DAW*

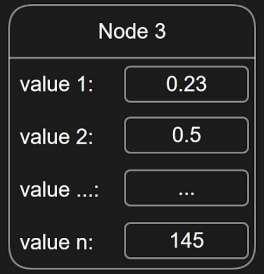
*and other time-based media software

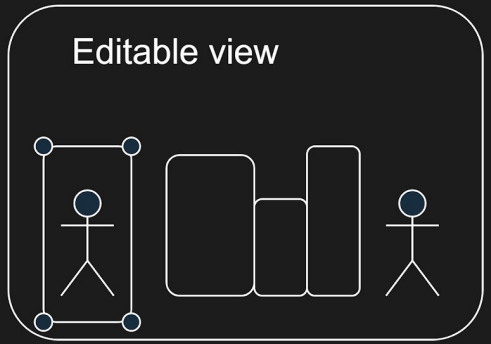


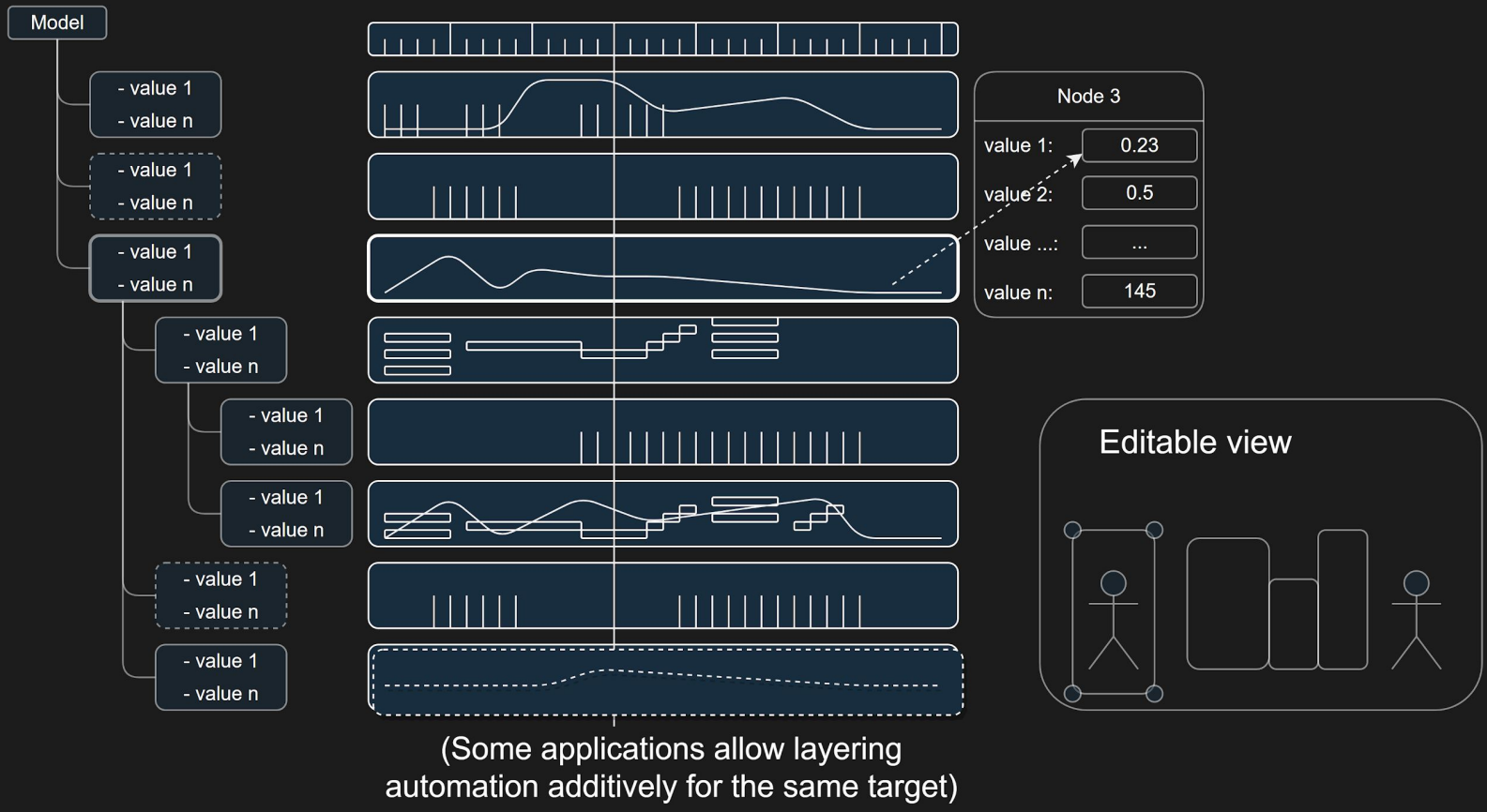

Model
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n


Model
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
Node 3
value 1:
0.23
value 2:
0.5
value ...:
...
value n:
145

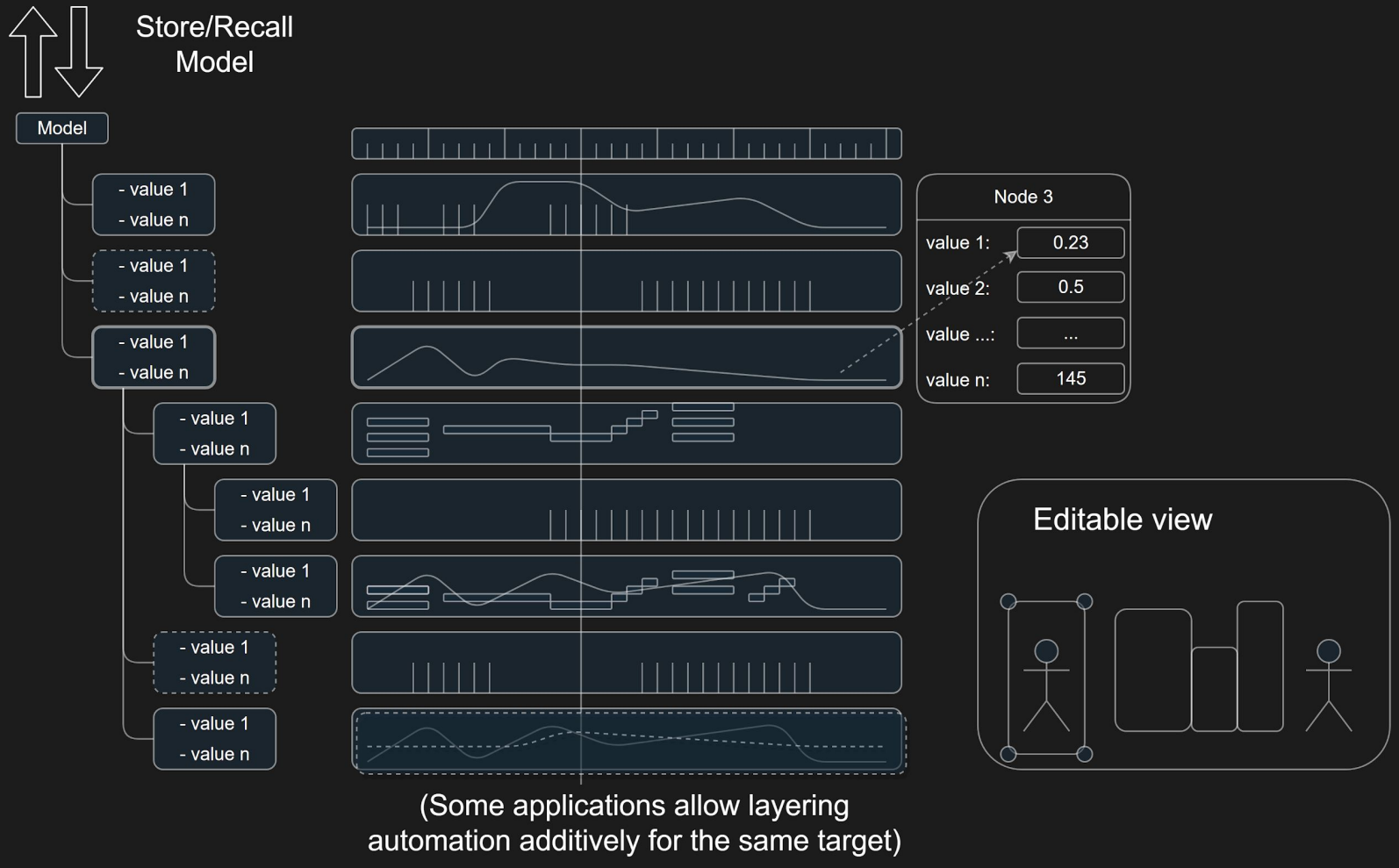

Model
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n

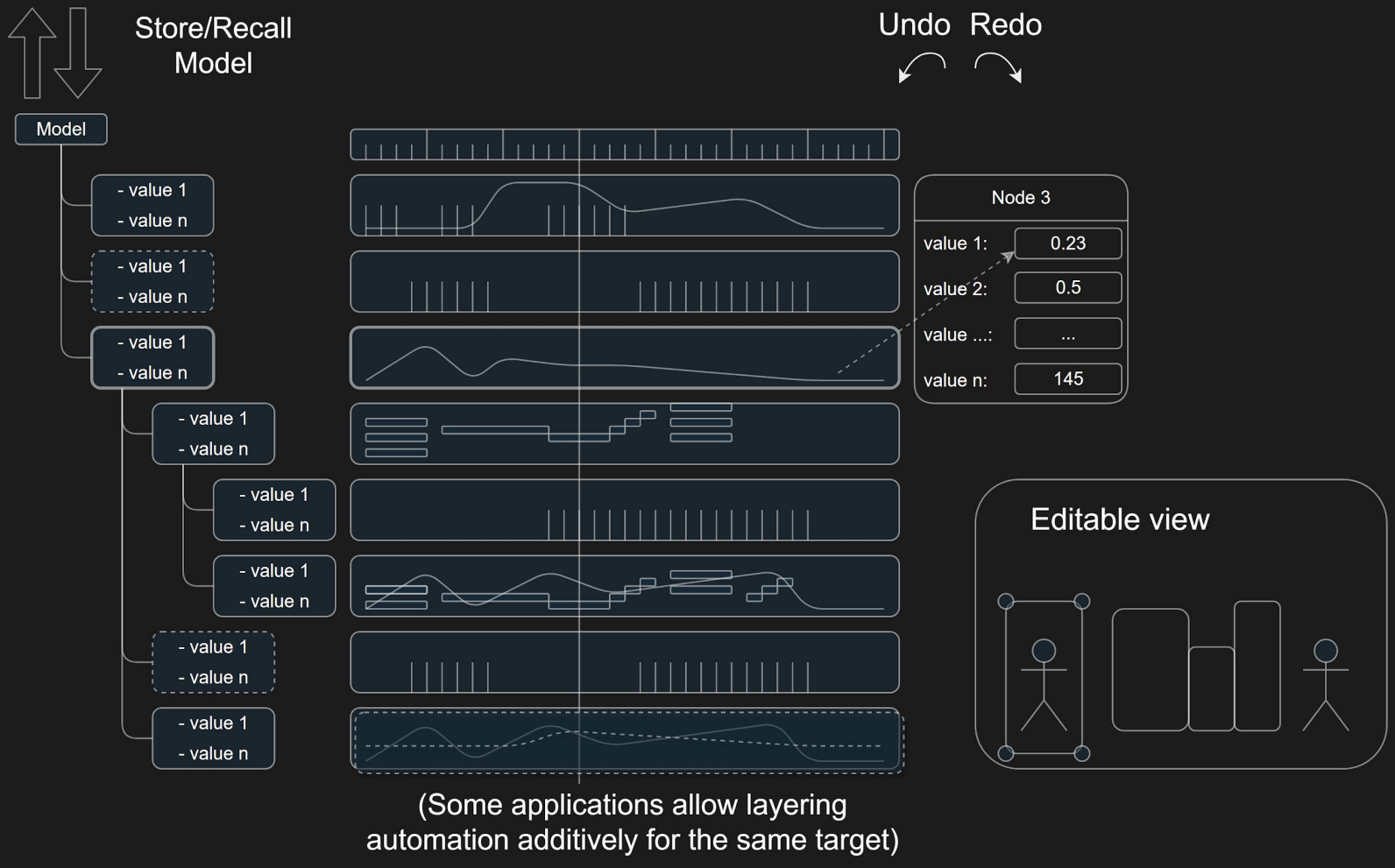

Node 3
value 1:
0.23
value 2:
0.5
value ...:
...
value n:
145

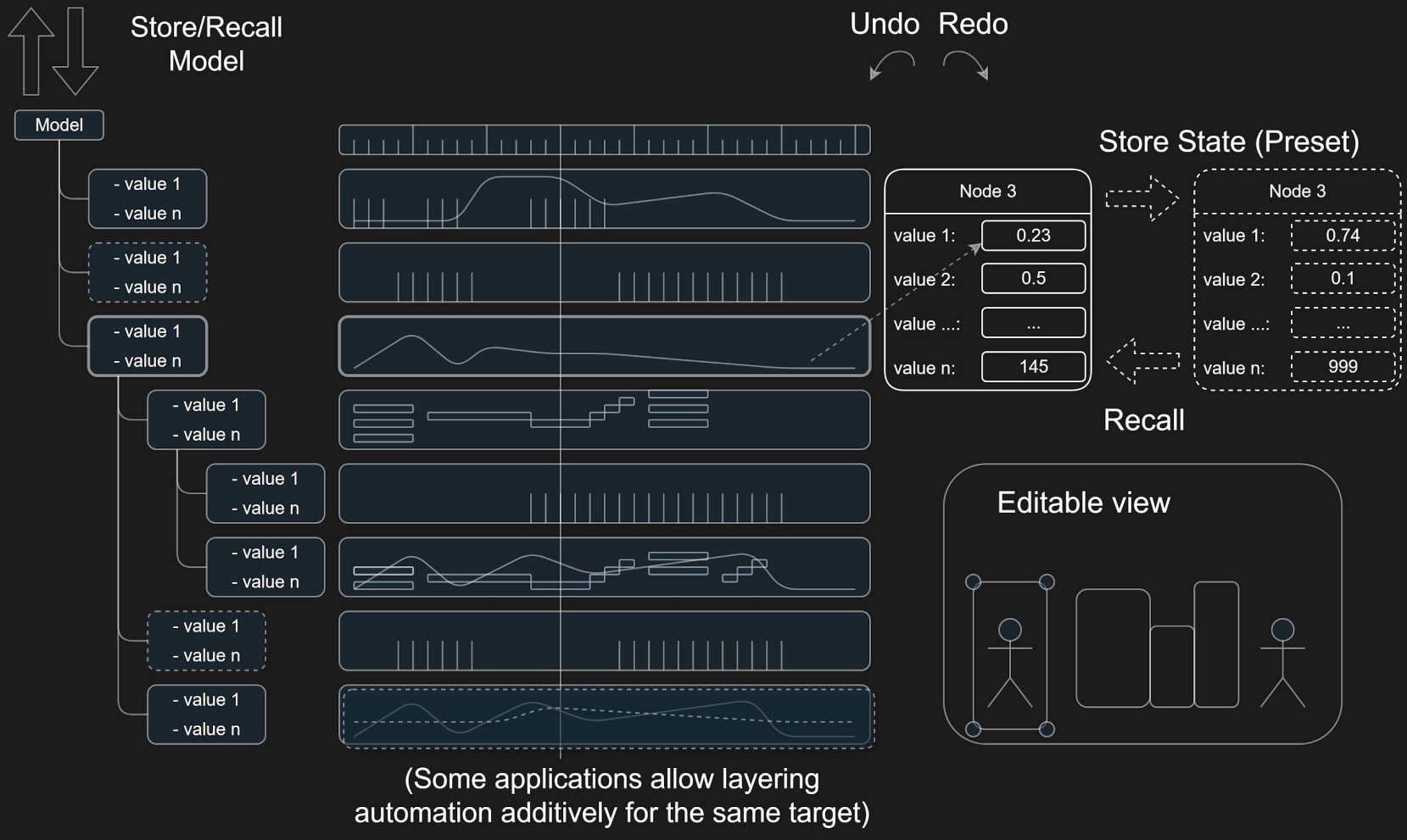

Editable view

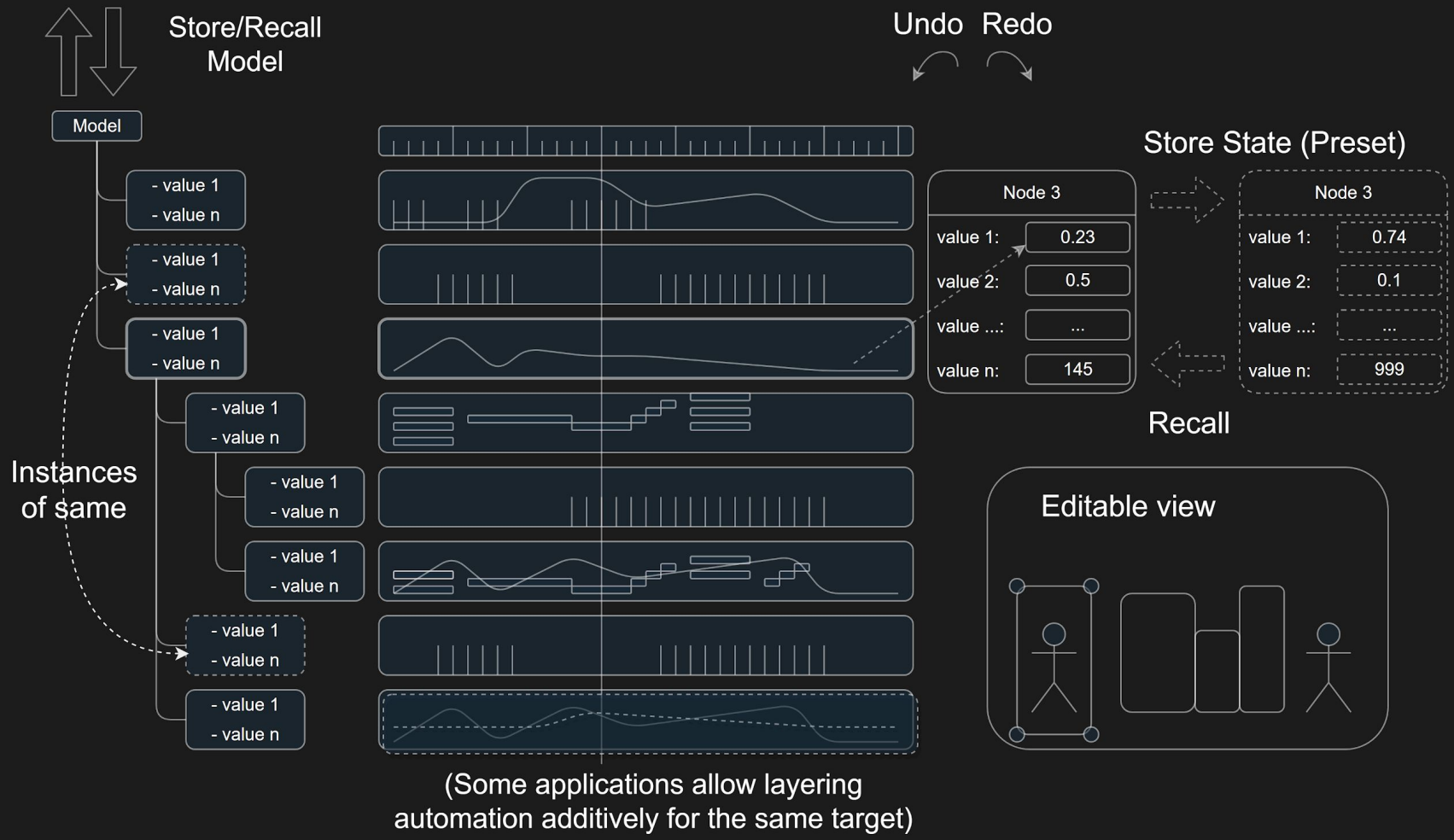


(Some applications allow layering automation additively for the same target)


Store/Recall Model
Model
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
Node 3
value 1: 0.23
value 2: 0.5
value ...: ...
value n: 145
Editable view
(Some applications allow layering automation additively for the same target)


Store/Recall
Model
Undo
Redo
Model
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
Node 3
value 1:
0.23
value 2:
0.5
value ...:
...
value n:
145
Editable view
(Some applications allow layering
automation additively for the same target)


Store/Recall Model
Undo Redo
Model
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
Store State (Preset)
Node 3
value 1: 0.23
value 2: 0.5
value ...: ...
value n: 145
Node 3
value 1: 0.74
value 2: 0.1
value ...: ...
value n: 999
Recall
Editable view
(Some applications allow layering automation additively for the same target)


Store/Recall Model
Undo Redo
Model
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
- value 1
- value n
Instances of same
Node 3
value 1: 0.23
value 2: 0.5
value ...: ...
value n: 145
Store State (Preset)
Node 3
value 1: 0.74
value 2: 0.1
value ...: ...
value n: 999
Recall
Editable view
(Some applications allow layering automation additively for the same target)

Also:

Connect physical controllers to parameters

End-user programming language

Customizing interface

Plug-in API’s

Patterns are a vocabulary

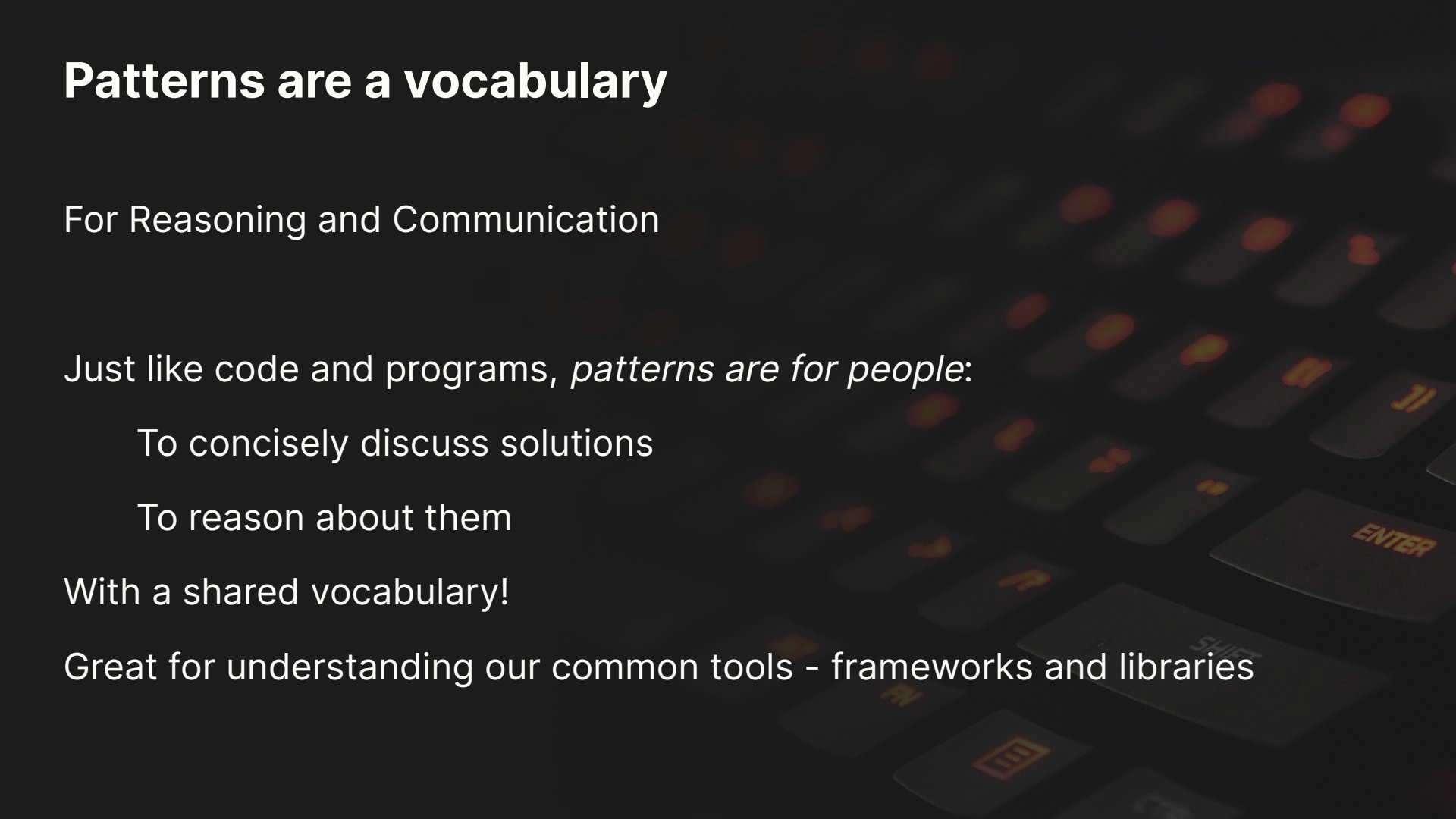
For Reasoning and Communication

Just like code and programs, *patterns are for people*:

- To concisely discuss solutions
- To reason about them

With a shared vocabulary!

Great for understanding our common tools - frameworks and libraries



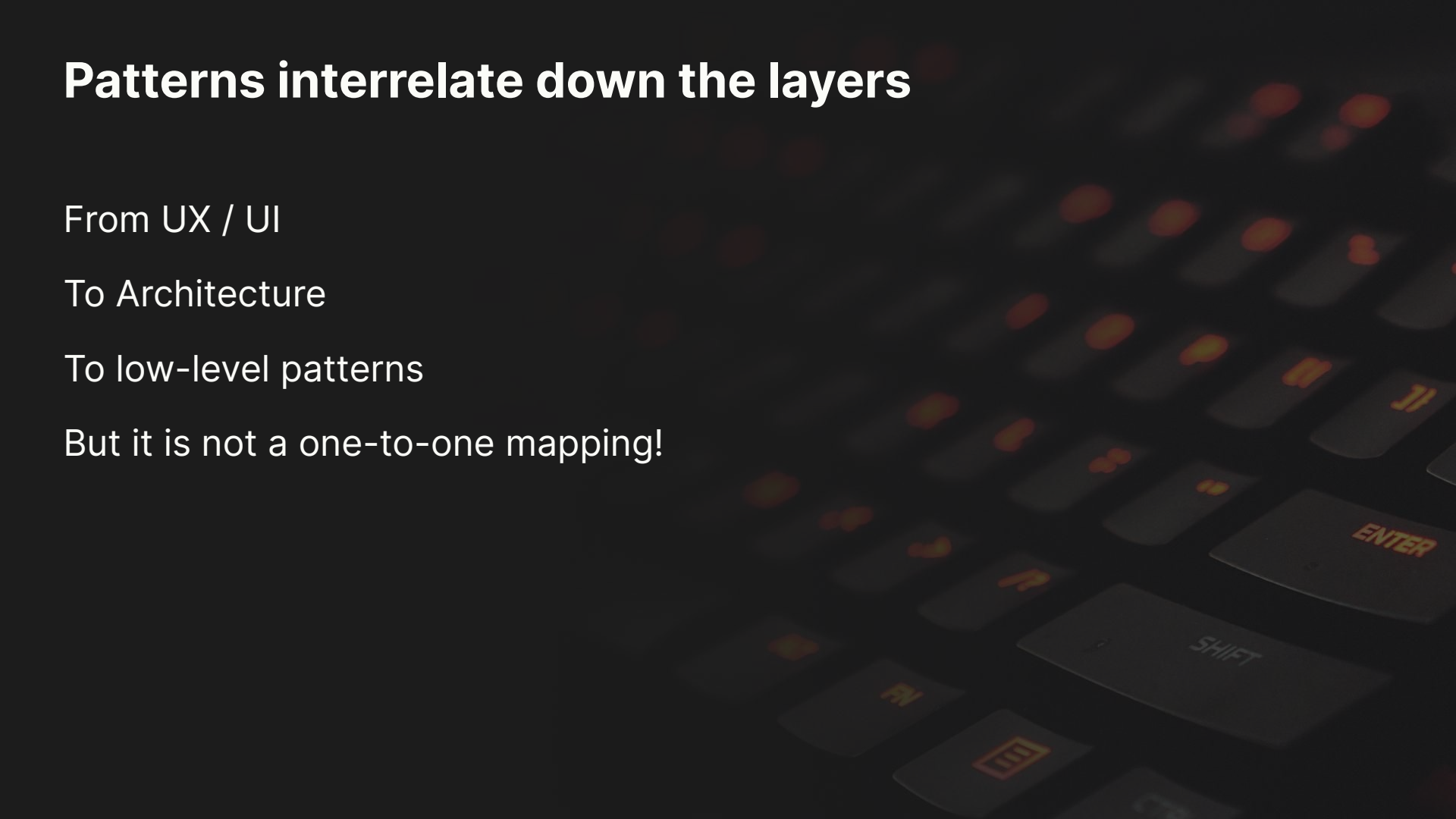
Patterns interrelate down the layers

From UX / UI

To Architecture

To low-level patterns

But it is not a one-to-one mapping!



Assembling Sushi and TWO

Common Between Sushi and TWO

Classes that acquire resources → abstract interfaces.

Created and assembled → through factories.

Passed into other classes → using dependency injection.

Makes for:

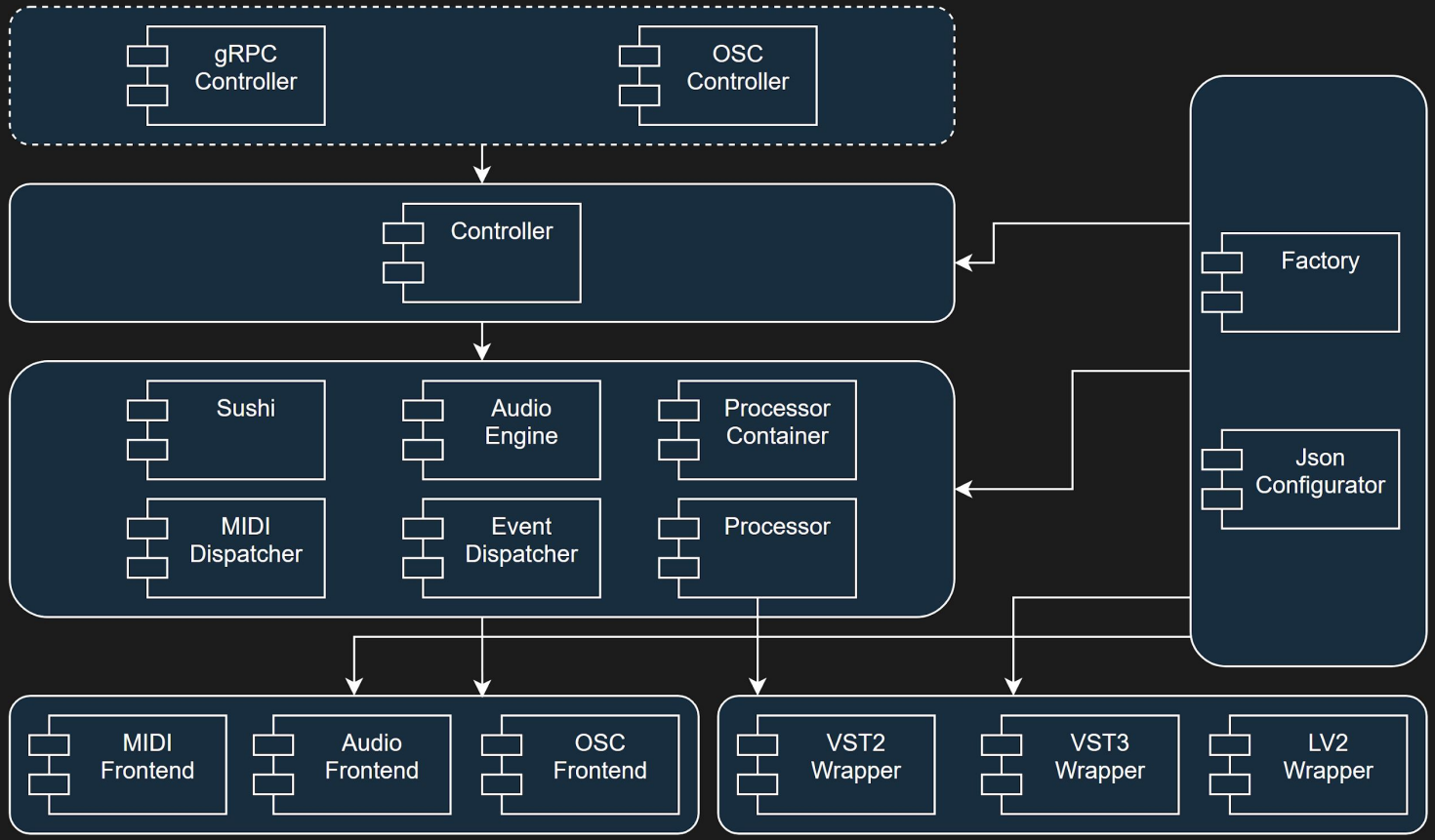
Highly modular, and easily testable code.

Google test/mock are used for both.

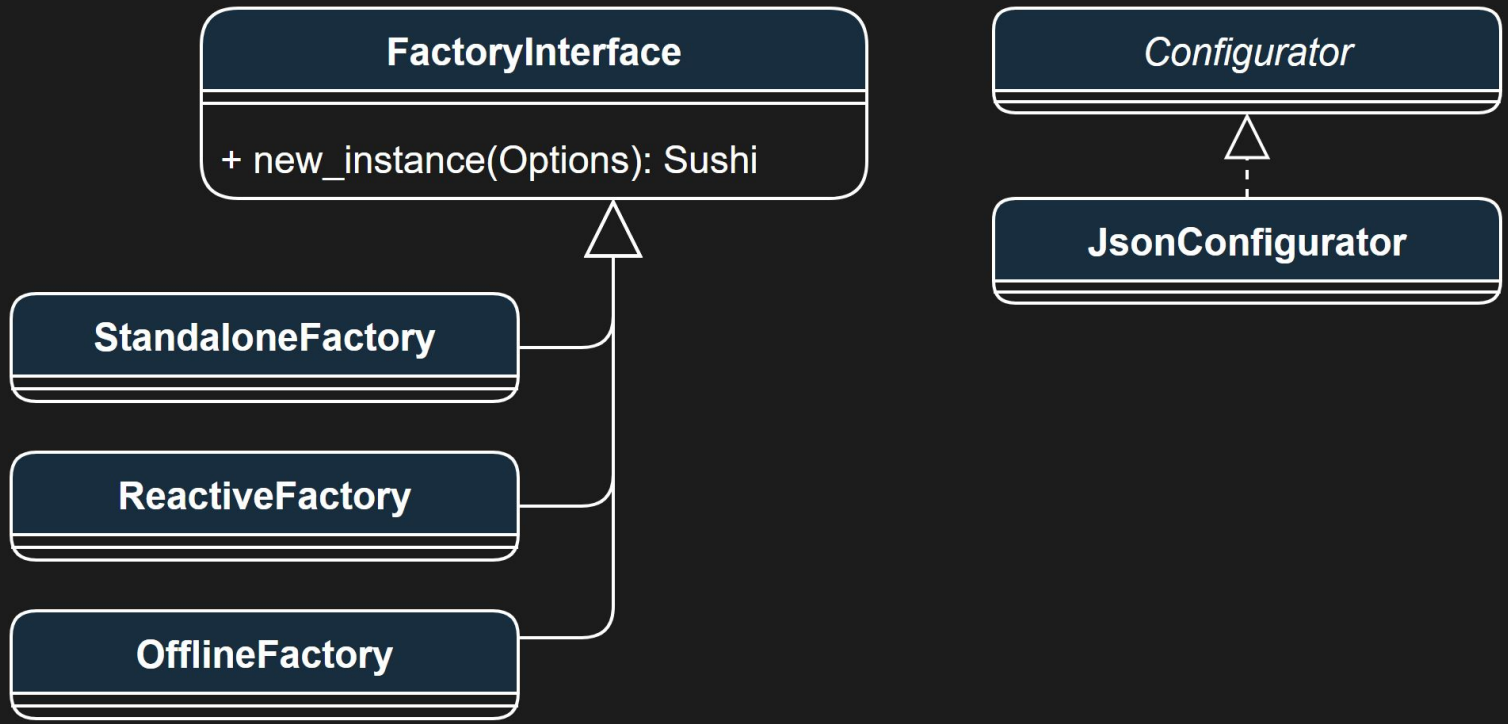
Both have logging functionality.

Sushi's Architecture

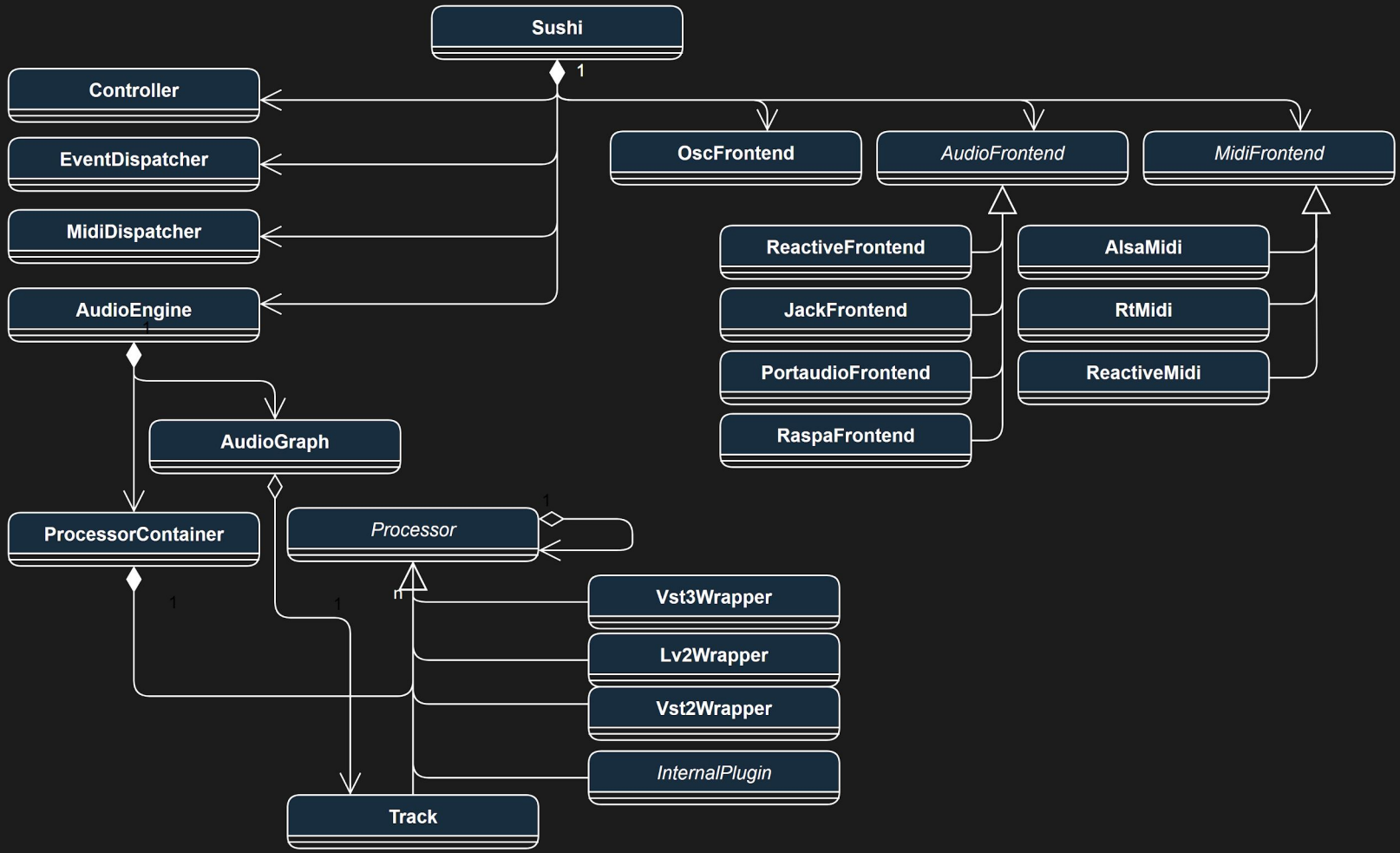
Main Modules and Layers

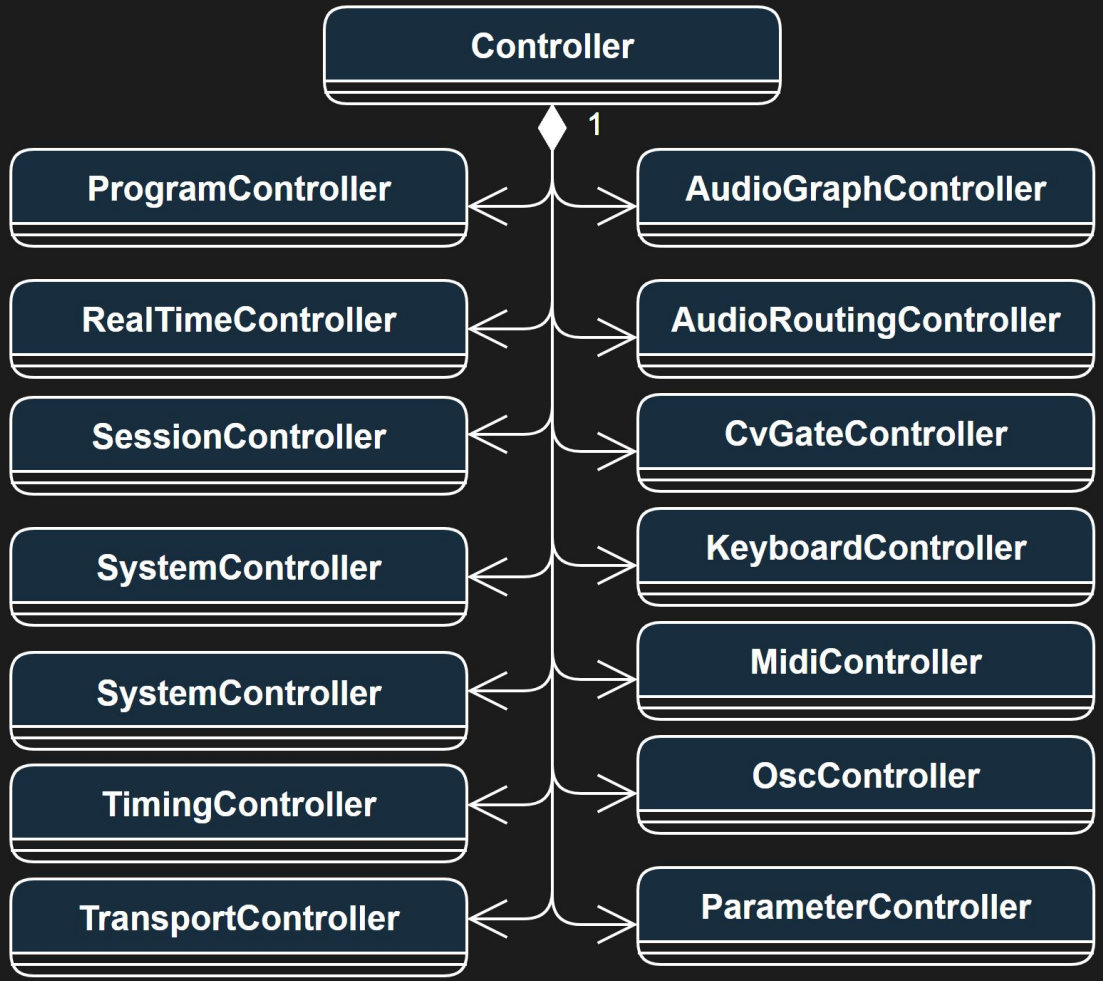


Instantiation and Configuration



Sushi


Sushi
1
Controller
EventDispatcher
MidiDispatcher
AudioEngine
1
OscFrontend
AudioFrontend
MidiFrontend
ReactiveFrontend
JackFrontend
PortaudioFrontend
RaspaFrontend
AlsaMidi
RtMidi
ReactiveMidi
AudioGraph
ProcessorContainer
Processor
1
1
1
n
Vst3Wrapper
Lv2Wrapper
Vst2Wrapper
InternalPlugin
Track


Controller
1
ProgramController
AudioGraphController
RealTimeController
AudioRoutingController
SessionController
CvGateController
SystemController
KeyboardController
SystemController
MidiController
TimingController
OscController
TransportController
ParameterController

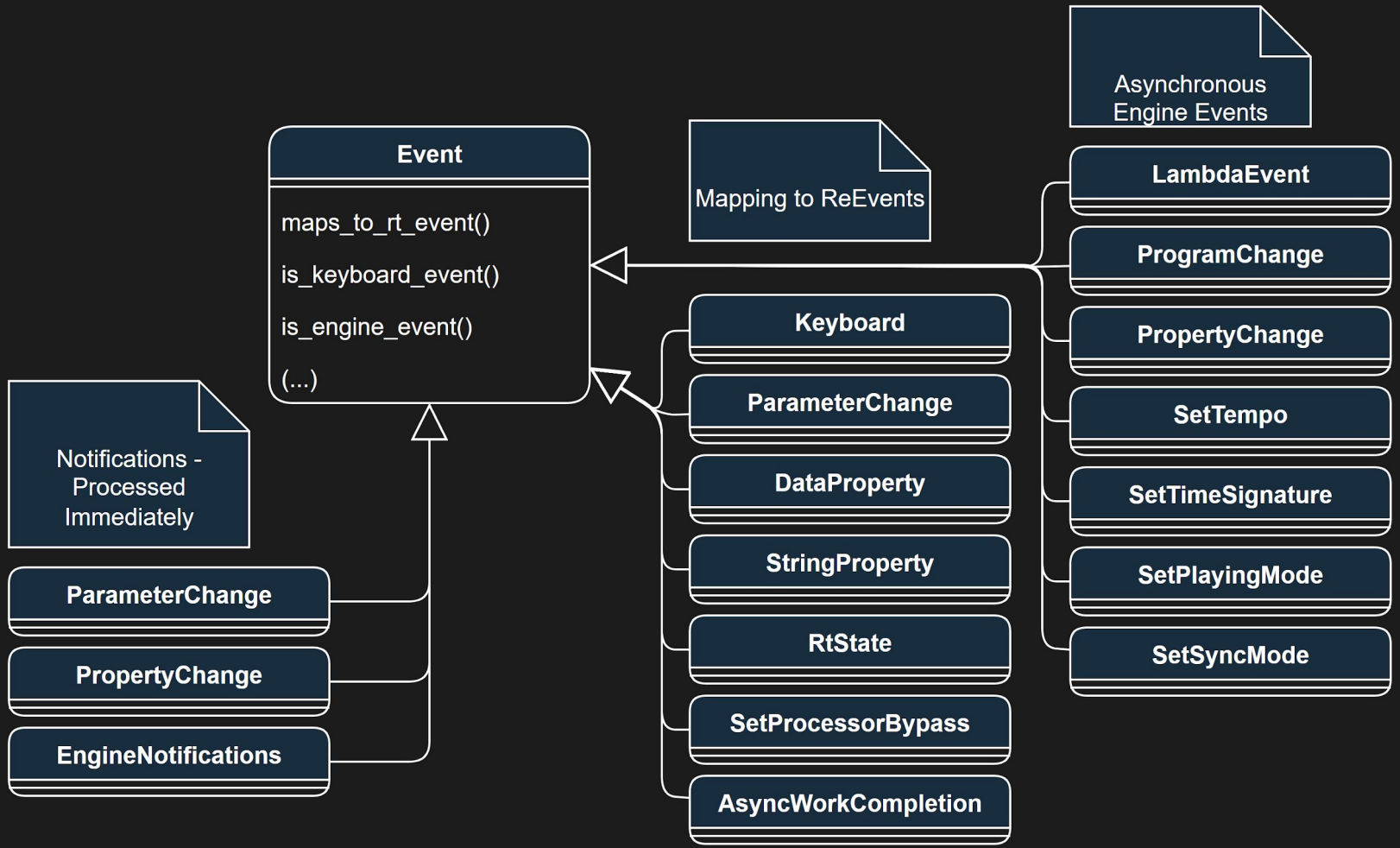
Messaging

Main event loop handles:

- Note events, parameter changes, etc.
  - Do not block for long.

- Worker for, heavy tasks

- Communication between non-rt threads and audio thread:
  - RtEvents, via lock-free queues.


Asynchronous Engine Events
Event
maps_to_rt_event()
is_keyboard_event()
is_engine_event()
(...)
Mapping to ReEvents
LambdaEvent
ProgramChange
PropertyChange
SetTempo
SetTimeSignature
SetPlayingMode
SetSyncMode
Keyboard
ParameterChange
DataProperty
StringProperty
RtState
SetProcessorBypass
AsyncWorkCompletion
Notifications - Processed Immediately
ParameterChange
PropertyChange
EngineNotifications

RtEvent

- Tagged union of events and commands - real-time safe
- Small - 32 bytes

Works like a base class, but still allows trivial copying.

Goes against *"thou shall not inherit without virtual destructor"*

It can be copied byte-by-byte without constructors or destructors

Audio Engine

Contains:

- Audio graph
- Lock-free queues - to and from the audio thread(s)

We have a rich AudioGraph - but I won’t detail it more.

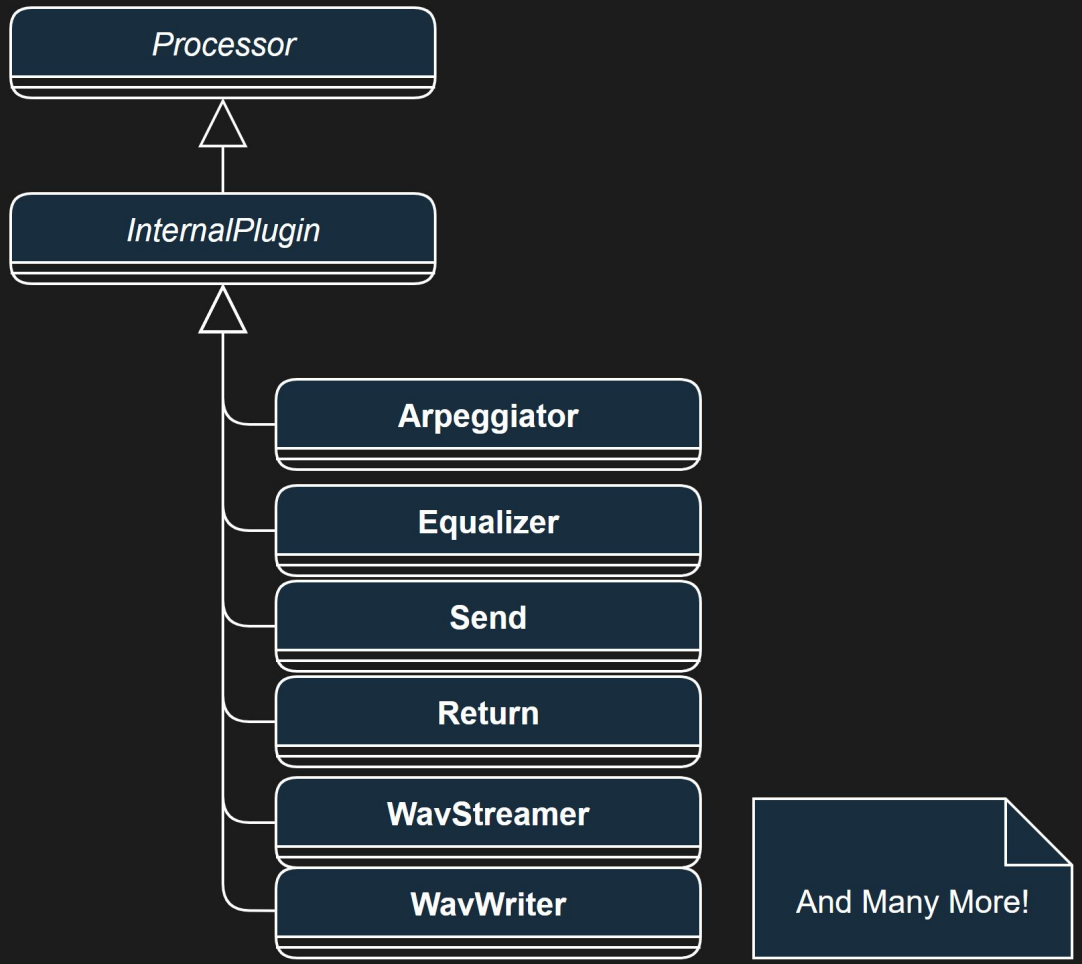
Beautifully covered by Dave Rowland at ADC ‘20:

Introducing Tracktion Graph: A Topological Processing Library for Audio

Internal Plugins

Sushi implements all audio processing functionality in internal plugins:

- Internal functionality
  - Send-Return
  - CV I/O
  - Audio file streaming
- Standard audio tools
- Brickworks library plugins



TWO's Architecture

Open Sound Control's structure is central

Open Sound Control

OSC 25 years old now!

Originally for music performance data

In widespread use today

OSC doesn’t *compete* with MIDI 1.0/2.0 - it *complements* it!

For end-user innovation

OSC is widely supported

Control applications:

TouchOSC - TWO - OpenStageControl - OSC/Pilot - Lemur

DAW's:

Ableton Live - BitWig - Logic - Reaper - MOTU Digital Performer - LV2 Ingen

Plugins:

Native Instruments' Reaktor - FAW Circle^2 ...

Visuals, Stage Lighting & Projection Mapping:

Resolume - TouchDesigner - Unreal Engine - Notch - VDMX - Synesthesia - MadMapper ...

OSC - Messages for a synth instance

/synth/oscillator_1/frequency, f, 440.0

/synth/filter/cutoff, f, 65.0

/synth/envelope, ffff, 0.5, 1.0, 1.0, 0.1

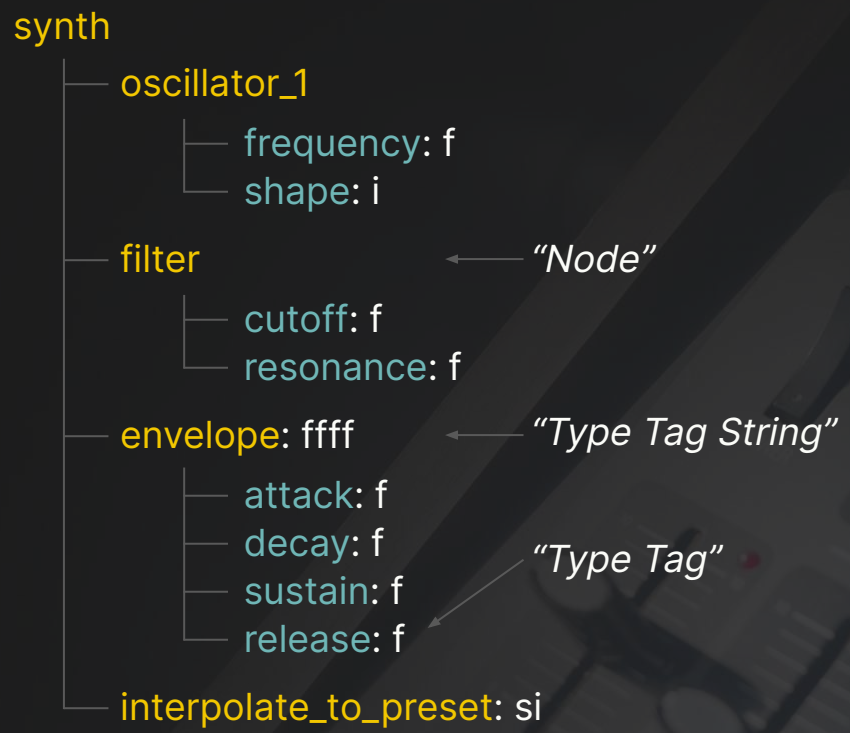
/synth/interpolate_to_preset, si, “lately_bass”, 1000

“Address Pattern”

“Type Tag String”

“Arguments”

OSC - Simple synthesizer namespace



TWO's User-facing features

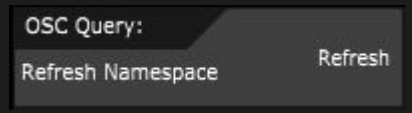
Namespace management

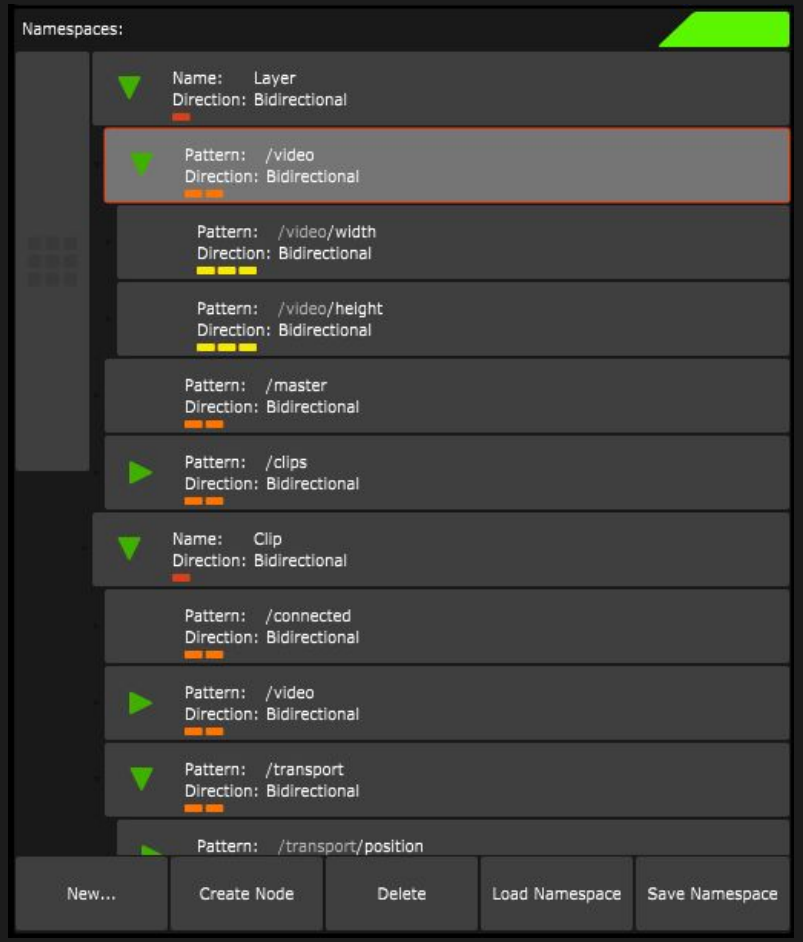
Central to TWO

Typically OSC support means a lot of tedious typing.

OSC Query automates that.

Otherwise TWO automatically builds namespaces.





Sequencing

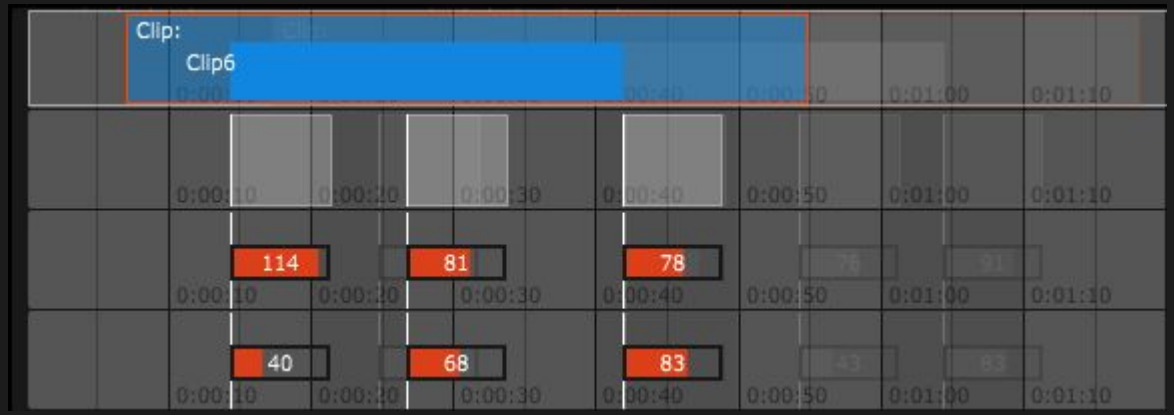
Record, edit and replay any messages.

Multi-layered sequencing

Additional modifying & interpolating layer of sequenced value changes.

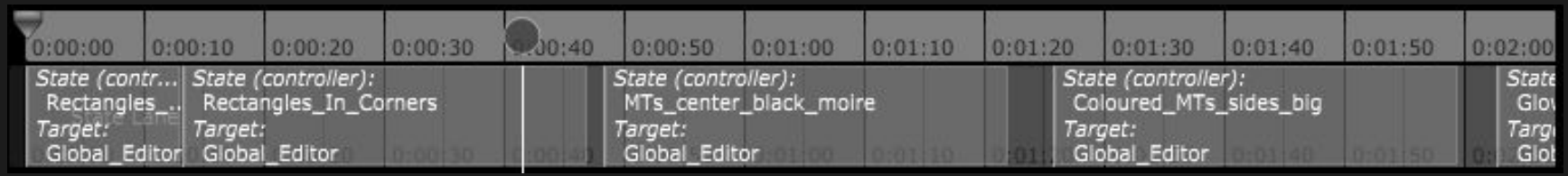
Instead of modifying the originally recorded data.

(Idea from Motion Capture software)



Interpolate between saved states

Immediate control of animation across large sets of parameters.



Connections and Mapping

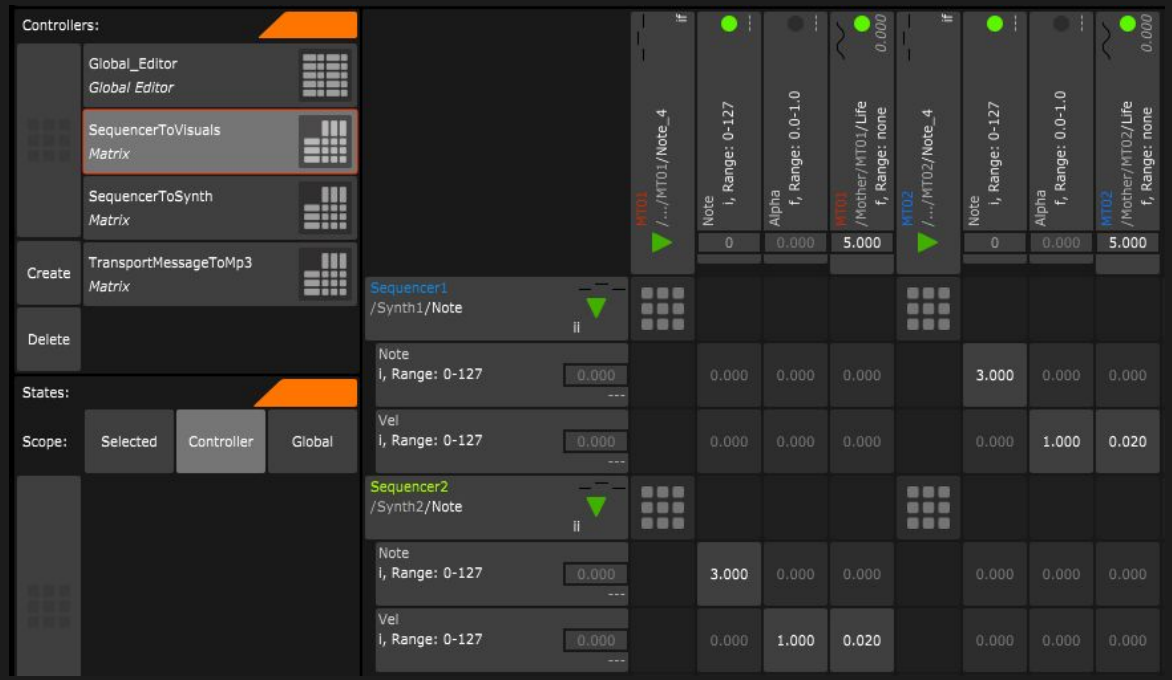
Mapping is central:

To control 3rd party media tools.

With Matrix:

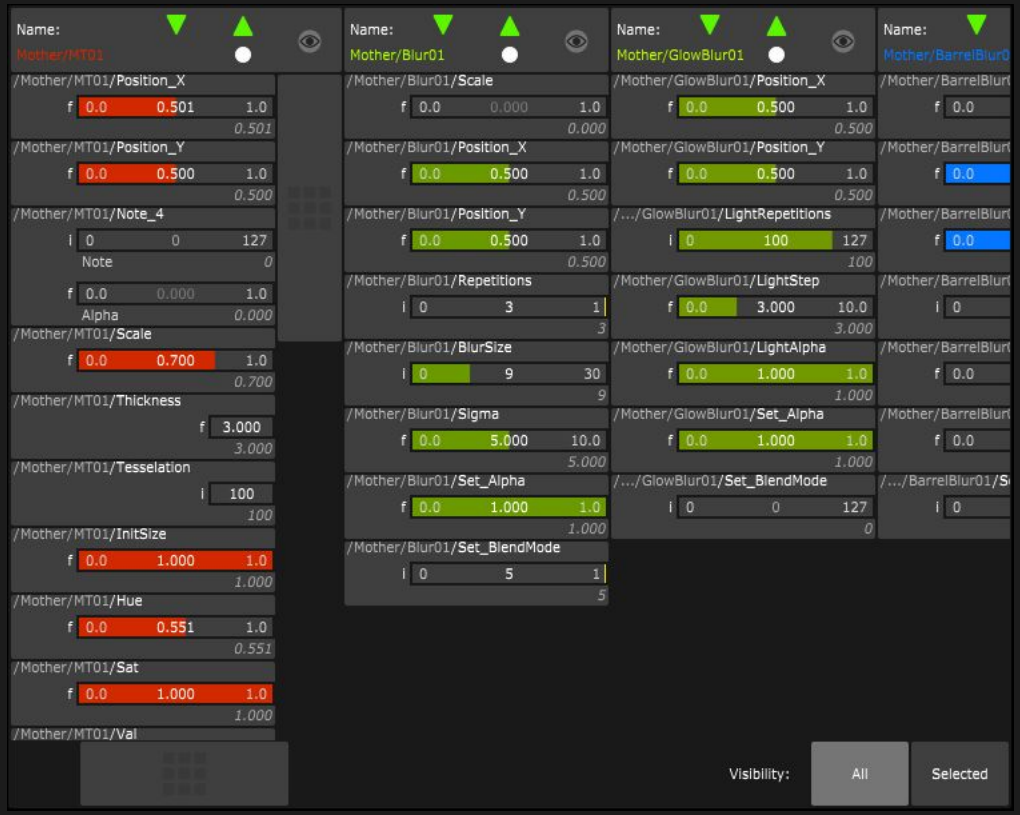
Gradually create, alter and destroy connections.

And even animate that!



UI for remote-controlling multiple targets

In the Editor, each column is a separate target.



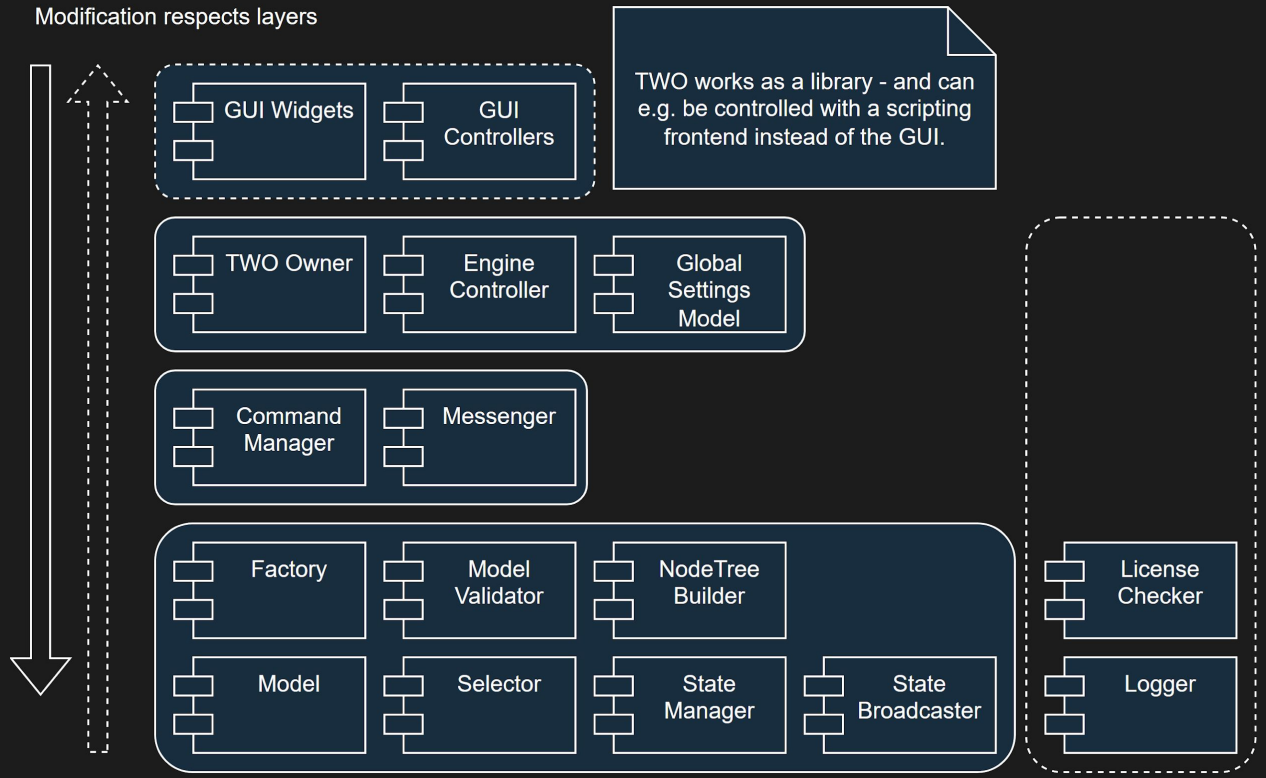
UI for remote-controlling multiple targets

You can also choose to make custom UI.

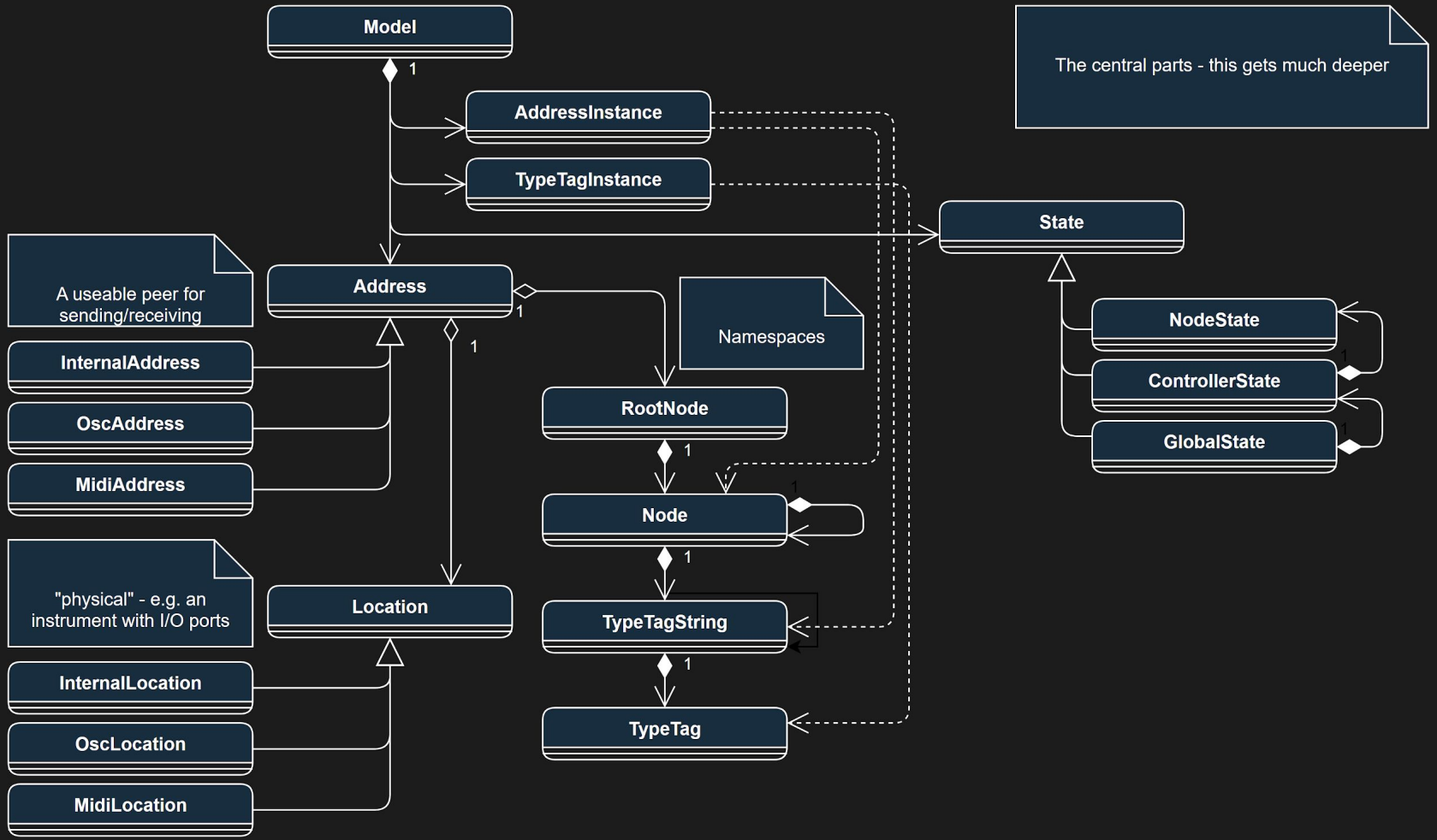
Like e.g. TouchOSC, Lemur

TWO's Internals

TWO Overall Architecture



But Notification can skip them


Model
1
AddressInstance
TypeTagInstance
The central parts - this gets much deeper
State
A useable peer for sending/receiving
Address
1
1
Namespaces
InternalAddress
OscAddress
MidiAddress
RootNode
1
NodeState
ControllerState
GlobalState
1
1
1
Node
1
1
"physical" - e.g. an instrument with I/O ports
Location
TypeTagString
1
InternalLocation
OscLocation
MidiLocation
TypeTag

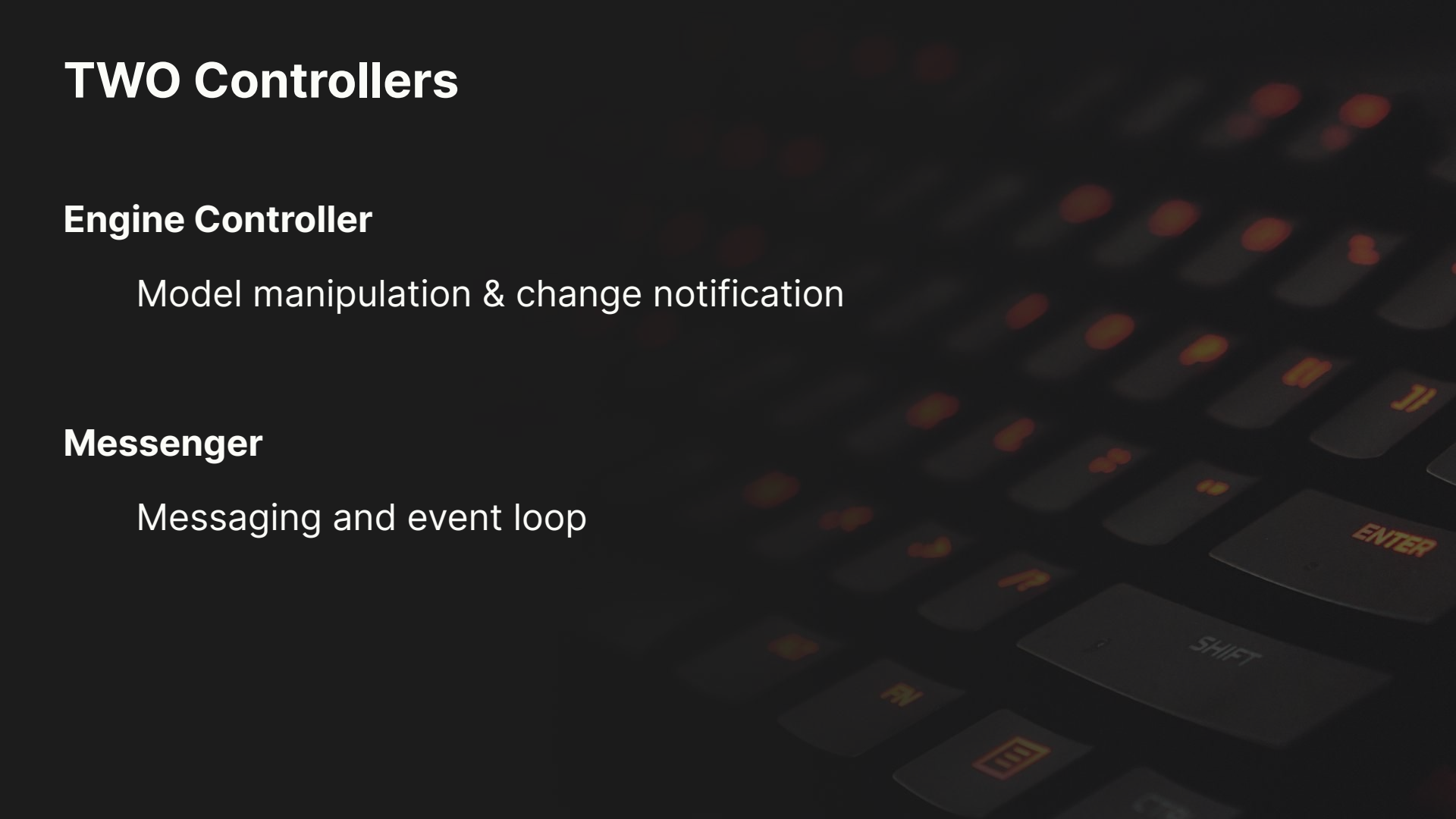
TWO Controllers

Engine Controller

Model manipulation & change notification

Messenger

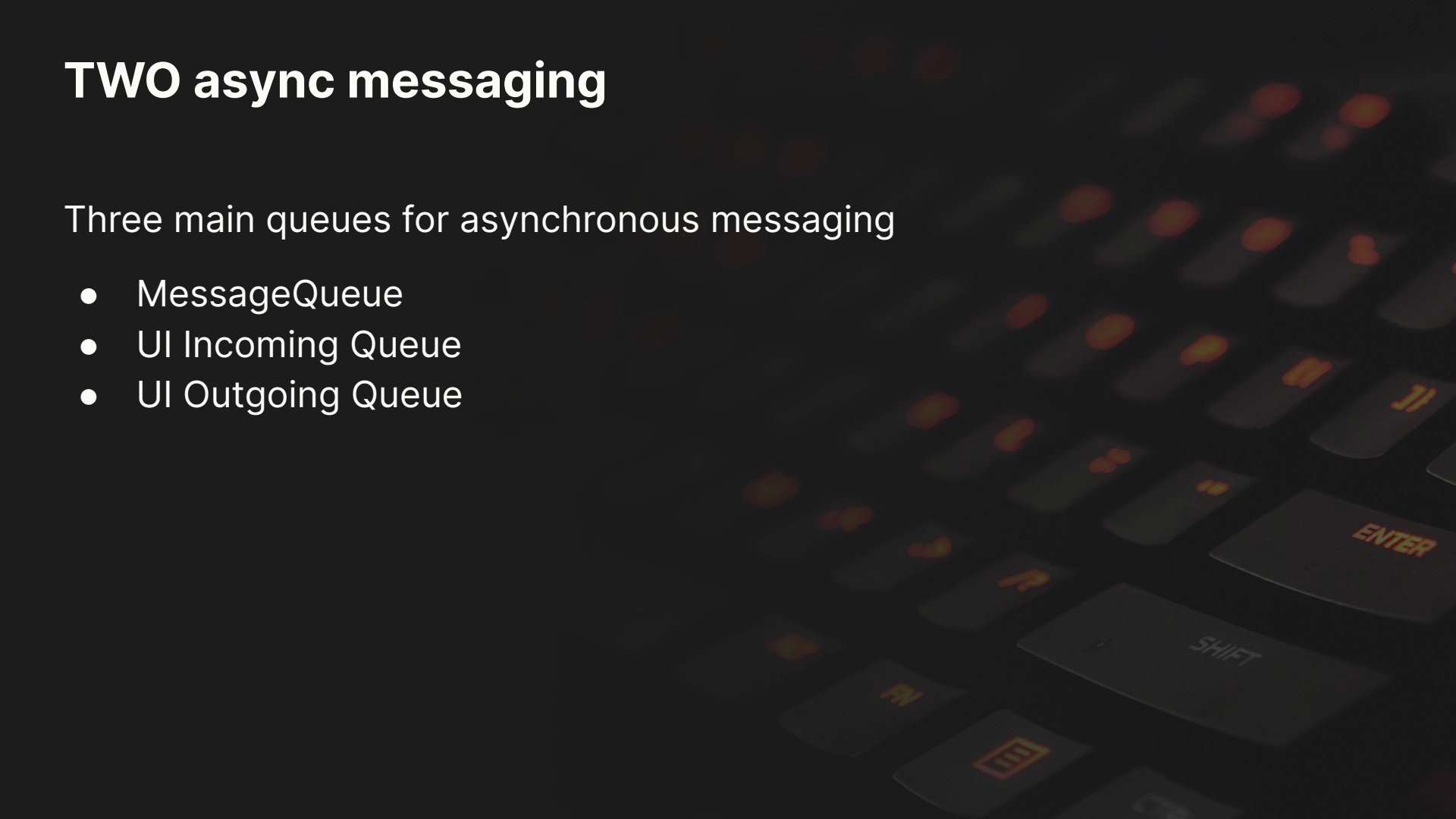
Messaging and event loop



TWO async messaging

Three main queues for asynchronous messaging

- MessageQueue
- UI Incoming Queue
- UI Outgoing Queue



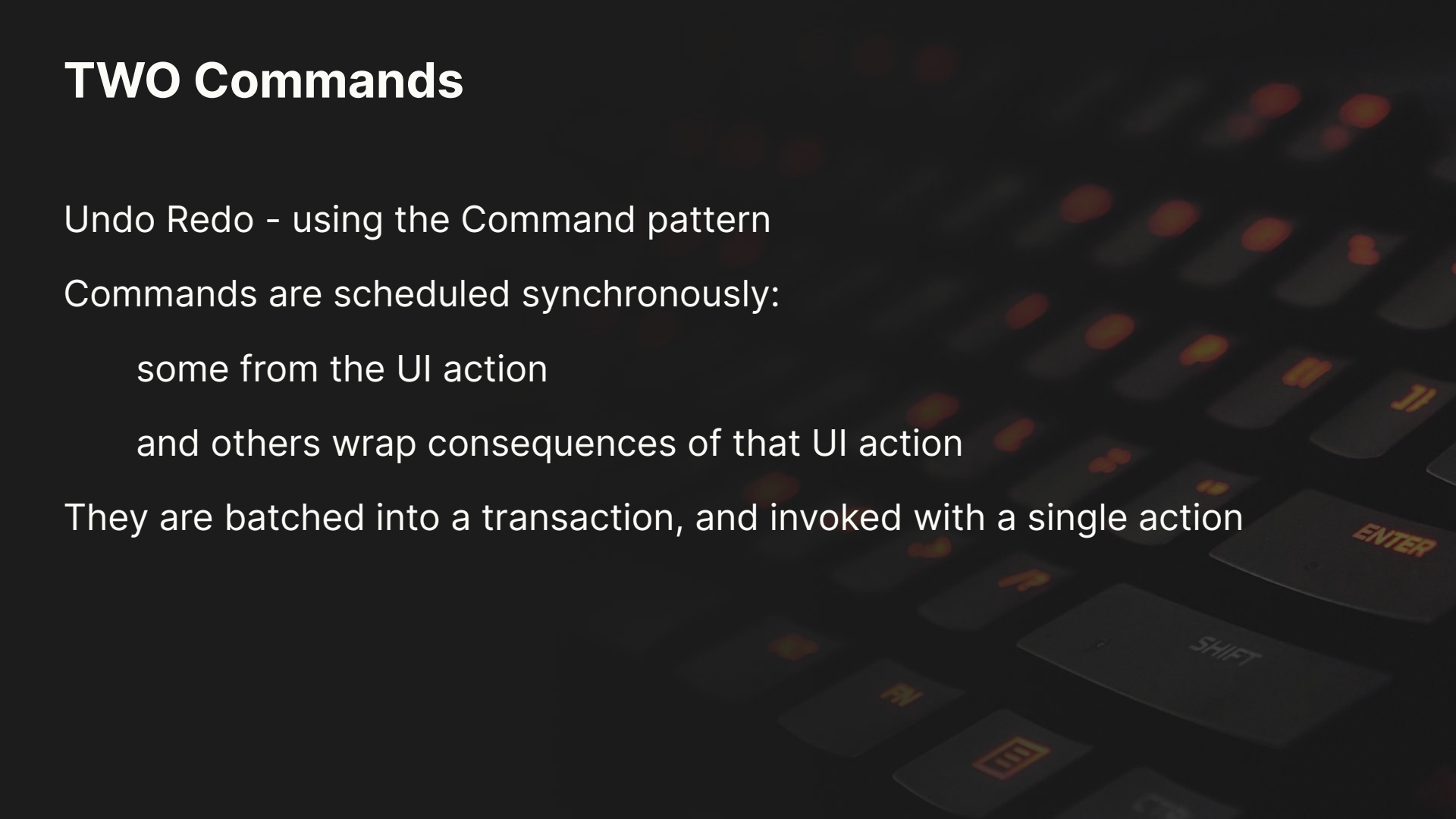
TWO Commands

Undo Redo - using the Command pattern

Commands are scheduled synchronously:

- some from the UI action
- and others wrap consequences of that UI action

They are batched into a transaction, and invoked with a single action



TWO UI

For the UI, TWO uses Juce more heavily than for the engine.

So there's less to say about it, assuming you're familiar with Juce.

The majority of the components are based on the Juce TreeView.

Suffice to add:

- TWO has many additional, view-specific, controller classes.
- The UI is the majority of the code in quantity - but not complexity.

Discussion

DAWs are “complex codebases”

Ideally, these avoid repetition, and re-use common patterns.

DAW’s ***will*** be complex

They don’t ***need*** to be large

Badly thought out → grow in size quickly, harder to work on.

With simpler codebases you can ad-hoc the architecture - not so with large.

Avoid the God Class

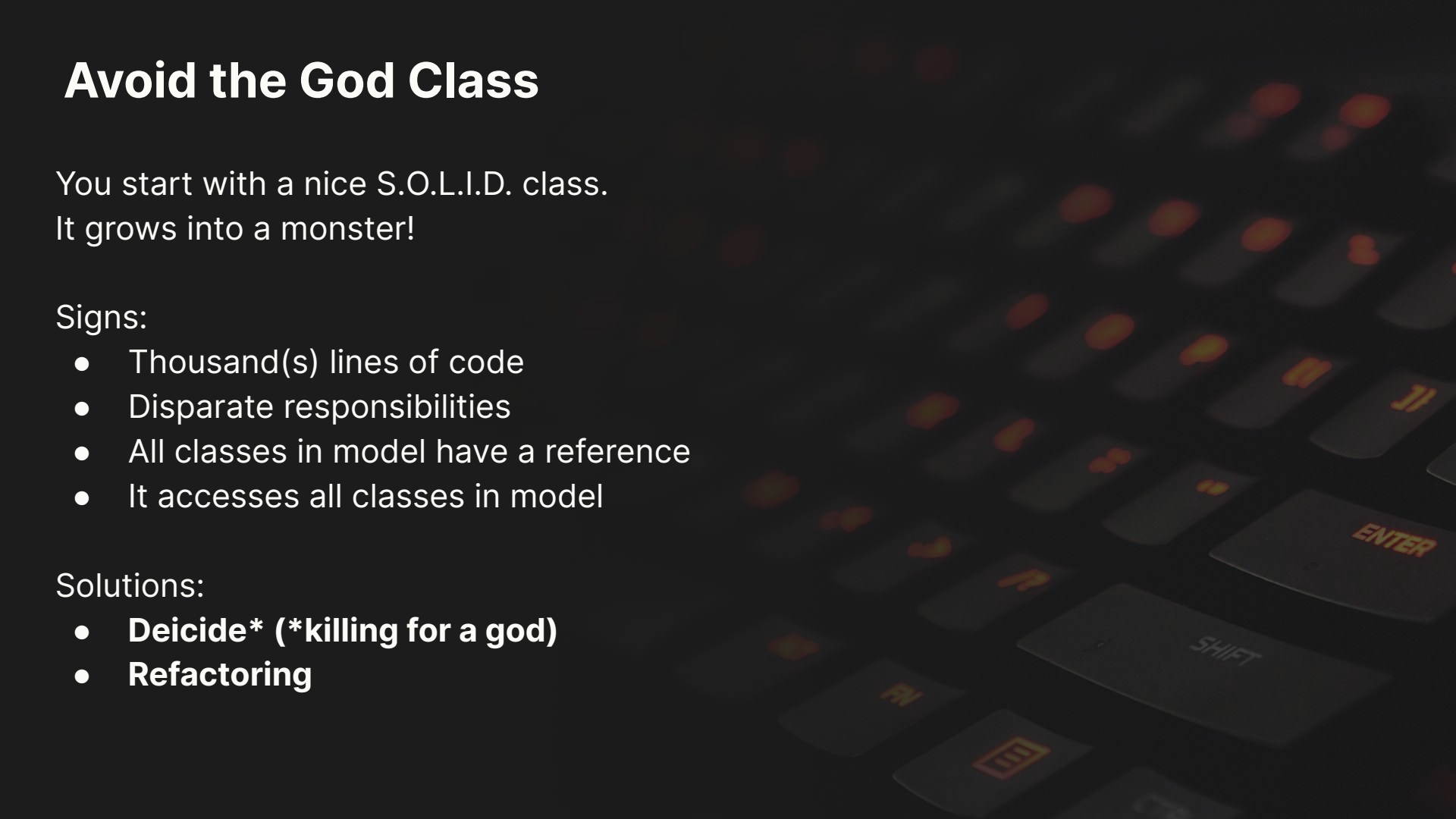
You start with a nice S.O.L.I.D. class.
It grows into a monster!

Signs:

- Thousand(s) lines of code
- Disparate responsibilities
- All classes in model have a reference
- It accesses all classes in model

Solutions:

- **Deicide* (*killing for a god)**
- **Refactoring**



Singleton - considered harmful?

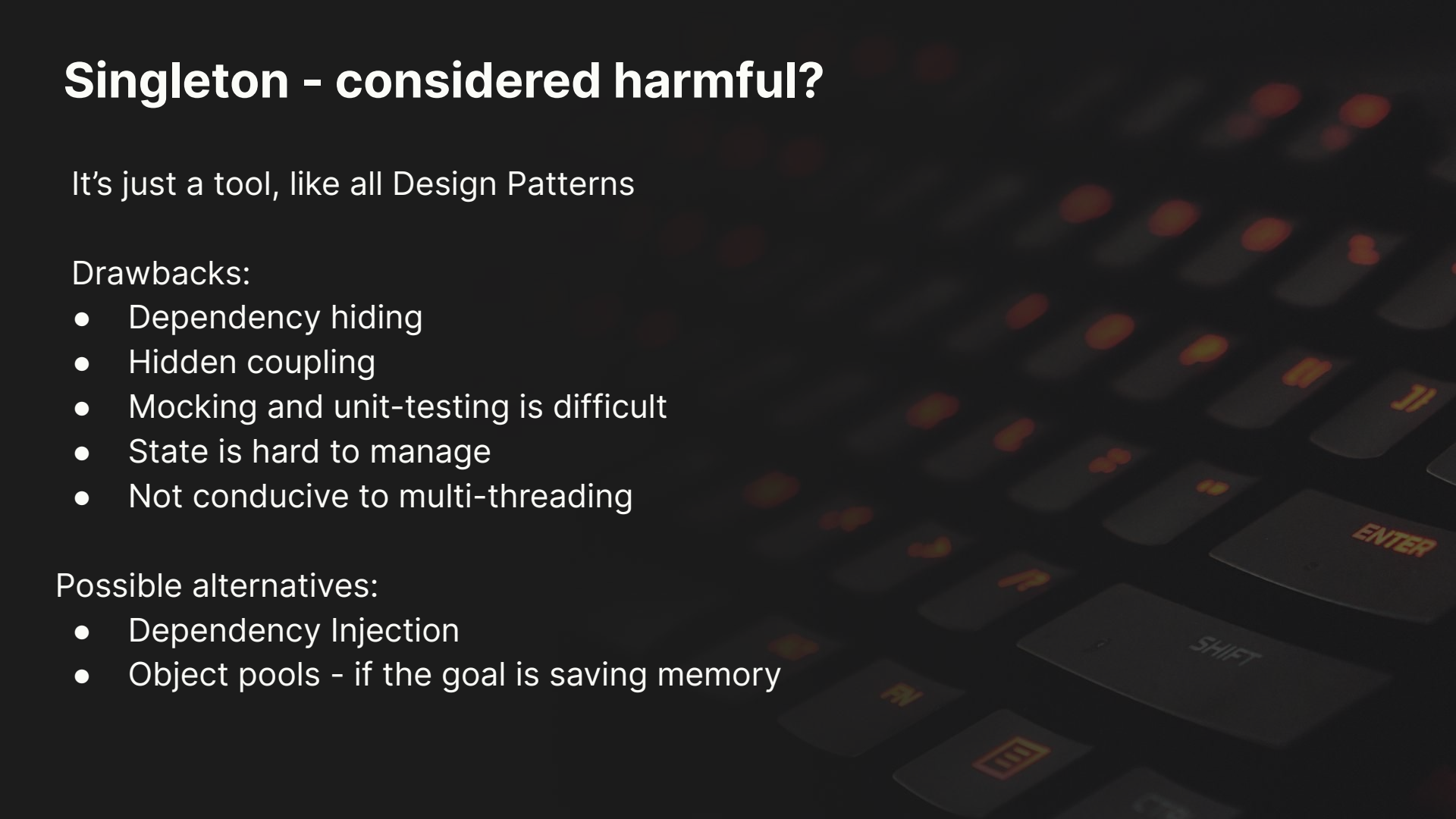
It’s just a tool, like all Design Patterns

Drawbacks:

- Dependency hiding
- Hidden coupling
- Mocking and unit-testing is difficult
- State is hard to manage
- Not conducive to multi-threading

Possible alternatives:

- Dependency Injection
- Object pools - if the goal is saving memory

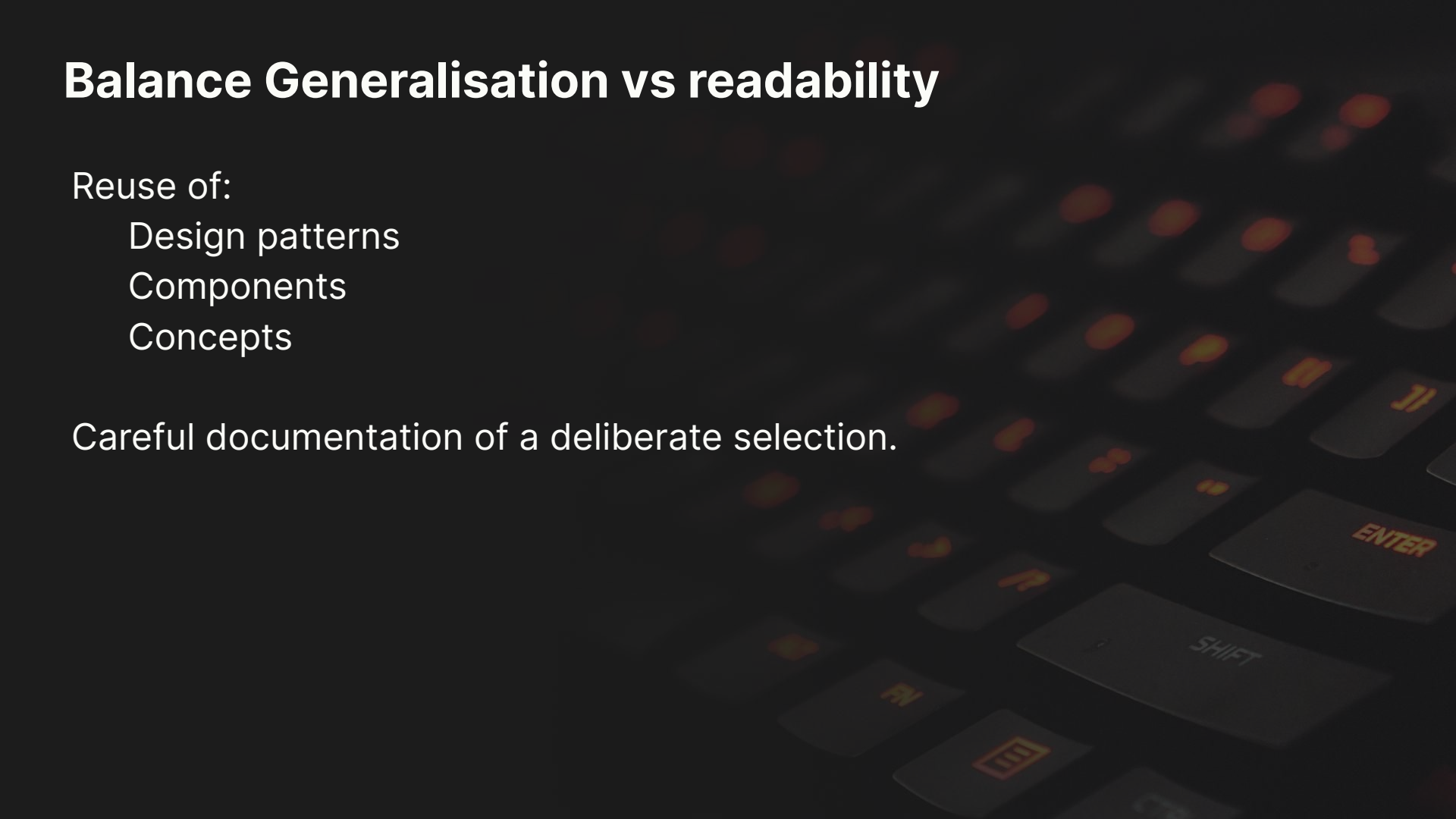


Balance Generalisation vs readability

Reuse of:

- Design patterns
- Components
- Concepts

Careful documentation of a deliberate selection.



What differences in TWO & Sushi explain the different design choices?

TWO has a complex model:

- Many more types of nodes, nested in a much deeper hierarchy.
- TWO also has a UI.

Sushi emphasises performance

TWO emphasises flexibility

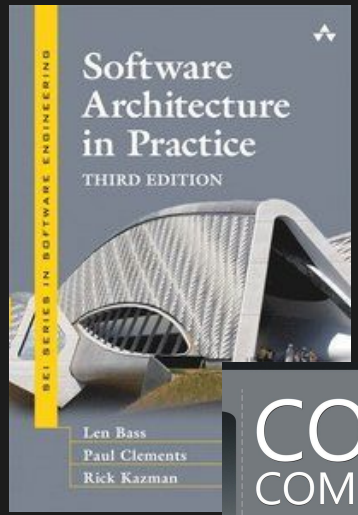
These clash, down to the low-level architecture.

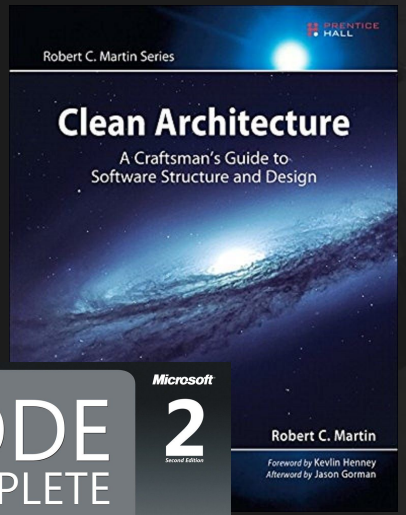
You may be familiar with S.O.L.I.D and YAGNI

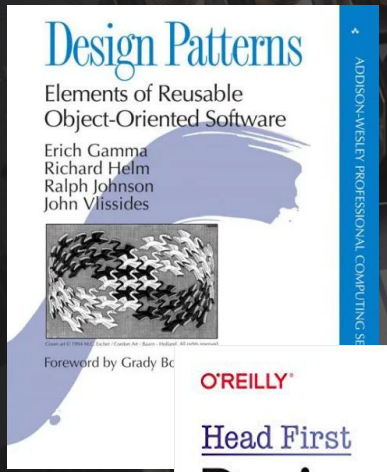
These principles apply to:

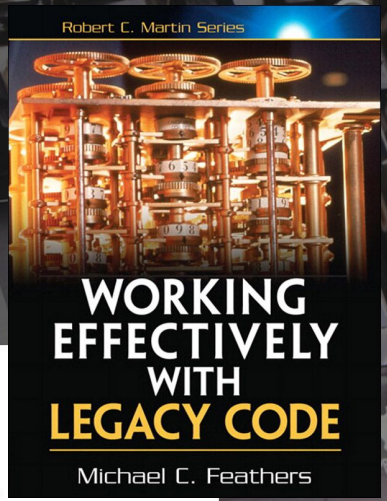
- Individual methods or classes
- And generalise to the architecture

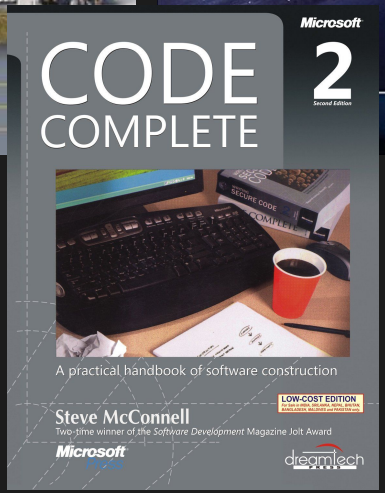
Recommended Reading

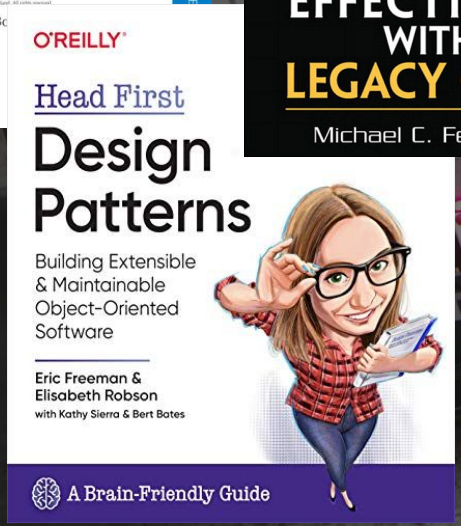












I hope you want to know more!

Go to Elk's GitHub (**github.com/elk-audio**) for the Sushi repository.

Then check out **elk-audio.github.io/elk-docs**

For additional documentation

If you want to give TWO a try, it's at: **controlmedia.art**

For the academics among you, you can search with my name on Google Scholar.

Questions?