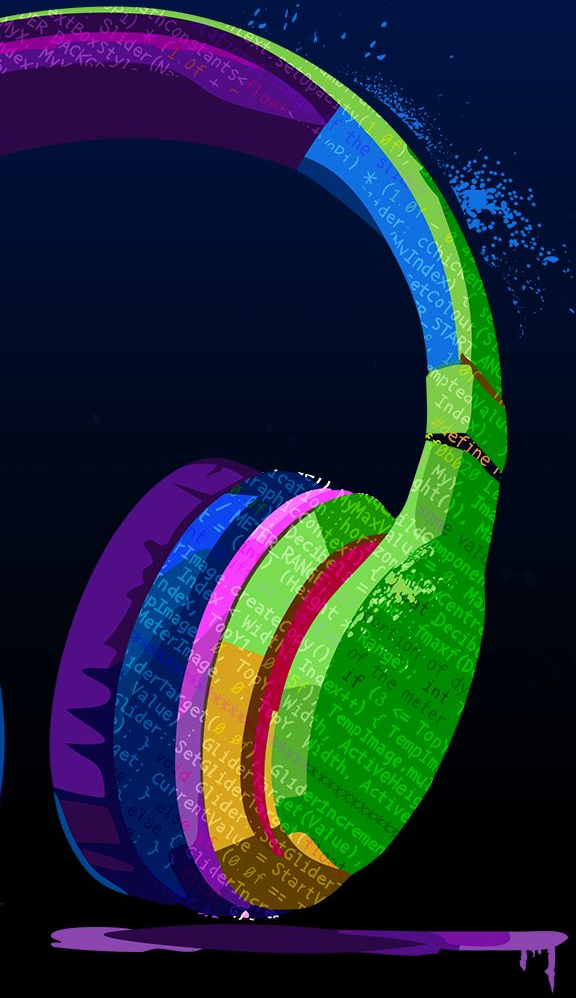


ADC24 Bristol

DIGITAL AUDIO WORKSTATION ARCHITECTURE

EVALUATION AND EVOLUTION

ILIAS BERGSTRÖM

Presenter

Ilias Bergström

Worked at

- Elk Audio
- SCISS Planetarium software

Background in audio / media tech research (PhD, Post-Docs)

Introduction

Last year: “Architecture of Digital Audio Workstations”

- Broad introduction
- Overview of the architectures of 2 DAW applications

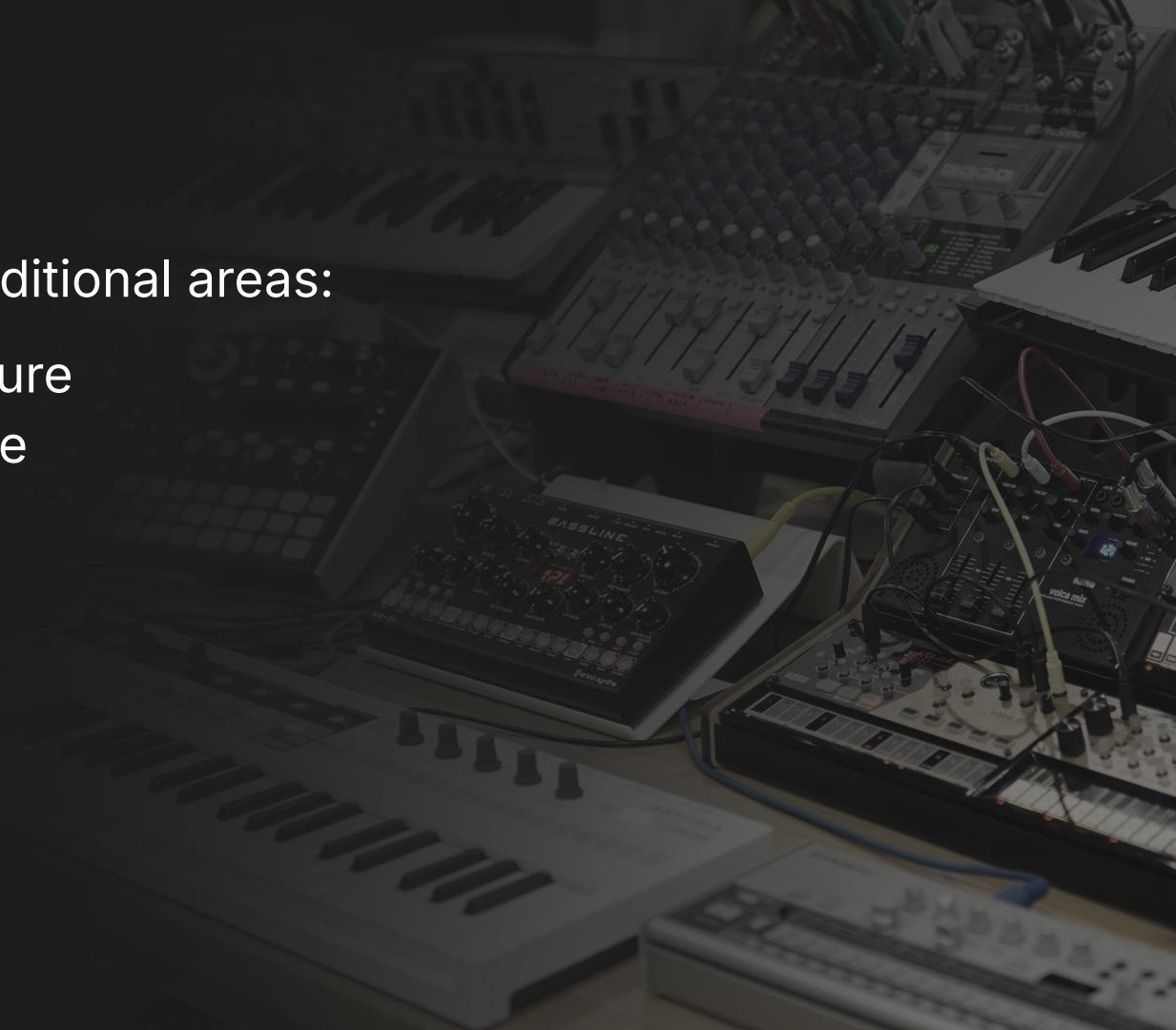
Covered a great deal of ground

...had to leave out important topics

This talk

Concentrates on two additional areas:

- Evaluating Architecture
- Evolving Architecture



Why bother?

- “Why make things harder than they need to be, just write the code already?”

- All about “just enough” Architecture and Design

Managing Complexity!

- I’ve worked on codebases spanning million+ L.O.C.
  - That “Emerged” from successful small projects
  - Very hard to understand!
- I’ve created several large applications from scratch
  - Learned from many mistakes along the way!

Our brains can’t cope with too much complexity

- Need repeated patterns, and recurring ideas
  - In architecture and design
  - Carefully selected & documented
  - To understand code, and explain to others
- Code should be written mainly to be read!

Defining Architecture vs Design

Definition of Software Architecture

Used in ADC ‘23 talk:

“The set of structures needed to reason about the system,

which comprise software elements,

relations among them, and properties of both”

(Bass, Clements, Kazman)

Further Definition

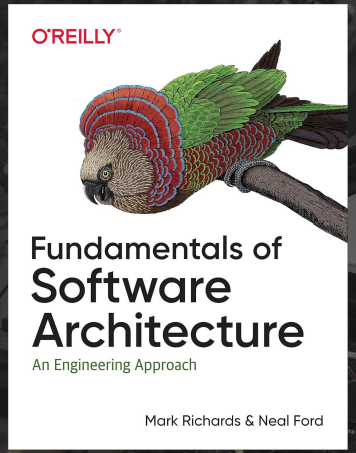
First law:

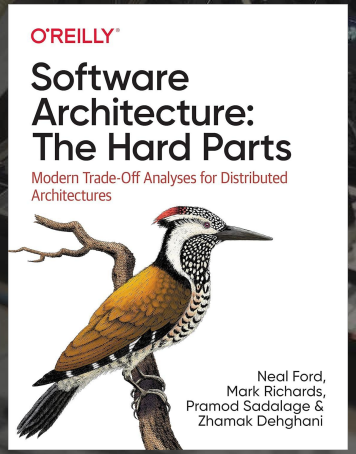
- Everything in architecture is a trade-off

Second law:

- ‘Why’ is more important than ‘How’

Architecture is about the ‘Stuff’ that is hard to change later





Architecture vs Design

Constant struggle:

- Keeping architecture and design as separate but related activities

“Never-ending argument”

Architects must understand design:

- But first *why* one choice has better trade-offs than another

Game Engine Architecture (book)

Established in its field

Mentions 1 architecture pattern:

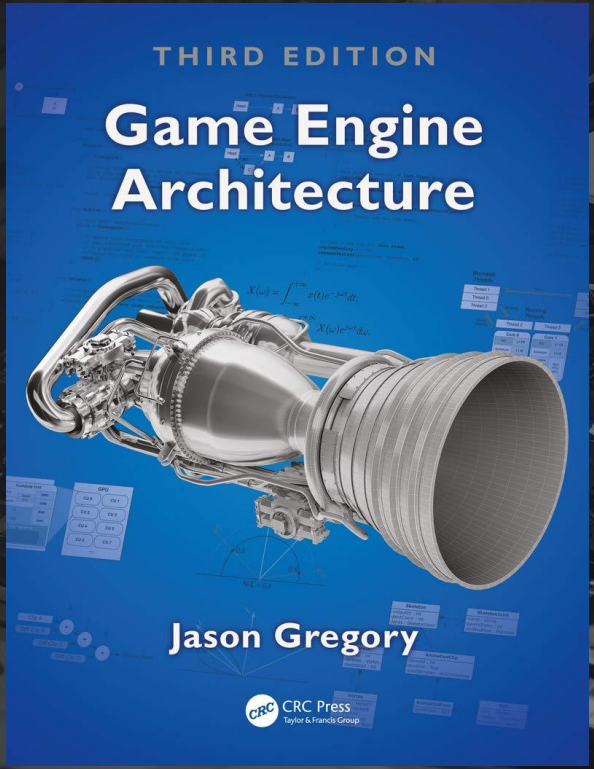
Layered architecture

And only Iterator & Singleton Design Patterns

Rest of book:

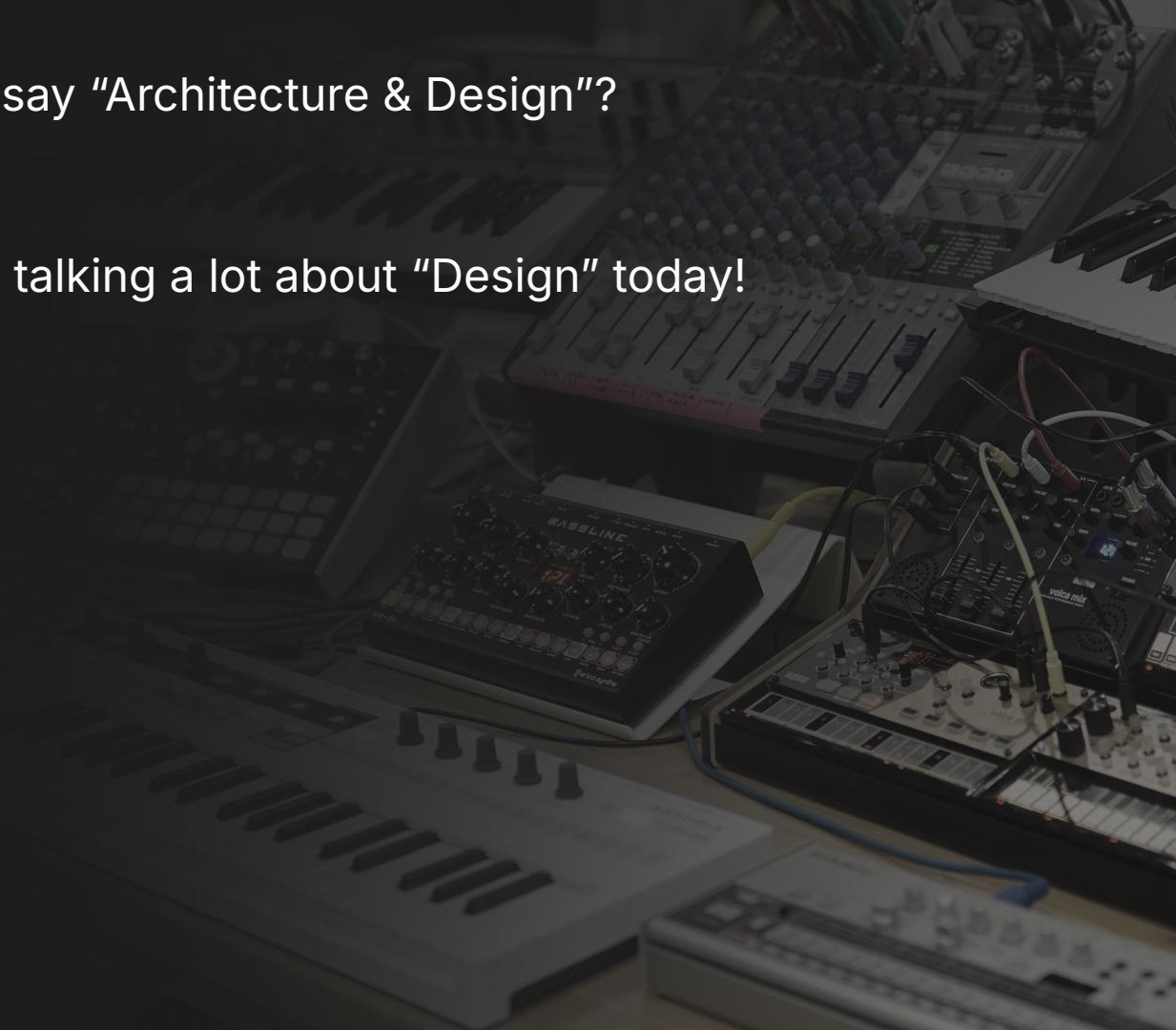
- Low-level optimization
- Memory management
- Lock-free programming

All important topics, and a good book!



Maybe my talk title should say “Architecture & Design”?

Regardless of labels: I’ll be talking a lot about “Design” today!



Evaluating Architecture

Criteria: What informs the architecture?

Functional Requirements:

- *“What the software needs to do”*

Design Constraints:

- *“...with zero degrees of freedom”*

These are used to formulate...

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”

Richards and Ford call these “Architecture Characteristics”

I’ll stick with “Quality Attributes”

Quality Attributes

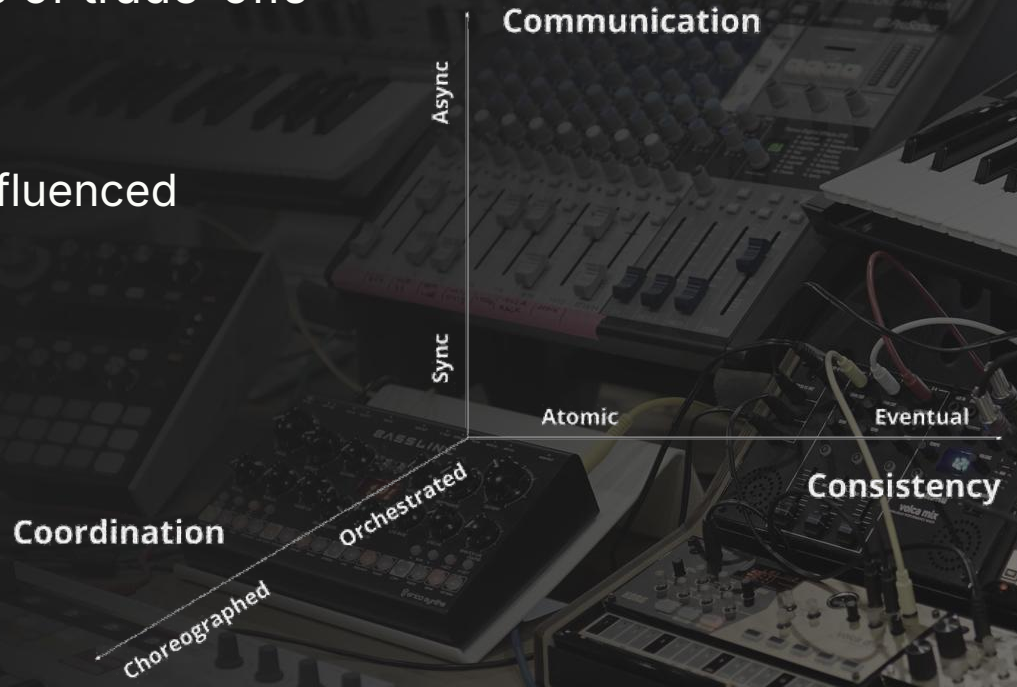
- Availability
- Interoperability
- Modifiability
- Performance
- Security
- Testability
- Usability

You use those that apply to your system at hand

Evaluation: Balancing between sets of trade-offs

No silver bullets, or best practices

“The architectural decision space is influenced by three interlocking forces” →



...Not a quantitative decision

Tradeoffs across several dimensions

In books they apply to Distributed Software

In my experience they apply also to DAWs

Evolution

Evaluation will be needed throughout:

- While creating the initial architectural design
- While maintaining the software, as requirements change

“Software is finished when no-one uses it”

超時空要塞
マクロス
MACROSS

Brief Recap: The two applications

Sushi

Elk’s headless DAW - Central in Audio OS

Has appeared 4 years at ADC (19, 22, 23 & now)

“Live” - more like Mainstage than Logic

So, e.g., it lacks timelines and a GUI

For embedded use:

- Originally for embedded hardware (Elk Audio OS)
- Later also “embedded” in desktop software and plugins

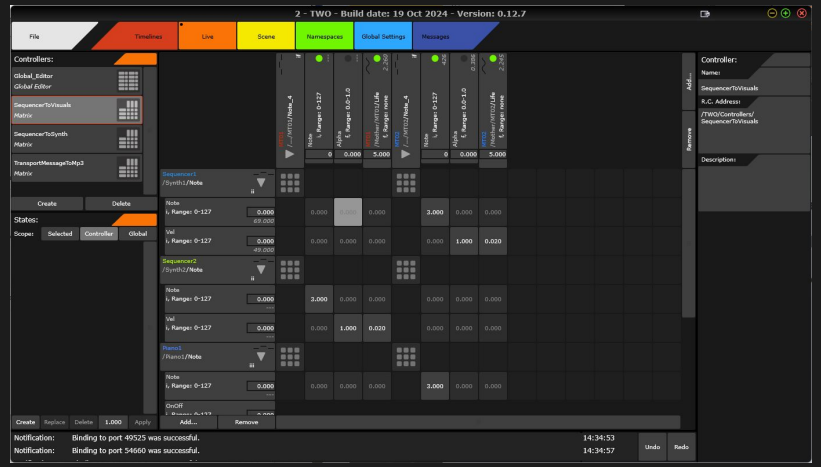
TWO, a “Media Control Workstation”

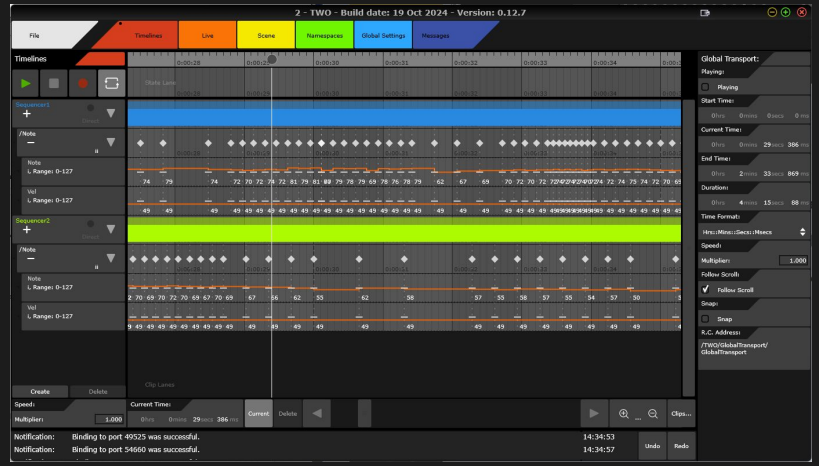
Like a DAW

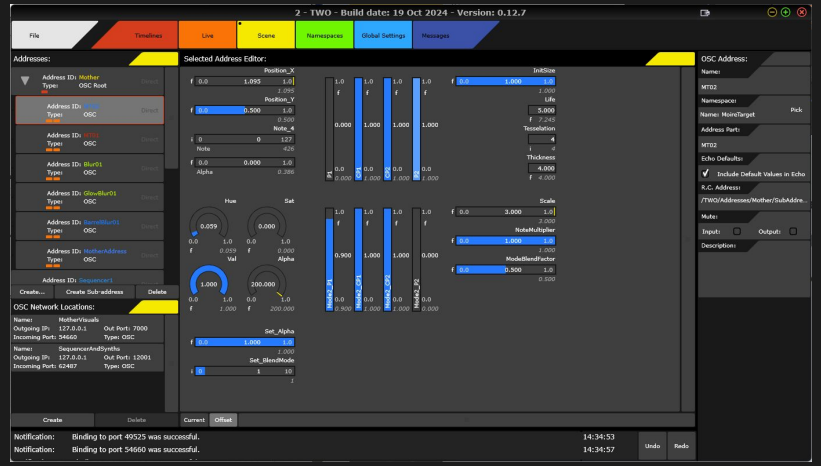
For all “Time-based media”

For many types of media control signals

OSC, MIDI, sACN, etc







Requirements
For TWO and Sushi

Why two different DAW’s?

Sushi and TWO, are very different

Both are DAWs

Where they differ - wildly:

- Requirements
- Resulting trade-offs

Little overlap between their target uses

If anything, I could integrate TWO to control Sushi. Maybe one day :)

The Quality Attributes listed are cursory, or this talk would be too long

But they'll do fine as examples!

Quality Attributes: Sushi

Sushi QA: Performance

- Highly optimised for embedded hardware platforms
- Limited resources available - compared to a desktop computer
- Latency and CPU efficiency are top priority

It’s for:

- Standalone instruments
- Effect pedals
- PA speaker arrays
- Even wearable devices (e.g. headphones)

Sushi QA: Interoperability

- Directly interface with ultra-low latency audio frontends
- Also support standard audio frontends, across desktop platforms
- MIDI on embedded & desktop
- Host plugins in a number of formats

Sushi QA: Modifiability

- Easily portable to new platforms
- With own Audio / MIDI frontend requirements
- Scale for their resources and needs
- Easily support new plugin formats

Sushi QA: Testability

- High unit-test coverage
- Real-time and non real-time integration tests
- Full system tests

Sushi QA: Usability

- Not an end-user application
- Usability requirements are purely with developers in mind
- Elk wants 3rd party developers to use Sushi

Needs

- Very clear API
- Easy to build, update, maintain
- Predictable through changes!

Quality Attributes: TWO

TWO QA: Performance

- Receive, generate, modify & forward, record, and play: Large amounts of control signals, at high (“interactive”) control rates
- Does not deal with audio-rate data
- UI should “feel” real-time:
  - Meant to be useable as a live performance instrument
  - Like Ableton Live can be used on stage

TWO QA: Interoperability

Support widely used control protocols

Only some initially

- OSC, MIDI, sACN (networked DMX)

Allow easily adding more, e.g.

- Legacy DMX & ArtNet
- MIDI 2.0

TWO QA: Modifiability

- Allow adding new protocol modules - plug-ins
- Expose core functionality for end-user scripting

TWO QA: Usability

- Crucial for end-user innovation application: It’s a creative tool
- Literature from NIME community applies
- Too vast a topic. Recommend:
  - “Usability Evaluation Considered Harmful (Some of the time)” [Greenberg and Buxton, CHI 2008]

    Great intro to tradeoffs & considerations!

  - Bill Buxton “Sketching User Experiences”

Architectural Decision Records

A great idea to keep "Architectural Decision Records"

- To write the outcome and process leading to architectural choices
- Throughout software’s lifetime

Otherwise:

- Knowledge will be lost
- Reasoning will become “legacy” and eventually “debt”

ADR Example

Title: Short description

Status: Proposed / Accepted / Suspended

Context: What forces the making of the decision?

Decision: Decision and justification

Consequences: What is the impact?

[Compliance, Notes]

Each ADR can be a page or less - no need for long essays

Concrete Examples

Changing requirements → architecture & design changes

Sushi was originally designed to run as a standalone application

- But, for internal and client projects
- It had to be refactored for use as library, and inside a DAW plugin

Need for “Reactive” Frontends

- A new “Reactive” audio frontend was needed
  - I.e. it does not directly interface with an audio API

- The frontend is invoked through the host audio callback
- Clock is owned by the host application

The old code

Sushi’s legacy Main():

- Reading / parsing configuration
- Instantiating Audio, MIDI and control frontends
- Running the “main loop”
- Deallocation and shutdown

A lot!

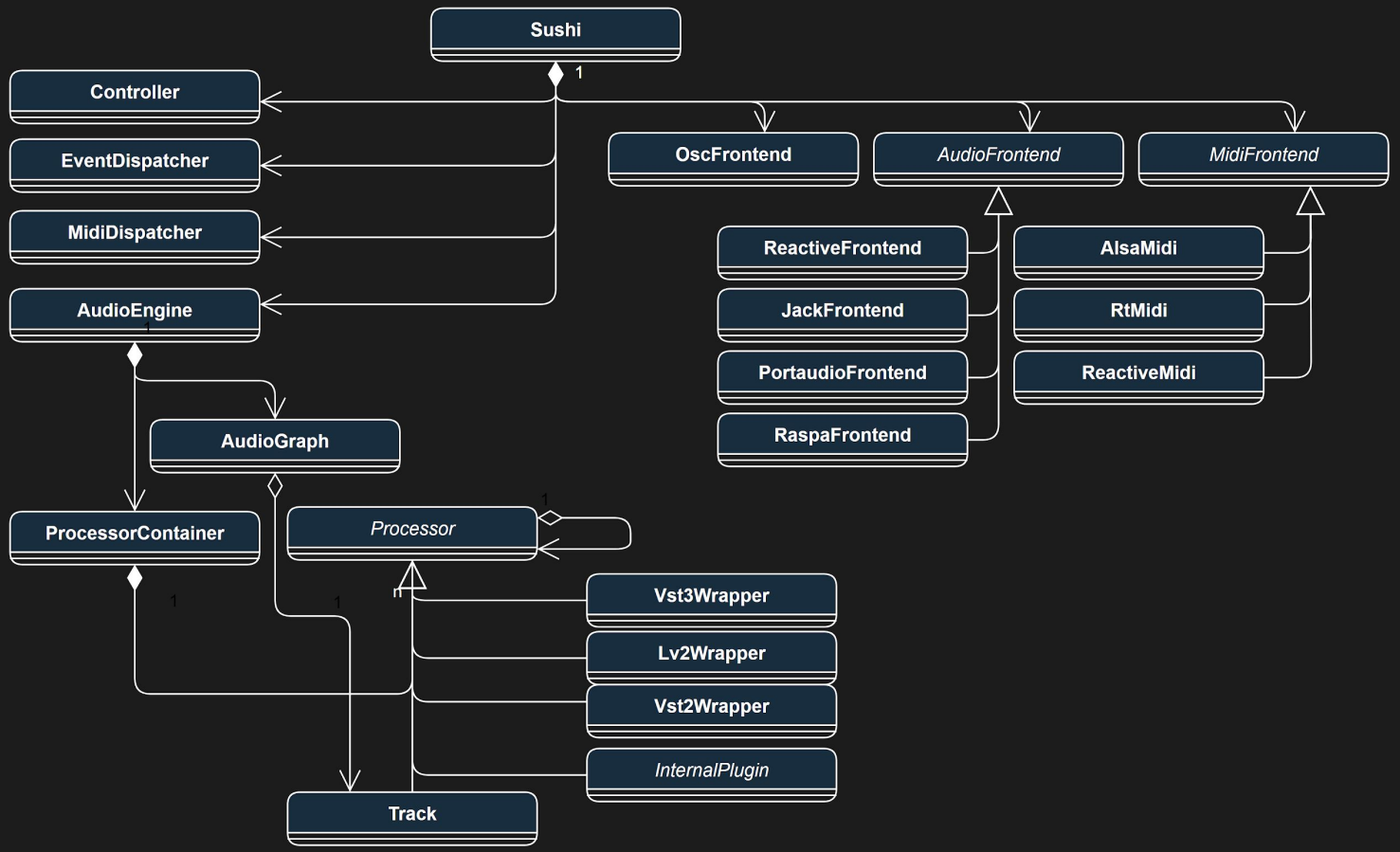
But it was originally small

...until, a few years later, it wasn’t

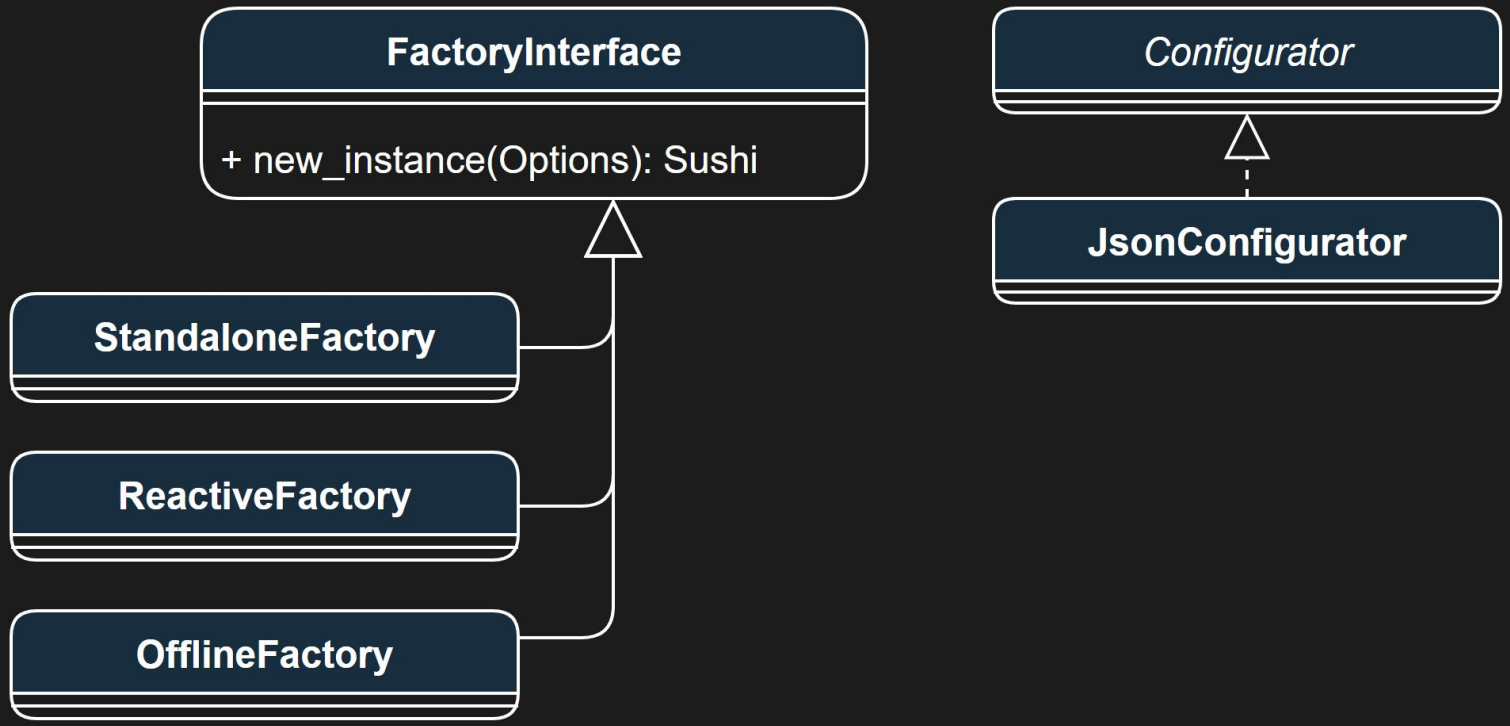
A great application for techniques in: “Working effectively with legacy code”

Refactoring!

Made up of sub-controllers - is the external interface.



Instantiation and Configuration



ADR:

Title: Sushi Instantiation Factory Refactor

Status: Accepted

Context:

- Main() method has too many responsibilities
- Sushi needs to be embeddable as a library
- Instantiation configuration is complicated & error prone

Decision:

- Break up main, embed instantiation into required factories, which assemble Sushi class
- Sushi class manages lifetime, and main loop

Consequences:

- Unit-test coverage improved, now covering brittle instantiation & configuration
- Code is easier to read, with improved separation of concerns
- It is easy and safe to add complex new frontend and controllers for “Reactive” use

Adding feature & Considering pre-existing architecture

Sushi didn’t have send-return in audio graph

Then, it was requested

Compromise:

- Didn’t want to replace existing design
- Most clients don’t need it - and prefer simplicity
- We created internal Send/Return plugins
- Latency penalty is a single buffer

Upside:

- Only used if needed
- Existing code is unchanged
- The effort is small

Dave Rowland’s Audio Graph talk

- I will not go into details
- This was the work of Gustav Andersson of Elk
- Dave Rowland gave an in-depth presentation at ADC 2020, on the new Traction engine audio graph

ADR:

Title: Sushi Send-Return

Status: Accepted

Context:

- Sushi’s audio graph lacks send-return, which some clients require
- Many clients also require running Sushi on minimal hardware, with maximal performance - and don’t need send-return

Decision:

- Implement send-return using internal plugins

Consequences:

- One-buffer latency when using send-return
- But when they’re not used, Sushi’s audio-graph is unchanged

Unpredicted complex feature addition

Adding timelines to live DAW (TWO)

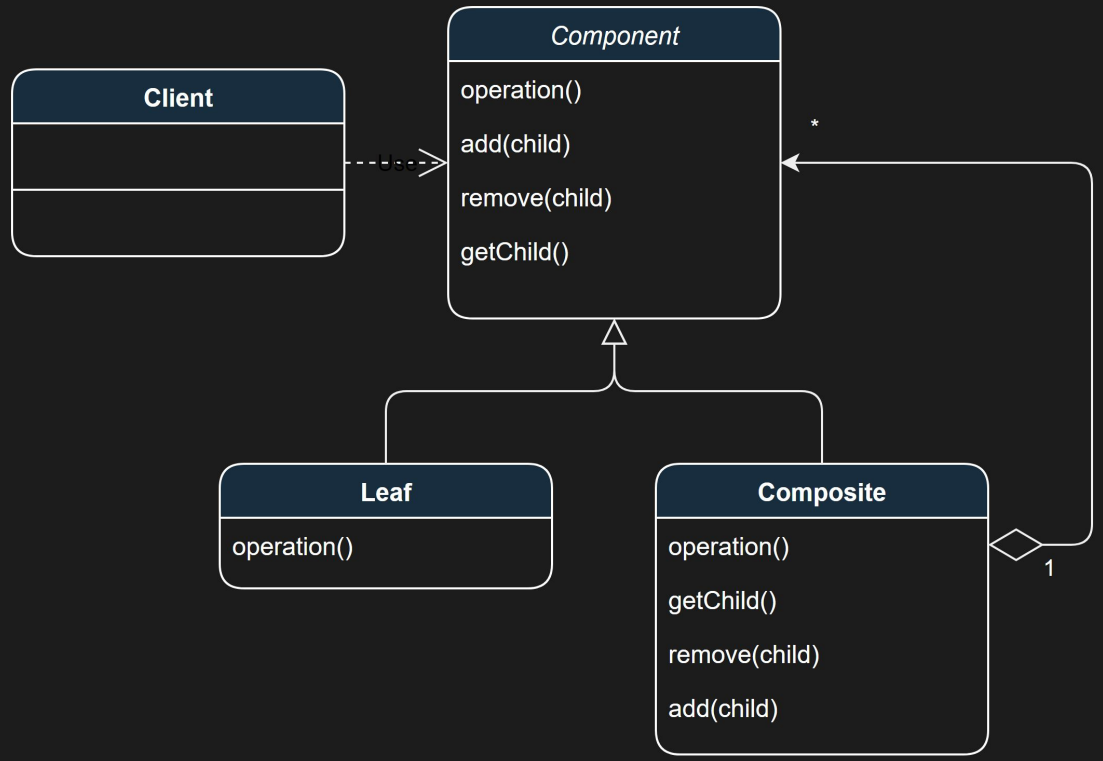
- Recording
- Playback
- Editing

Originally TWO was conceived only for live use

Eventually, Timelines seemed like a useful addition!

The original architecture was permissive enough to allow it without significant alterations

Composite Pattern



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

Lets clients treat individual objects and compositions uniformly.

Besides Composite, I added

- String ID - making each component addressable over OSC
- Listener list for change subscription

Convenient and efficient enough for few 100s of “components”/nodes

Simple, reusable and recognizable across code

Specifically for keyframes, I break with that model

- Model was slow & memory-intensive
- A recording can contain potentially millions of messages

That’s not a problem with the original design - the two can co-exist

For keyframes

- Each key has to use minimal memory
  - No ID, no listeners, only payload and time
- Naturally ordered by “timestamp”
  - Adding to end and seeking is cheap
  - Reordering rare, so it can be expensive
- Allocated with memory coherence
  - So that they’re efficiently traversed

I *could* have kept Composite, refactoring out ID and listeners

But this was cleaner - a change concentrated in one place

Data-Oriented Design

Is big for Game Engines and not only

Arthur Carabott also discussed this on his Monday ADC talk

What I have now gave satisfactory performance so I stopped here

I write TWO myself in my spare time

If this was my day-job I'd also use custom allocators, and tweak keyframe element size down to the absolute minimum

Maybe I'll do it anyway one day

ADR

Title: Timeline Keyframes do not use Composite

Status: Accepted

Context:

- Timelines addition requires potentially many hundreds of thousands of keyframes
- But, Composite base-classes have a memory footprint that then grows unwieldy

Decision:

- Do not use Composite base-classes for Timelines keyframes
- Duplicate model traversal, loading and saving for these

Consequences:

- The memory footprint can be precisely controlled to the bare minimum needed
- There is unnecessary code duplication for timeline keyframes

Address sub-optimal design

What to do when a design choice is found to be suboptimal?

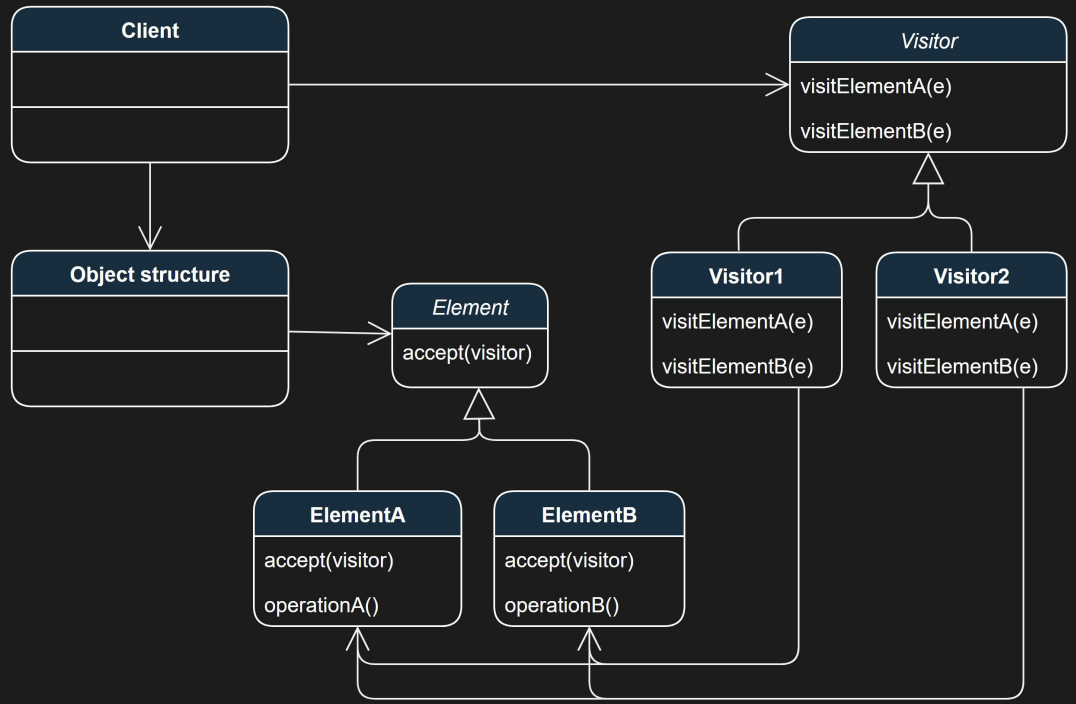
- Mistaken design creates “debt”
- There’s a penalty in addressing the issue
- The code has grown and become harder to correct

Living with it

Not - yet worth the effort

In TWO, the visitor pattern is used to iterate over the composite structure of the model

Visitor Pattern

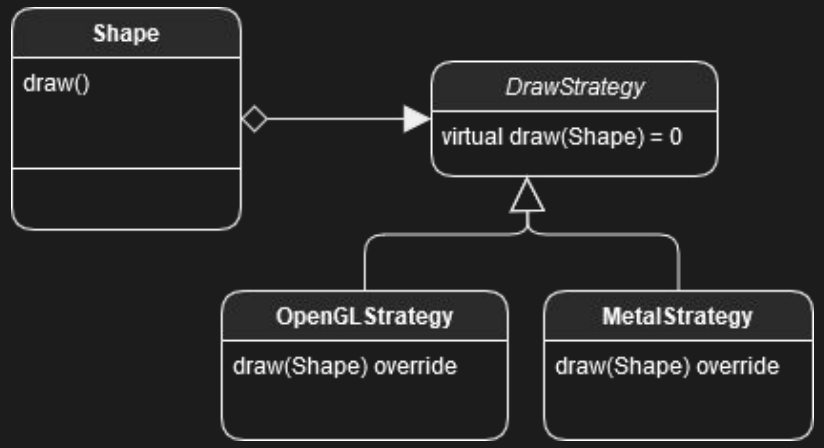


Represents an operation to be performed on the elements of e.g. a Composite structure.

Original GoF is verbose and coupled.

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

Strategy Pattern



Defines a family of algorithms, encapsulates each one, and makes them interchangeable

Lets the algorithm vary independently from the clients that use it

Operations and model are kept separate

Two dimensions:

- Several Types
- Different Operations to these

Visitor and Strategy both achieve separating concerns

Tradeoff:

- Visitor →Flexible Operations, Stable Types
- Strategy →Stable Operations, Flexible Types

The idea was there would be many operations

But that assumption proved to be wrong!

Strategy allows nodes with operations as plugins

A library with

- New model nodes (e.g. for MIDI 2.0)
- Operations for these (e.g. serialization)

Can be added, with a plugin API

With Visitor this isn’t possible: it needs to know about all nodes.

ADR outline

Title: TWO uses Visitor for model operations, when Strategy is better

Status: Suspended

Context:

- The model Composite structure’s nodes vary frequently, while operations do not
- Modifying Visitors when nodes are added/changed, is cumbersome
- An API for adding new nodes through dynamic libraries is not possible

Decision:

- Wait with the refactor of using Strategy, until there is a user-facing need (e.g. plugins)

Consequences:

- The codebase will keep being a bit harder to work with than ideal

Dealing with it

A “God Class” gradually appeared and needed removing

It started innocently enough, as a “Controller” class, during the first implementation of TWO

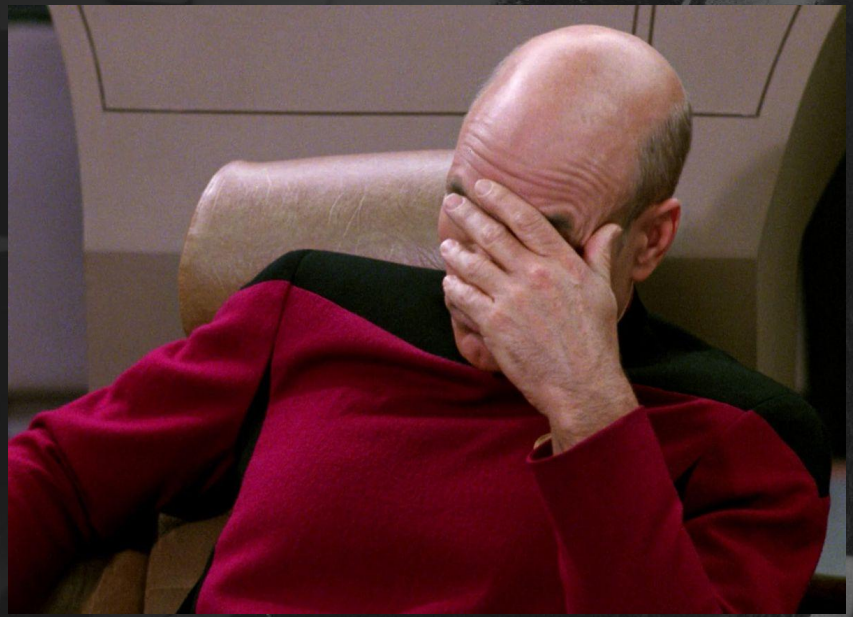
This exposed an API for

- Accessing and editing Model (Composite structure)
- Scheduling actions and allowing their undo/redo
- Model validation
- Tracking active selections
- etc

You see where this is going

Each object instance in TWO had a pointer to this Controller

I didn’t know better 15 years ago :)



It wasn’t so bad in the beginning

No choice is necessarily always bad - it depends

A small God may even be fine in a smaller program

It is better than using Singletons for the subsystems:

- It allows lifetime management
- And mocking of the subsystems exposed

But as a Trojan horse:

- Ends up having way too many responsibilities
- Separation of concerns becomes very diffuse
- Build times become unnecessarily long

ADR outline

Title: God Class should be replaced with task-specific modules

Status: Accepted

Context:

- What started as a small “Controller” has over time grown to have too many responsibilities.
- The code is hard to read and change, and build times are very long

Decision:

- Break up the “controller” into task-specific modules
- “Dependency-Inject” each into nodes that need them only, and remove “controller”

Consequences:

- Code is easier to read and modify
- Build times are reduced
- Easier to unit-test

Discussion

“Architecture & Design”?

For microservices, web development:

Architecture & Design are easily distinguished

- Each microservice is a few thousand lines of code
- API between them is clear
- “Design” is a simpler problem
- Microservices are easy to tell apart

“Architecture & Design”?

DAWs & Game Engines:

- Often a single “monolith”, of even millions of lines
- Layered / service based archietcture

Delineation is harder - at what level of granularity do you draw the line?

Moreover:

- “Design” can be much more involved
- Actual design “details” *are* crucial decisions
- Require informed trade-offs
- Are “Stuff that is hard to change later”

With the Architecture vs Design distinction in mind:

Large part of my presentation last year concerned design, more than architecture

And some of the examples I brought up, some fit the criteria for “architecture” of Richards and Ford

No need to nitpick - but worth noting!

Connect QA’s to decisions: Sushi Instantiation

- Initially single main(), while verbose, was not a problem
- Tried and tested code
- Edge-case bugs only triggered by a highly unusual conditions
- With the new requirement for Sushi to be embeddable, it did become a problem
- An explicit, carefully considered design was needed, where there previously was only an “Emergent” one

Connect QA’s to decisions: Sushi send-return

- API “Usability” QA - Consistency
- Also for performance QA
- Avoiding change in deployed code, kept us from refactoring the audio graph
- Implemented design allows send-return, only if explicitly needed
- And we did not have to engage in a costly rewrite

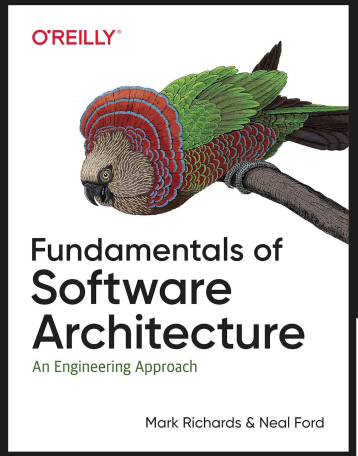
Connect QA’s to decisions: TWO Timelines

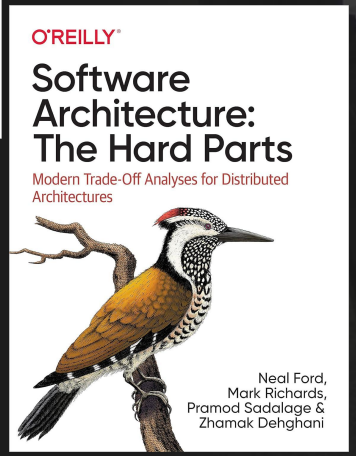
- The existing model would be fine for keyframes if they were few
- For performance QA, the break was needed

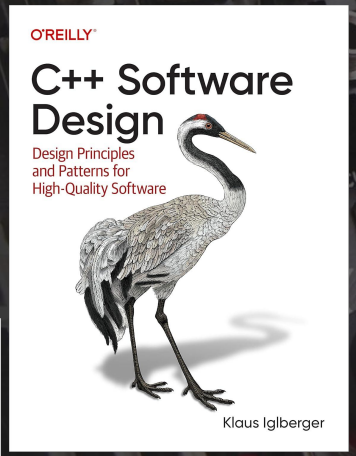
Connect QA’s to decisions: TWO Strategy refactor

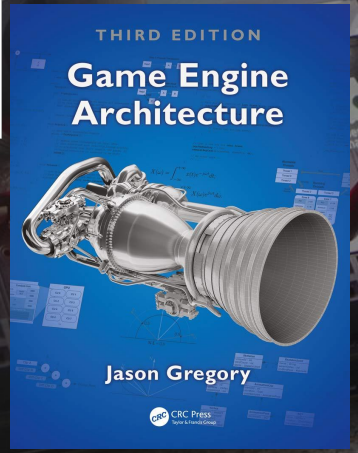
- Connecting to Extensibility QA
- Plugin API is planned for TWO’s future, not now
- So, the refactor is documented as Suspended in an ADR
- Stating that design is good enough for now

(more) 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?