Algorithmic Architecture

Performant Architecture in the Evolving Regulatory Landscape

Jamie Allsop
DSP background with a PhD in adaptive framework design

focused on C++ & standards work

passionate about agile

fiddle with python—pypi/cuppa for Scons

ended up at NYSE Euronext

now director at clearpool.io
➤ regulations and change
➤ problems, people and software
➤ architecture and performance
UK Nationalises Northern Rock

More Subprime losses for Bear Stearns, Citigroup & Merrill Lynch totalling $34 Billion

More Subprime losses for UBS & Morgan Stanley totalling $19 Billion

Subprime losses for UBS, Citigroup, Merrill Lynch, Deutsche Bank, Wachovia, Credit Suisse, Bank of America & Barclays totalling $39 Billion

More Subprime losses for UBS & MBIA of $21 Billion

More Subprime losses for UBS, Citigroup, Merrill Lynch & RBS now totals $122 Billion

Losses for HSBC of $17 Billion

Losses for UBS, Citigroup, Merrill Lynch & RBS now totals $122 Billion

FreddieMac stops buying subprime mortgages & MBS

Bear Stearns liquidates two MBS funds

Subprime losses for HSBC of $17 Billion

SEC: Regulation NMS: Order Protection (Trade-Through) Rule, Market Access Rule, Sub-Penny Rule, Market Data Rules

SEC: Uptick Rule Repealed


Short Selling Bans
<copy>

**2008**
- Lehman Brothers Bankrupt, Merrill Lynch bought by BoA for $50 billion
- Losses for UBS, Citigroup, Merrill Lynch & RBS now totals $122 billion
- DIJA falls 1874.19 pts in all-time worst weekly performance
- TARP financial stabilisation package enacted
- Greece bonds get Junk rating
- 4547.05
- "Fear"

**2009**
- FannieMae & FreddieMac placed into Gov. Conservatorship
- Greek bonds get Junk rating
- SEC: Reg SHO - Rule 201: Alternative Uptick Rule—Short Sale-Related Circuit Breaker
- Original BASEL III Proposal
- Dodd-Frank Reform Act Passed (inc. Volcker Rule)
- SEC: Ban on Stub Quotes
- Bailouts for Big Banks
- TARP financial stabilisation package enacted

**2010**
- €85 Billion EU rescue for Ireland
- €110 Billion EU/IMF Greek bailout
- €78 Billion EU loan for Portugal
- TARP financial stabilisation package enacted

**May 6 Flash Crash**
- DJIA biggest ever intraday fall
- 998.5

© 2015 Jamie Allsop

Algorithmic Architecture
</copy>
€78 Billion EU loan for Portugal

EU rescue for Ireland

Billon EU/IMF Greek bailout

$110 Billion EU/IMF Greek bailout

$85 Billion EU rescue for Ireland

$78 Billion EU loan for Portugal

2nd Greek bailout of €130 Billion

Knight Capital Group accidentally deploy test software in prod resulting in $440 Million loss

Botched Facebook IPO

SEC launches MIDAS: Allows full depth market analysis

EU Initial MiFID II Proposal: covers OTFs, HFTs, Consolidated Tape, Derivatives, Increased Transparency

EMIR comes into force

SEC Rule 613: Consolidated Audit Trail (CAT) RFP

SEC: Sponsored Access Rule

EU Initial MiFID II Proposal: covers OTFs, HFTs, Consolidated Tape, Derivatives, Increased Transparency

SEC Market Information Data Analytics System (MIDAS) RFP

Botched Facebook IPO

SEC: Ban on Stub Quotes

Original BASEL III Proposal

May 6 Flash Crash DJIA biggest ever intraday fall

DJIA biggest ever intraday fall

998.5

2011

2012

2013
Knight Capital Group accidentally deployed test software in production, resulting in a $440 million loss.

SEC launches MIDAS: Allows full depth market analysis.

J.P. Morgan Chase pays record $13 billion fine for selling overvalued MBBs.

Successful Twitter IPO.

Reduction in FED stimulus of Bond Markets.

Phasing in of BASEL III / CRD IV Minimum Capital Requirements.

SEC: Reg SCI (Systems Compliance & Integrity).

SEC Rule 613: Consolidated Audit Trail (CAT) RFP.

“Too Big to Fail Banks”

Market Volatility

Insufficient Oversight

Unpopular Gov. Bailouts

Mistrust of Technology

Evolving Regulatory Landscape

Uncertainty
Regulations are currently seen as the best way to protect the markets and their participants from themselves.
But Regulations are a Moving Target

Regulations Change

for many reasons but ultimately they change

*stuff* happens and regulations are often seen as the answer

regulations create loop-holes that need plugged

regulations create industries that themselves need regulated
There are often Hard Constraints

minimum throughput?
availability?
disaster recovery?
average latency?

worst case latency?
proof of compliance?
audit trails?
accuracy of data capture?

many constraints driven by regulations
Let’s simplify this…

Business Rules

Performance Requirements

conflicting

Hard Constraints

Regulations

Interpretation

Technology Required To Satisfy All Requirements
Addressing Difficult Problems

- Regulations
- Proof of Compliance
- Audit Trails
- Business Models, Sales, Revenue
- Market Positioning
- Performance Constraints
- Performance Goals
- Evolving Hardware
- Future Proofing
- Business Decisions
“We fail more often because we solve the wrong problem than because we get the wrong solution to the right problem”

— Ackoff 1974

How can we classify problems?

Tame Problems

- may be simple or highly complex
- definitive stopping point
- consensus on how to proceed

- can be broken down into parts and solved
- solutions can be determined to be successful...or not

Gather Data → Analyse Data → Formulate Solution → Implement Solution
Messes

Organised complexity
• clusters of interrelated or interdependent problems

Systems of problems
• problems that cannot be solved in relative isolation from one another

Messes are puzzles – we don’t solve them instead we resolve their complexities
Tidy up that mess!!!

- not sufficient to just break the system into parts and fix components
- instead look for patterns of interactions between parts
- beware of identifying a mess as a tame problem—the evolving mess can be even more difficult to deal with
- interactive complexity—what can go wrong?
- coupling—the degree to which we cannot stop an impending disaster once it starts
Refactoring vs Bugfixing?
Conflicting **social** ethics and beliefs

Smart, informed people **disagree**

**Divergent** problems with no promise of a solution

**Evolving** set of **Interlocking** Issues and Constraints

Many Stakeholders

Constraints **change over Time**

**Wickedness**
Know your demons...

- No definitive Problem == No definitive Solution
- Cannot be solved by a Linear or “Waterfall” process
- Studying followed by Taming does not work
- No stopping rules
- Finished when we Exhaust Resources
- Solutions not Right or Wrong but Better or Worse
- Poor choices create more Wicked Problems
How we deal with problem complexity

Wicked

Mess

Tame

Dynamic Systems Complexity

Behavioural Complexity

High

Low

Reliance on Quantitative Assessment

Solution is Scientific

Reliance on Qualitative Assessment

Resolution is Social / Ethical / Political / Moral / Behavioural
Let’s consider the question of Healthy Markets
The markets involve people

The markets involve systems
What represents “good liquidity”?
- Tighter Spreads?
- Order Book Depth?
- What about “phantom” Orders?

Lots of People and Lots of Systems
Characteristics of a Healthy Market?

- Volatility?
- Liquidity?
- Data Access?

High Behavioural Complexity

Wicked Mess

Regulations Developed to Promote Healthy Markets
Approaches to Wicked Problems

- Iterative
- Timeboxing
- Qualitative Progress Assessment by Expert Stakeholders
- Getting the right Stakeholders together
- Communication Transparency
- Listening and Establishing Trust

Sounds a lot like Agile Development?
Agile and We’re Done?
What do we mean by Architecture?

➤ The product of Design and Implementation - what you see when you step back and look at your system

➤ Encoded in the Architecture are the choices made and compromises reached

Whose choices?
Whose compromises?
Another view on Architecture

Marketecture vs Tarchitecture?

Marketecture: Anything that is concerned with how revenue is generated for a product or how it is marketed as working, or how it is sold

Marketecture impacts Tarchitecture
Dangers in evolution

Marketecture is often driven by decisions that have no regard for the technical impact.

- Stakeholders change
- Goal posts move
- “Power without responsibility”
- Poor choices baked in early
- What’s most important?
Is often an observed **sketch** of the system

Actual architecture exists based on the **source code**

Pinpointing which aspects contribute to any characteristic of the system can be difficult

Changing it is usually hard
Agile Architecture

Is Fragile Architecture?

- Hard choices early so later choices are easier
- Evolving to an appropriate architecture
- Deferring choices to last responsible moment
- Natural calcification along the way

Evolves to better Architecture?
Agile Architecture is a good starting point—evolving to an appropriate architecture. Can we do better?
Let’s look at a real world example as a starting point
Following the Flash Crash the SEC launched an investigation into the causes.

The SEC were presented with architectural overviews of how the systems involved behaved, and how they were evolved.
Their focus was on Market Data Publication. Slow and delayed quoting was experienced during the Flash Crash.
What can we tell from looking at this picture?

Data flow
Networking
Queuing
Decisions
Processors
Data stores
Message Passing?
Synchronisation?
Bottlenecks?
Let's Fix That…

...so only trades are affected

Requirement!!! Trades must not be lost and must not be duplicated
See the difference? Analytics
Task Optimisation: Sync all outstanding writes at once
There are a lot of things we cannot tell from looking at the diagrams
What about...?

How are stale quotes handled during a recovery?
When and why are zero quotes published?
Are the recovery requirements reasonable?
Which version was in production at the time?
Did the system behave correctly?
Is there information to make that determination?
How was memory managed?
How many cores did deployment machines have?
Details, details, details...

Reasons why...

Risk Averse Business
Correctness the highest priority, then performance
Ultimate priority was performance
Worst case performance requirements
Architecture should evolve to improve performance
There were 2 versions live in production
A Story...
Not the Whole Story
Nice diagrams typically do not reflect the reality of a code-base

It would be nice if it did
Some things we can conclude

➤ Performance improved by doing the right thing
➤ Not by optimising existing behaviour
➤ Local optimisation only done when solution good enough
Let’s look at some possible future systems that all do the same thing...
The same thing in a different way with different trade-offs: Performance trade-offs
Improving Performance

Do the current thing better/quicker

Task Optimisation Approach

- Bubblesort $O(n^2)$
- DFT $O(n^2)$

Achieve the same thing in a different way

Algorithmic Optimisation Approach

- Timsort $O(n \log n)$
- FFT $O(n \log n)$

Sorting

Frequency Analysis
Prefer to optimise at the highest level possible
The fastest way to do something is not do it at all
Environmental Influences

➤ Architecture for wicked problems typically a “mess”
➤ Many stakeholders and evolving problem domain over time adds “wickedness”
➤ Decomposing and understanding interactions difficult
➤ Such architecture, good or bad, is often hard to reason about in a way that maps directly to code
➤ Favours Task Optimisation
We want to reason about this...

But we can only see this...
What we really want is an Architecture that

- favours algorithmic optimisation
- has a clear mapping to code
- allows an optimal solution
- is adaptive to a changing environment

an “Algorithmic Architecture”
Relies on being able to decompose the Architecture into discrete elements treating them as Building Blocks
We Achieve This By

➤ Exposing a Vocabulary *that can map to code and is*
➤ Decomposable
➤ Composable
➤ Independently Orderable
➤ Compactible
➤ Substitutable
1

Expose a vocabulary

the first step in moving towards an algorithmic architecture is to identify a vocabulary suitable for the domain

➤ implies decomposability
➤ implies extensibility
Must be a **common** vocabulary

A common vocabulary’s primary concern is not ensuring the best use in the description of a possible solution—rather it is focused on ensuring that all stakeholders can communicate sufficiently their position within it—it is **shared**
Must be **domain specific**

The vocabulary must support natural domain specific terms as understood by most stakeholders—it is not sufficient to simply adopt a general vocabulary based on general design patterns (but they help)
Identify *concepts*

Focus on identifying *concepts* over specific realisations. Refinement to more concrete terms is best reserved for supporting substitutable elements in an architecture.
Vocabulary Checklist

➤ must add in clarity of communication
➤ should have consensus on basic meanings
➤ does not need to be complete
➤ but should be sufficient to model basic systems
➤ may capture concepts at different levels in a system
➤ should be possible to describe a system
➤ vocabularies can grow and evolve
Decomposable

it should be possible to decompose the architecture into vocabulary elements that communicate the intent of the system

➤ implies partitioning interfaces
composable components can be assembled together to complete more complex tasks

➤ implies common approach to communication
Independently orderable

it should be possible to re-order components of the architecture that do not have an ordering relationship

➤ implies loose coupling
it should be possible to compact the architecture such that placeholder vocabulary elements can be optimised away

implies facilities to offset the cost of abstraction
Substitutable

vocabulary elements should be replaceable by differing implementations with differing performance trade-offs

➤ implies consistent, clean interfaces
Recommendations

- Define building block vocabulary elements
- Avoid shared state
- Favour message passing
- Make synchronisation points explicit in the architecture
- Support push and pull models
- Separate Data and Command paths
- Static Polymorphism for Performance
Simple Example

1. Design Using a Real Vocabulary of Real Components

2. Compact Architecture by removing conceptual components

3. Compile to Optimised Implementation with zero abstraction cost
Different Performance Trade-offs

Multi-threaded
Push Model

Single-threaded
Pull Model

Latency Agnostic
Coroutines?
Lock-free Queues?
Context-switches?

Scaling Agnostic
Single Process → Multiple Processes?
Single Core → Multiple Cores?
Single Server → Multiple Servers?
Code Mapping Example

Receive Packets

```c++
void on_packet(const data_packet& Packet, int InstrumentId)
{
    const void* Data = nullptr;
    size_t Size;
    if (Packet.read(Data, Size))
    {
        InboundPacketBus_.push(
            std::allocate_shared< packet_t >(Allocator_, 
                std::chrono::nanoseconds(clock_t::now().time_since_epoch()),
                SubscriberId_,
                InstrumentId,
                static_cast<const char*>(Data), Size)
        );
    }
    else
    {
        // log error: cannot read message
    }
}
```
```cpp
void process( const shared_inbound_packet& InboundPacket )
{
  if( SeqNum == ExpectedSeqNum )
  {
    ExpectedSeqNum = SeqNum + NumMsgs;
    GapHandler_.update_expected_seq_num( ExpectedSeqNum, ChannelId );
    while( shared_message_t Message = InboundPacket->pop_front() )
    {
      if( FramingStrategy_->incoming_message_triggers_send( OutboundPacket_->size(), Message->size() ) )
      {
        SeqNum_ += NumMsgsInPrevPacket_;  
        LastFrameTime_ = clock_t::now().time_since_epoch();
        OutboundPacket_->assign_seq_num( SeqNum_ );
        OutboundPacketBus_->push( OutboundPacket_ );
        NumMsgsInPrevPacket_ = OutboundPacket->header().num_msgs();
        OutboundPacket_ = std::make_shared<outbound_packet_t>( format::delivery_flag::original_message );
      }
      OutboundPacket_->push_back( Message );
      if( FramingStrategy_->packet_requires_immediate_send( OutboundPacket_->size(), Message->last_message_in_packet() ) )
      {
        SeqNum_ += NumMsgsInPrevPacket_;  
        LastFrameTime_ = clock_t::now().time_since_epoch();
        OutboundPacket_->assign_seq_num( SeqNum_ );
        OutboundPacketBus_->push( OutboundPacket_ );
        NumMsgsInPrevPacket_ = OutboundPacket->header().num_msgs();
        OutboundPacket_ = std::make_shared<outbound_packet_t>( format::delivery_flag::original_message );
      }
    }
  }
  else
  {
    // send command::category::notification - packet_discarded
  }
}
else if( SeqNum > ExpectedSeqNum )
{
  ExpectedSeqNum = GapHandler_.handle_unexpected_packet( InboundPacket, ExpectedSeqNum, ChannelId );
}
else if( SeqNum < ExpectedSeqNum )
{
  // log and ignore
}
}
Lastly...

Publish Packets

Vocabulary elements map directly to code

➤ Code still lives in separate ‘modules’
➤ Maintained and tested separately
➤ Communication through building block interfaces
➤ Abstraction cost removed but clarity retained
➤ Easy to change, fix, replace
Additional Benefits of a Common Vocabulary
Common Vocabulary → Tiered Structure

Source code is arranged in tiers facilitating a layered development structure and allowing critical code to retain high quality and performance.

- Projects, Deployments and Configuration
  - New York Equities Platform
- Product Libraries
  - matching_engine, gateway
- Domain Specific Libraries
  - gap_handler, format, session
- Domain Agnostic Libraries
  - multicast, bus, concurrency
- Standard & Thirdparty Libs
  - boost, std, asio
Stable Foundations

Tiers form a pyramid of code with the foundations formed by re-usable components and libraries of well tested code.
Developer Growth

➤ Allows different experience and skillsets to be catered to throughout the team
➤ Provides clear opportunities for progression and personal growth—minimising turnover and helping attract the best developers

Projects, Deployments and Configuration

Product Libraries

Domain Specific Libraries

Domain Agnostic Libraries

Standard & Thirdparty Libs

<table>
<thead>
<tr>
<th>Quality and Technical Knowledge required</th>
<th>Visibility on progress from a business perspective</th>
<th>Domain Specific Knowledge</th>
<th>Domain Agnostic Expertise needed</th>
<th>Team Lead Possibilities</th>
</tr>
</thead>
</table>
Contrast with Disparate Vocabulary

Disparate Vocabulary Hurts Development
- Product Siloed Development
- Competing underlying frameworks
- Explosion of Code

With A Common Vocabulary
Less is More
- Possible to adopt a Core Framework
- Product Building Focused more on Assembly
- Scales across Teams and Geographies
- Developer and Business share the vocabulary
Accelerated Development

Products based on shared framework
- Development rate increases over time
- Framework stabilises over time
- Developer turnover less impact

Minimal Toolchain possible
- Hiring Easier
- Maintenance Easier
- Faster Learning

C++ (core language, high perf, servers), Python (web-server, scripting, builds, test), Javascript (web-clients), SCSS (presentation), Postgresql (data storage)
We favour a more holistic view of development — one that puts people as a central aspect of architecture
Final Thoughts

In a highly regulated, ever-changing, environment with extreme performance constraints it is increasingly difficult to avoid full system rewrites to meet changing requirements.

Algorithmic architecture is primarily about adhering to certain principles and concepts where the goal is to facilitate clear understanding within complex and changing problem domains.

The goal of those principles is to allow optimisation (and general improvement) of an architecture to occur at the highest level possible—the architecture itself—allowing adaptivity and evolution.
Thank you for Listening

Questions?

clearpool.io

jamie.allsop@clearpool.io