C24

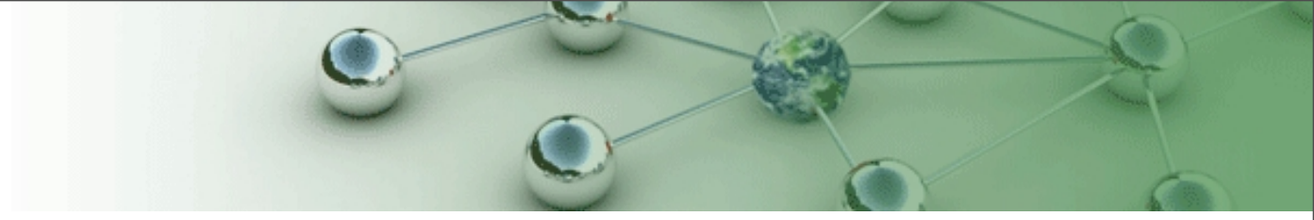
www.C24.biz

Financial Big Data

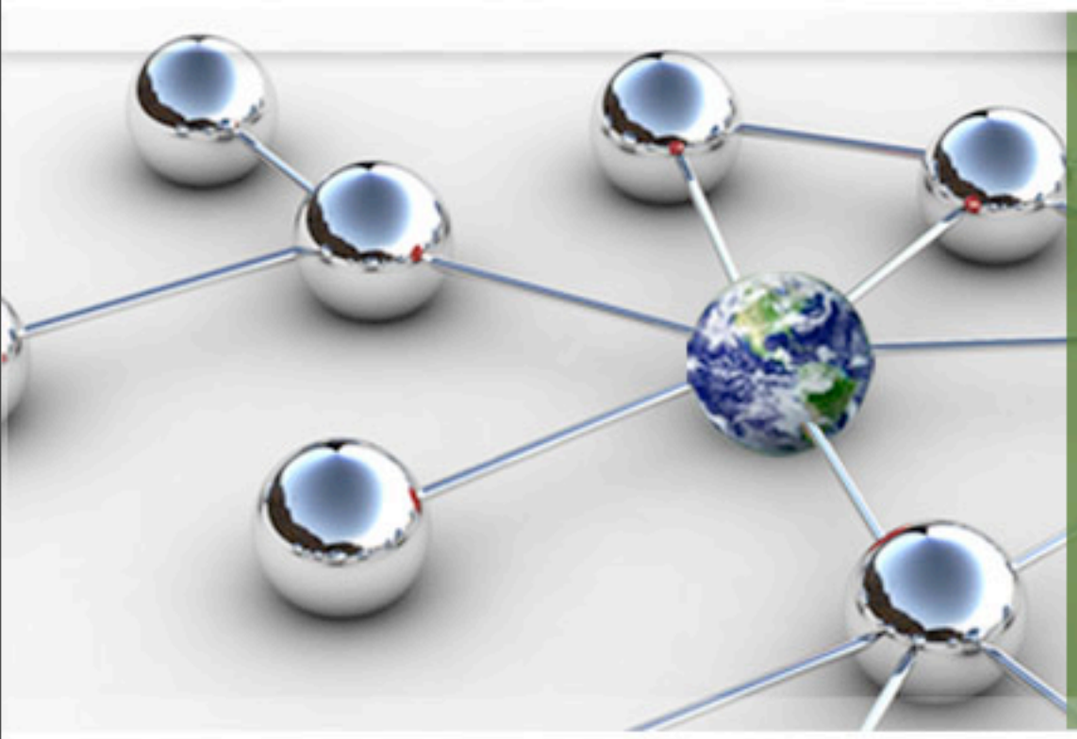
Loosely coupled, highly structured

Andrew Elmore

Agenda



- Why traditional RDBMSs aren't an ideal fit
- Why storage format is important
- Using NoSQL stores to store & query financial data
  - Examples using GemFire, GigaSpaces & Integration Objects
- Scaling complex processing using grids



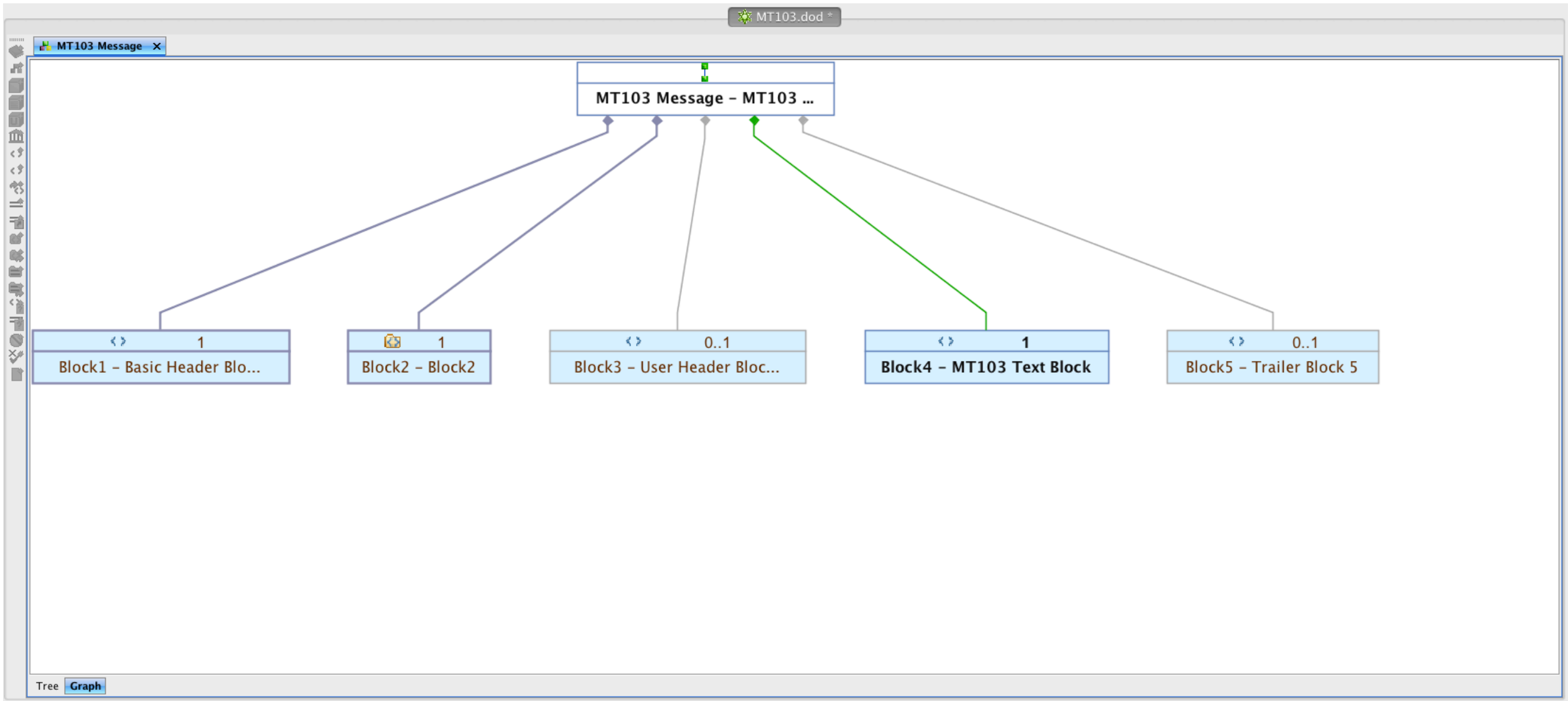
Message Structure

The Humble MT103

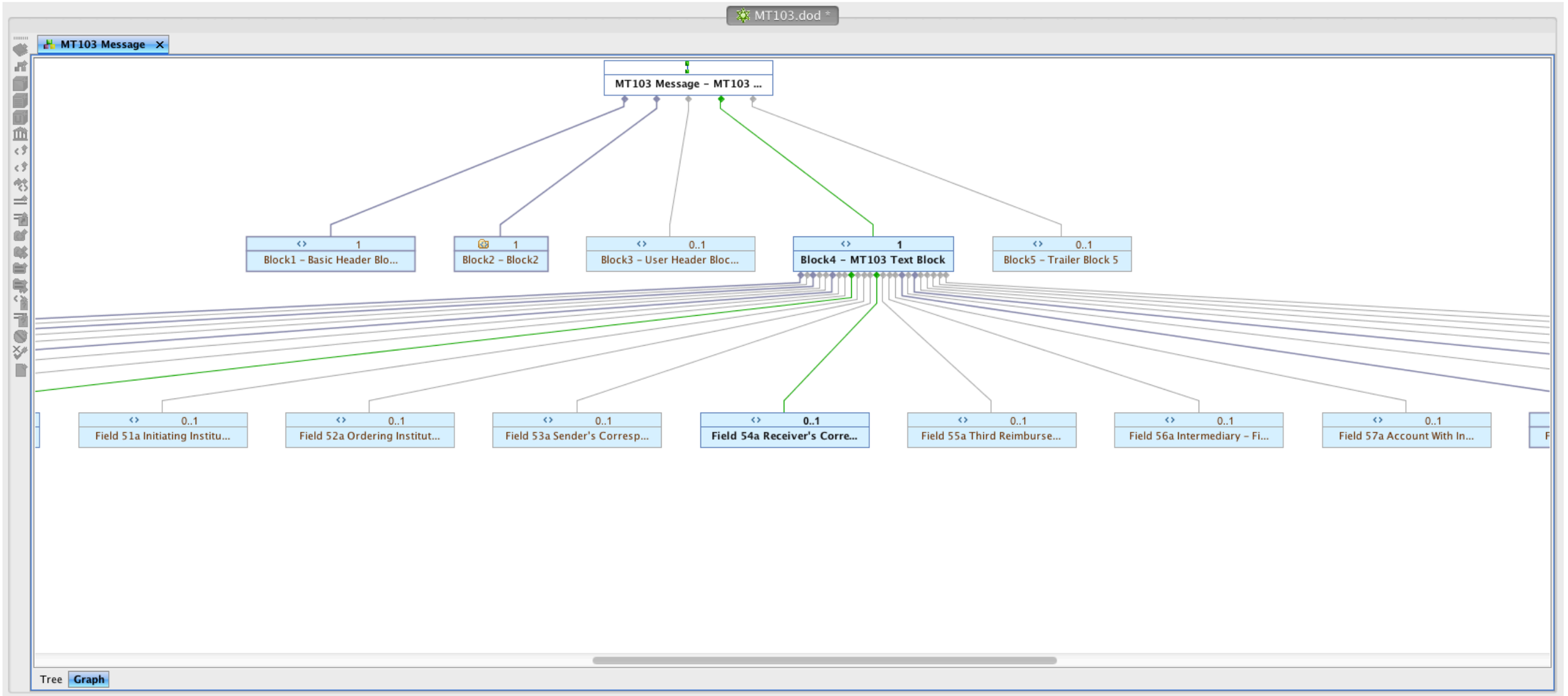
- A commonly used SWIFT message
- Used for credit transfers
- Mid-complexity compared to other SWIFT messages

```
{1:F01MIDLGB22AXXX0548100000}{2:I103BKTRUS33XBR
DN2}{3:{108:valid}}{4:
:20:8861198-0706
:23B:CRED
:32A:000612USD73025,
:33B:USD73025,
:50K:GIAN ANGELO IMPORTS
NAPLES
:52A:EFGHITMM500
:53A:BCITUS33
:54A:IRVTUS3N
:57A:BNPAFRPPGRE
:59:/20041010050500001M02606
KILLY S.A.
GRENOBLE
:70:Invoice: 100001
:71A:SHA
:72:/ABCD/narrative
//more narrative
/EFGH/narrative
-}
```

MT103 Structure

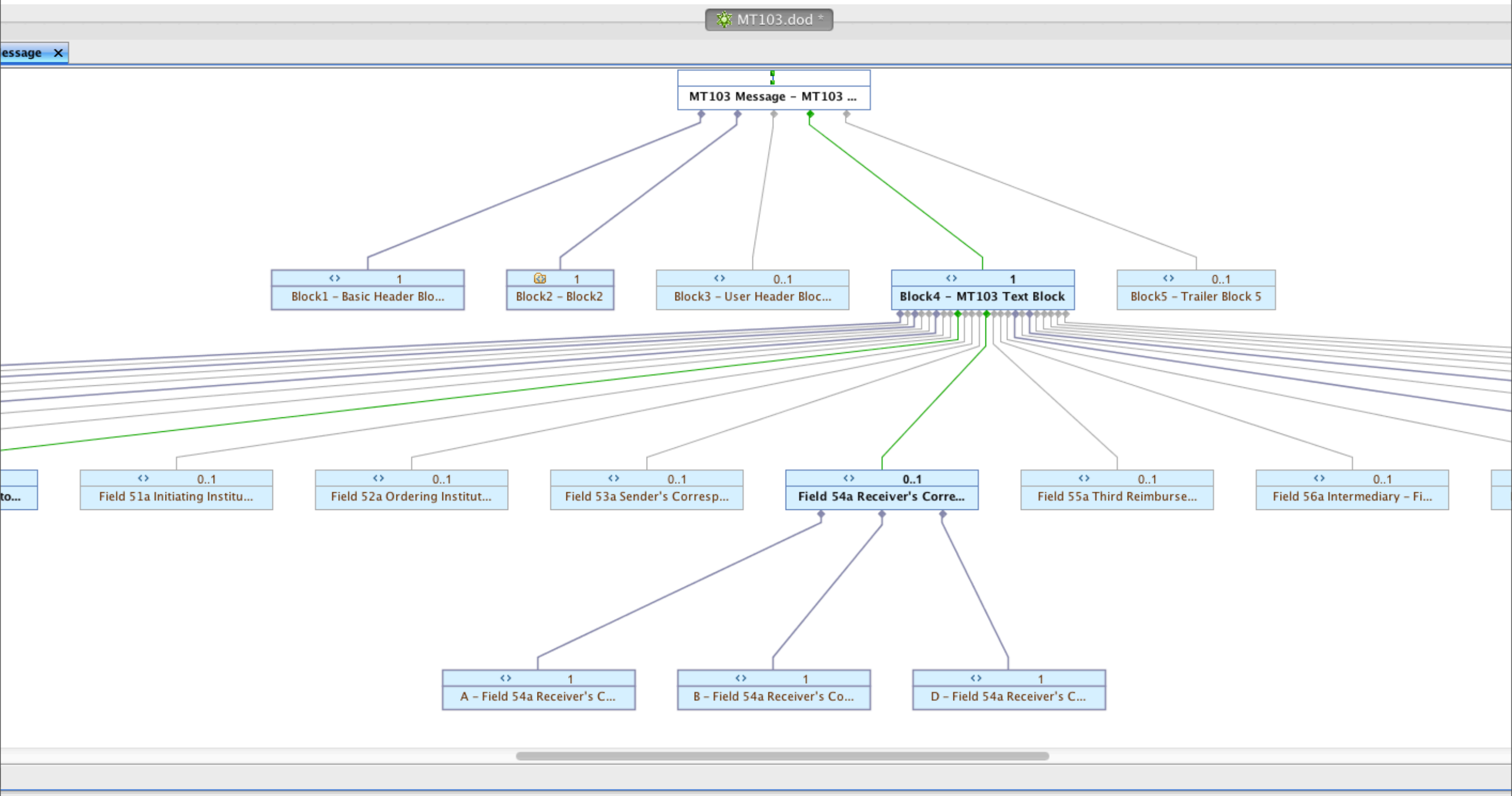


MT103 Structure



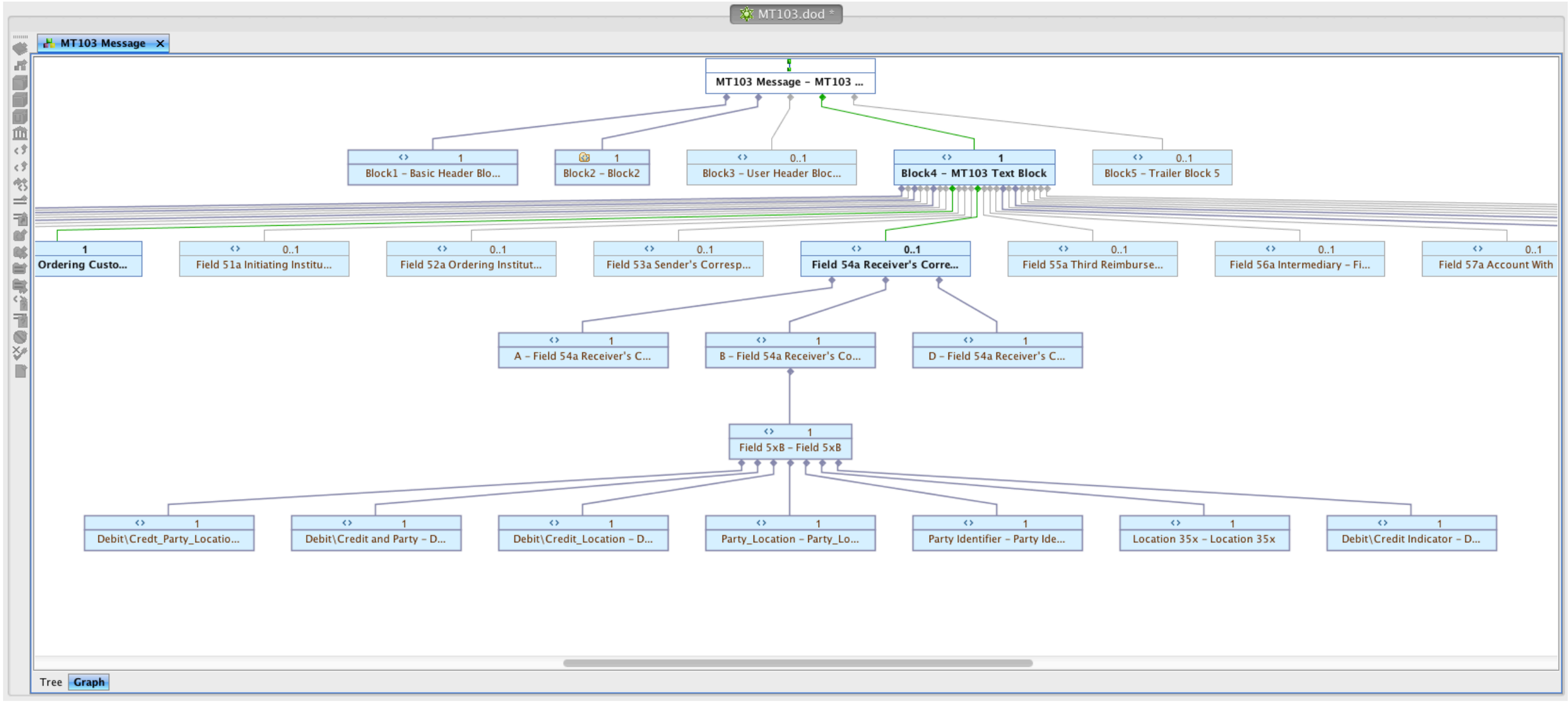
- The MT103 Block 4 has 22 fields
  - Some are optional
  - Some repeat a variable number of times
  - Some are grouped into (repeating) sequences

MT103 Structure



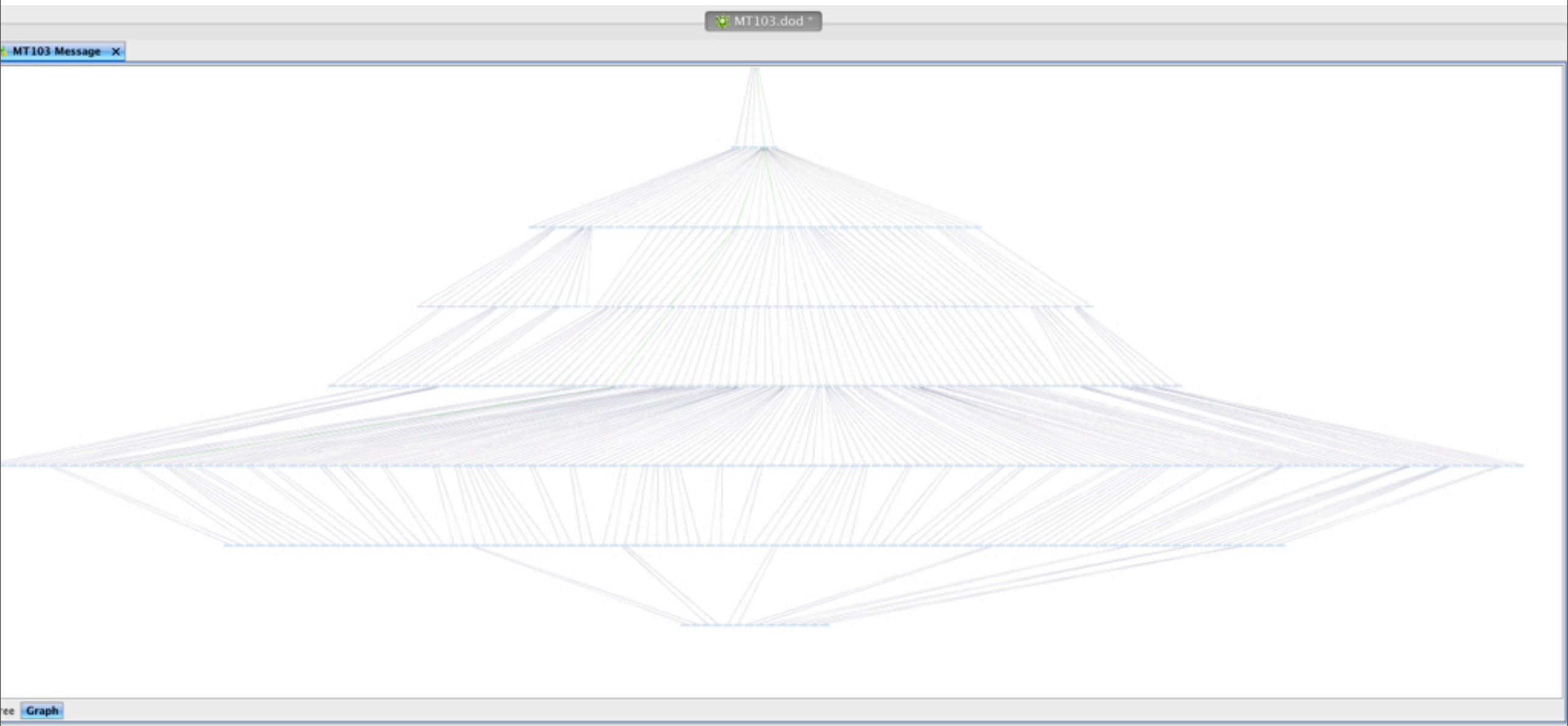
- Some fields have multiple options

MT103 Structure



- Each field can be composed of multiple components

The Full MT103 Definition



To Make Matters Worse...

- There are hundreds of SWIFT messages
  - Each organisation typically uses a subset
- The standard changes every year
  - Firms may need to support multiple versions concurrently
- This is just one of many standards
  - FIX, FpML, SEPA, ...

RDBMS & Financial Messages

- Storing the full message in a relational structure is very time consuming and fragile
  - In practice it's not viable

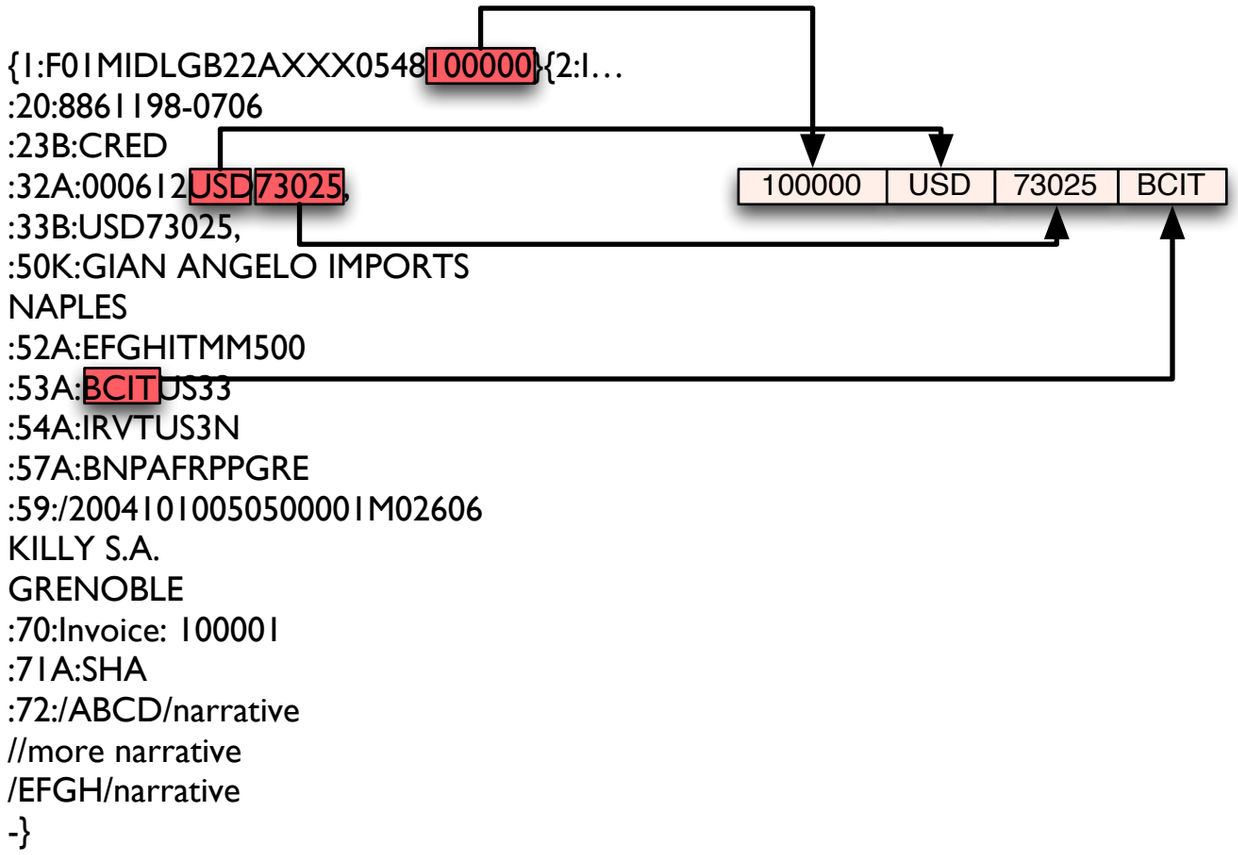
- What options do we have?

Lossy Storage

- If we only care about a subset of fields, we can construct a simpler storage structure

- Information thrown away however is lost forever
  - Advances in processing capacity however mean that new applications are being found for historical data

- Hybrid solution is to store original source message alongside in a CLOB
  - Not readily accessible but at least it's available



XML Overload

- As message formats are hierarchical, we can construct an XML representation of the messages

- Store this in a CLOB

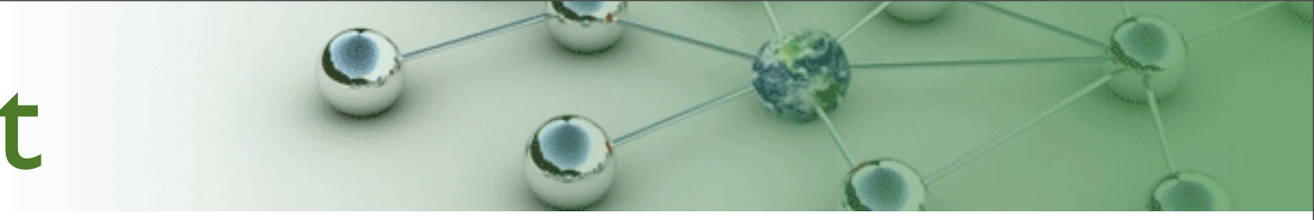
- Either create additional columns for indexed values or use RDBMS with XML index support

- Extraction of data either requires multiple XPath queries or parsing the XML to resurrect the original message/object
  - Slower than querying simple cols and (probably) an artificial format

100000	USD	73025	BCIT

```
<?xml version="1.0"?>
<MT103 xmlns="http://www.c24.biz/IO/SWIFT2012">
   <Block1>
      <ApplicationID>F</ApplicationID>
      <ServiceID>01</ServiceID>
      <LTAddress>
         <BankCode>MIDL</BankCode>
         <CountryCode>GB</CountryCode>
         <LocationCode>
            <LocationCode1>2</LocationCode1>
            <LocationCode2>2</LocationCode2>
         </LocationCode>
         <LogicalTerminalCode>A</
LogicalTerminalCode>
         <BranchCode>XXX</BranchCode>
      </LTAddress>
      <SessionNumber>0548</SessionNumber>
      <SequenceNumber>100005</
SequenceNumber>
   </Block1>
   <Iblock2>

      ....

</MT103>
```

Message Format



- Our in-memory structure needs to model all the data we need for parsing/processing in an efficient structure
  - e.g. a tree of (Java) objects

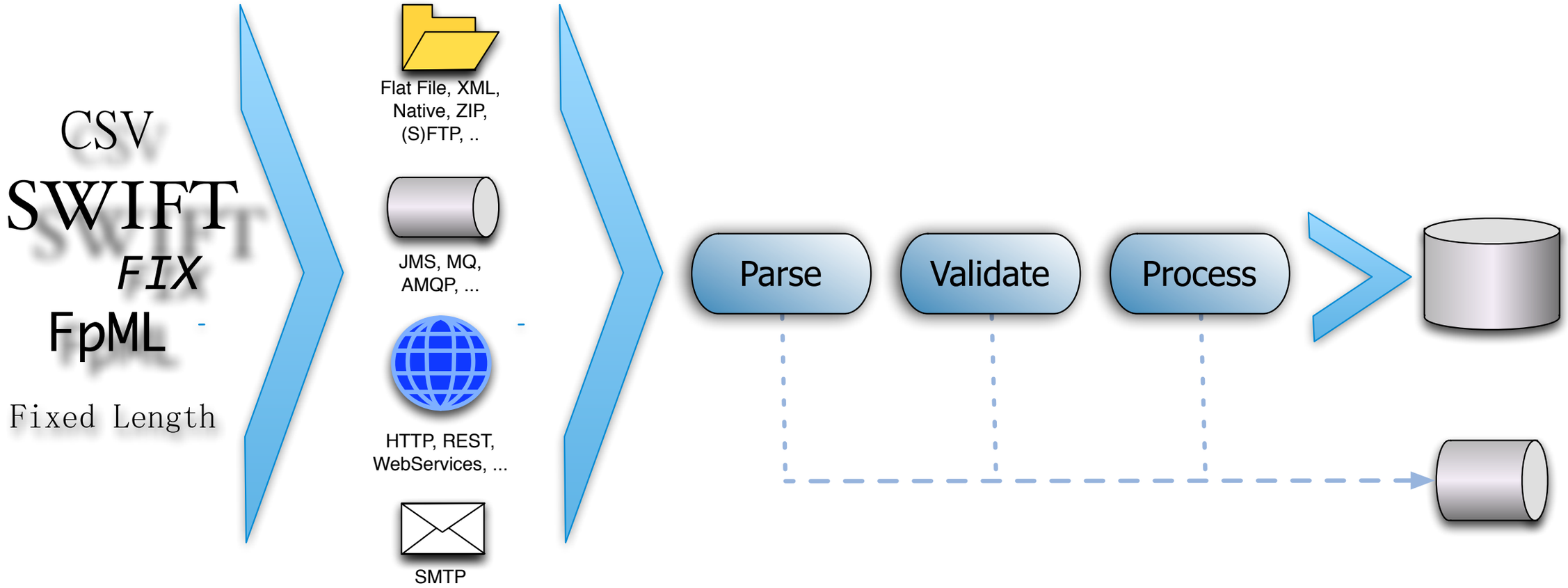
- Ideally we'll persist the same structure directly into the store
  - Persistence and loading are therefore fast
    - Do we even need to persist to disk?
  - We need ready access to the original message format too

- The store must be able to index arbitrary attributes in the persisted object tree for querying

- The store should be capable of extracting individual values from the object tree

Future Proofing

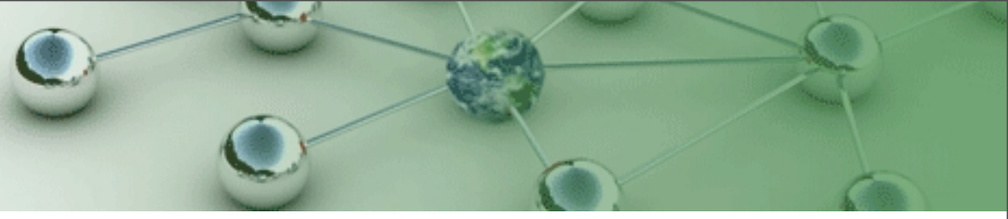
- New clients == new message formats
  - Same purpose, just different presentation
- Systems architected to work regardless of format & transport



C24 Integration Objects

- Pre-built models for common standards
  - Updated as standards evolve
- Easy to import/create new models
- Java binding toolkit
  - Parse messages into queryable objects

Partners & Platforms



Hortonworks

Neo4j

springsource

mongoDB

{name: “mongo”, type:“DB”}

hadoop

MuleSoft™

GIGASPACES

synapse

FuseSource
integration everywhere

redhat.

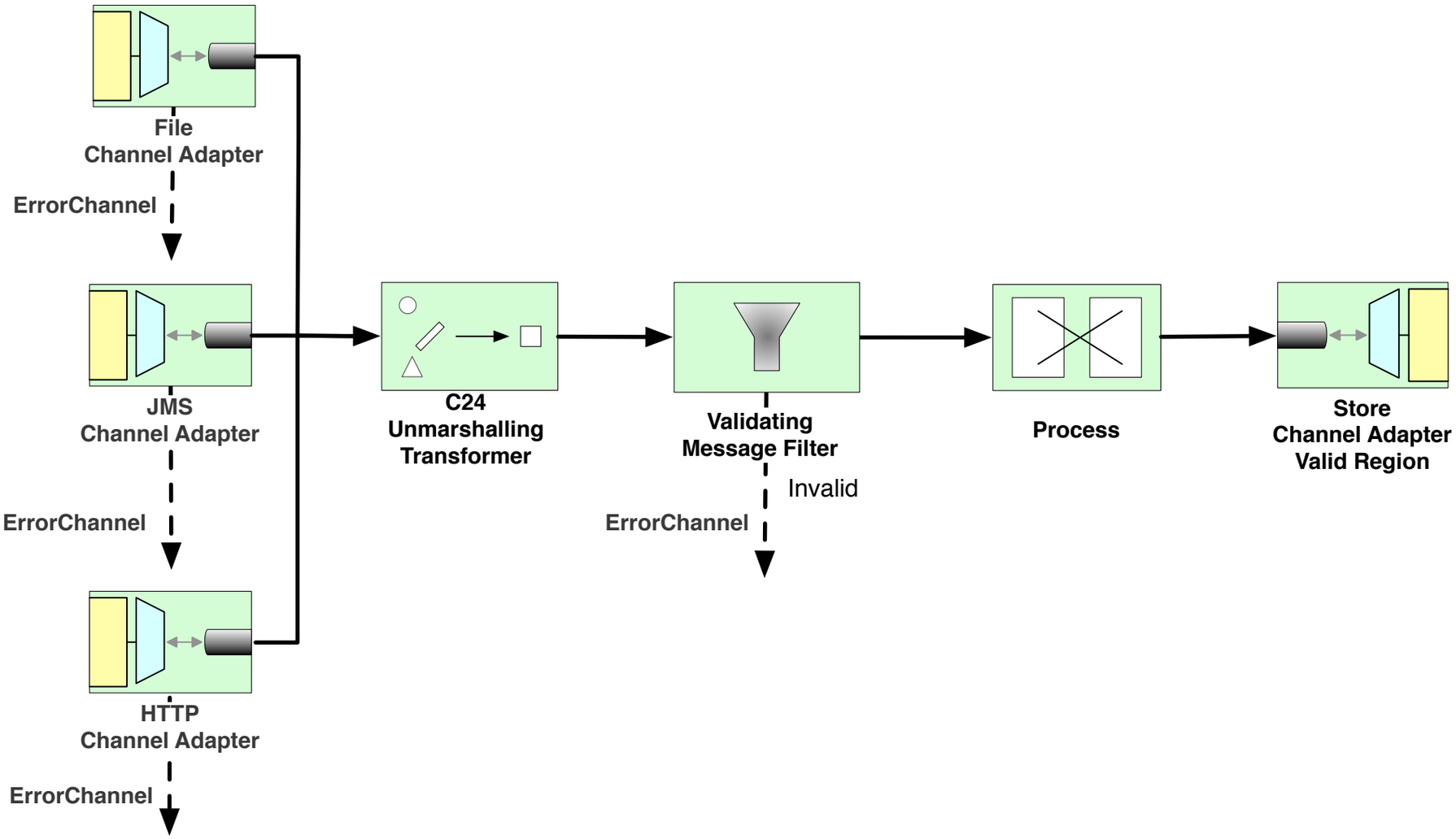
JBoss

Apache Camel™

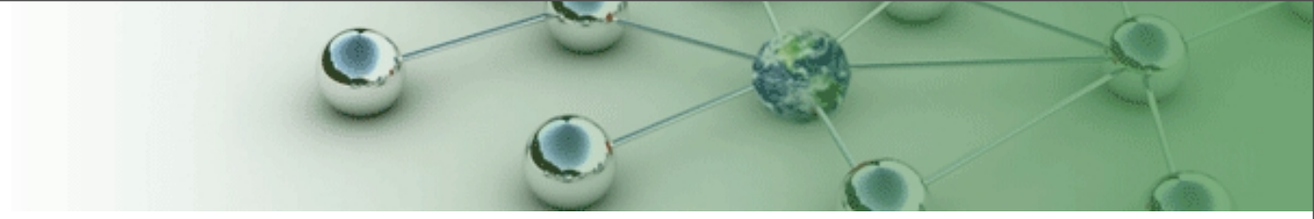
vmware

Typical Integration Flow

- Linear process, potentially replicated



Acquisition

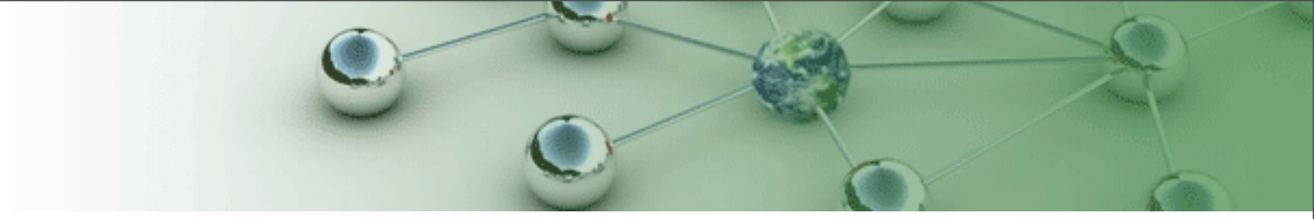


```
<!-- Read the SWIFT message files -->
<int-file:inbound-channel-adapter directory="data/swift"
                                  filename-pattern="*.dat"
                                  prevent-duplicates="true"
                                  channel="parse-swift-message-channel"
                                  id="inbound-transport-swift-file">

        <poller fixed-rate="100" task-executor="swiftThreadPool"/>
</int-file:inbound-channel-adapter>

<task:executor id="swiftThreadPool" pool-size="8"/>
<channel id="parse-swift-message-channel"/>
```

Flow-based Processing

```
<!-- SWIFT Parsing -->
<c24:model id="swiftMessageModel" base-element="biz.c24.io.swift2012.MT103Element"/>
<int-c24:unmarshalling-transformer input-channel="parse-swift-message-channel"
                                   output-channel="validate-message-channel"
                                   model-ref="swiftMessageModel"
                                   source-factory-ref="textualSourceFactory"/>


<!-- Validate and filter out invalid messages -->
<int-c24:validating-selector id="c24Validator" throw-exception-on-rejection="true"/>
<filter input-channel="validate-message-channel"
        output-channel="process-message-channel" ref="c24Validator"
        discard-channel="errorChannel"/>


<!-- Process - stub out for now-->
<bridge input-channel="process-message-channel"
        output-channel="persist-valid-message-channel"/>


<!-- Persist -->
<channel id="persist-valid-message-channel"/>
```

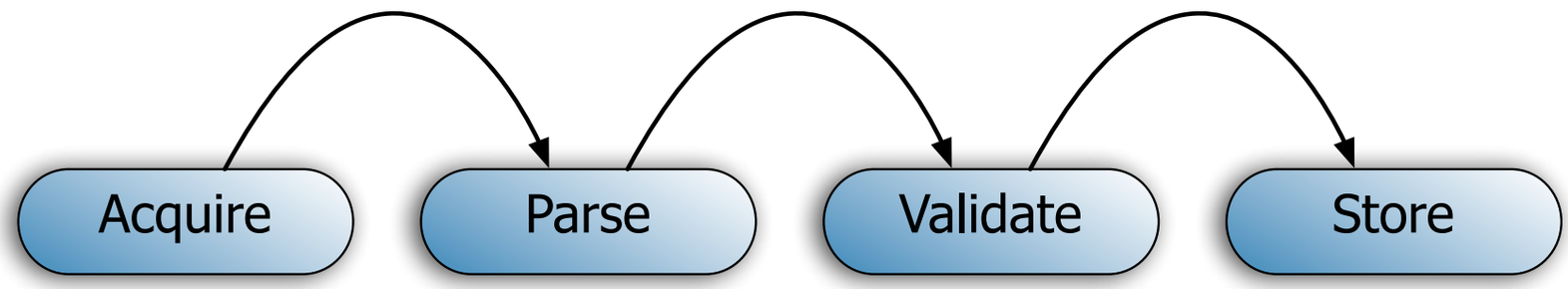
Persistence



```
<int-gfe:outbound-channel-adapter id="validStore"
                                  channel="persist-valid-message-channel"
                                  region="valid">

    <int-gfe:cache-entries>

        <beans:entry key="headers[id]" value="payload"/>

    </int-gfe:cache-entries>

</int-gfe:outbound-channel-adapter>
```

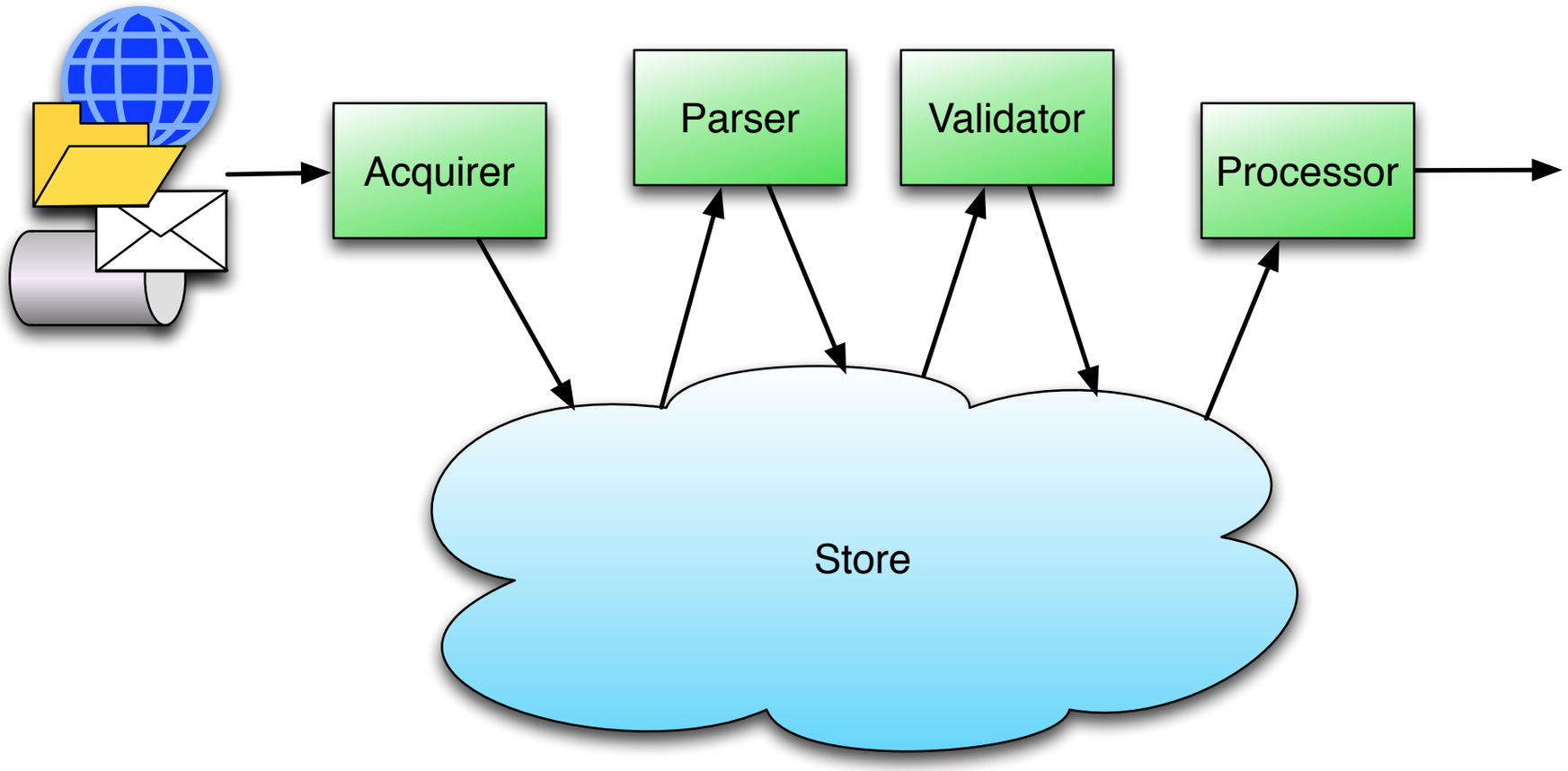
Flow-based Architecture

- All of the scalability is at the front-end
  - Requires us to spread input load evenly across processing units



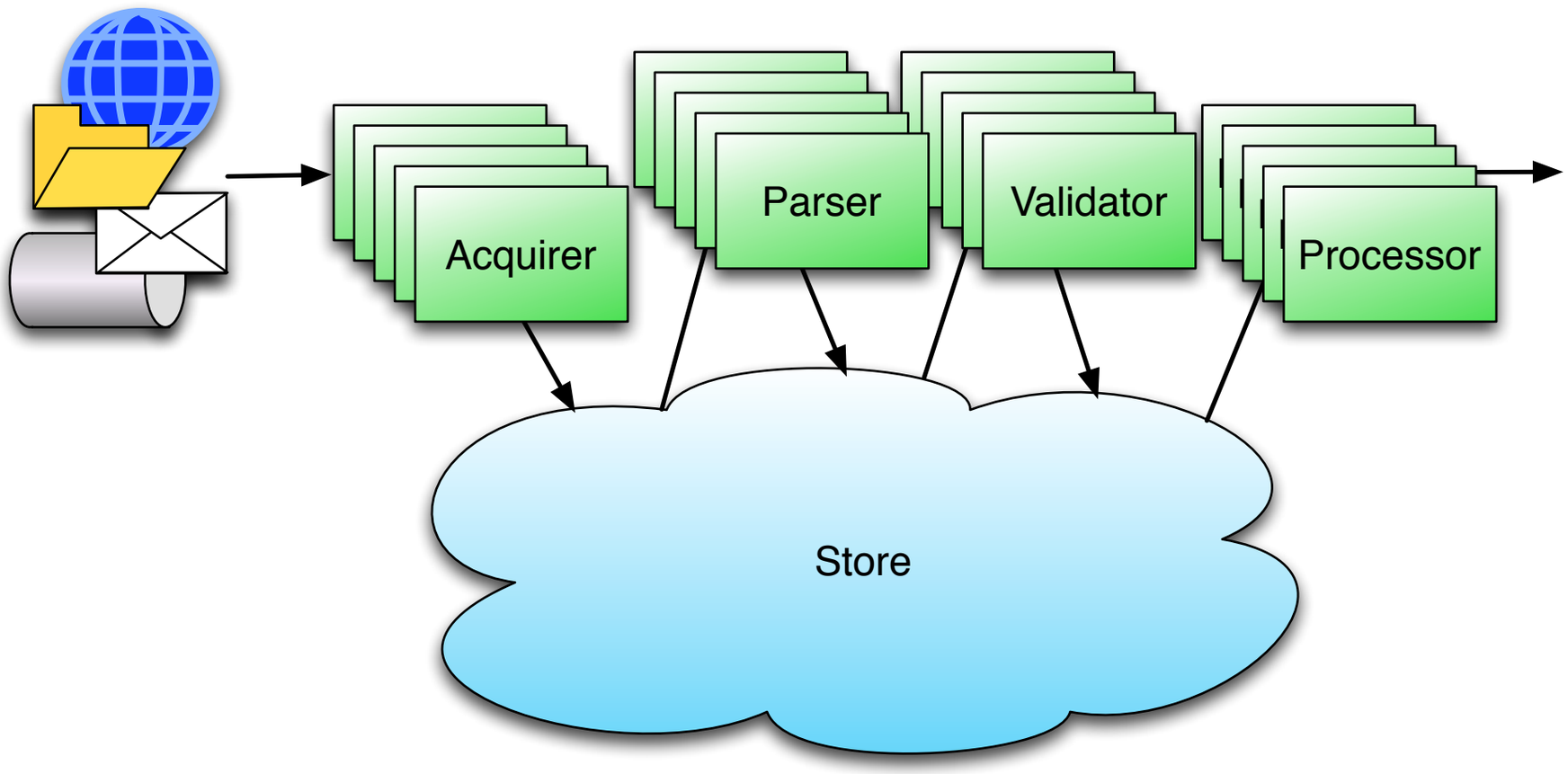
- Alternatively we can use the store and make our flow event-based

Event-based Architecture



- The store ‘triggers’ processing
  - Listeners
  - Subscriptions
  - Querying
  - ...

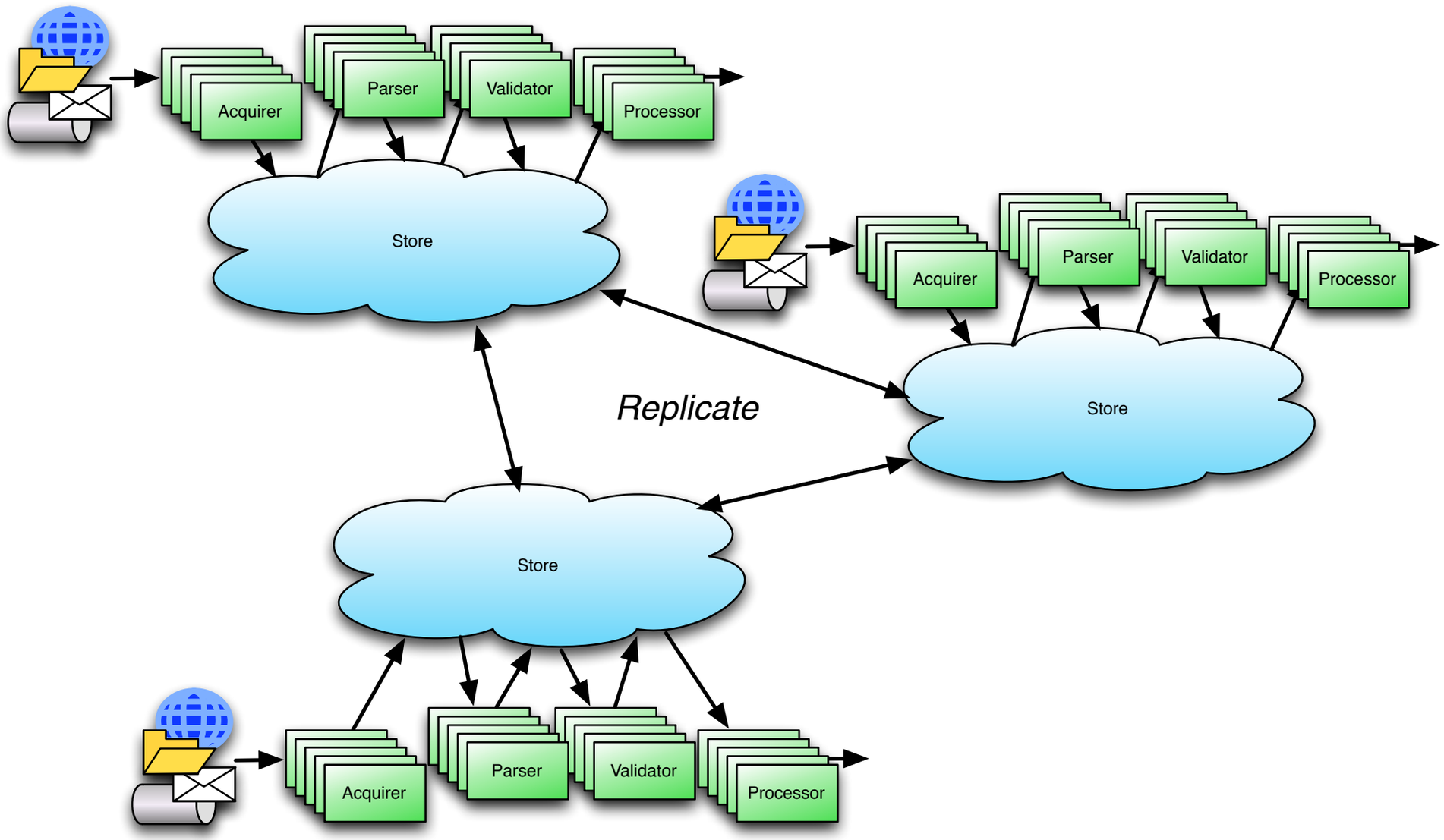
Scaling Up

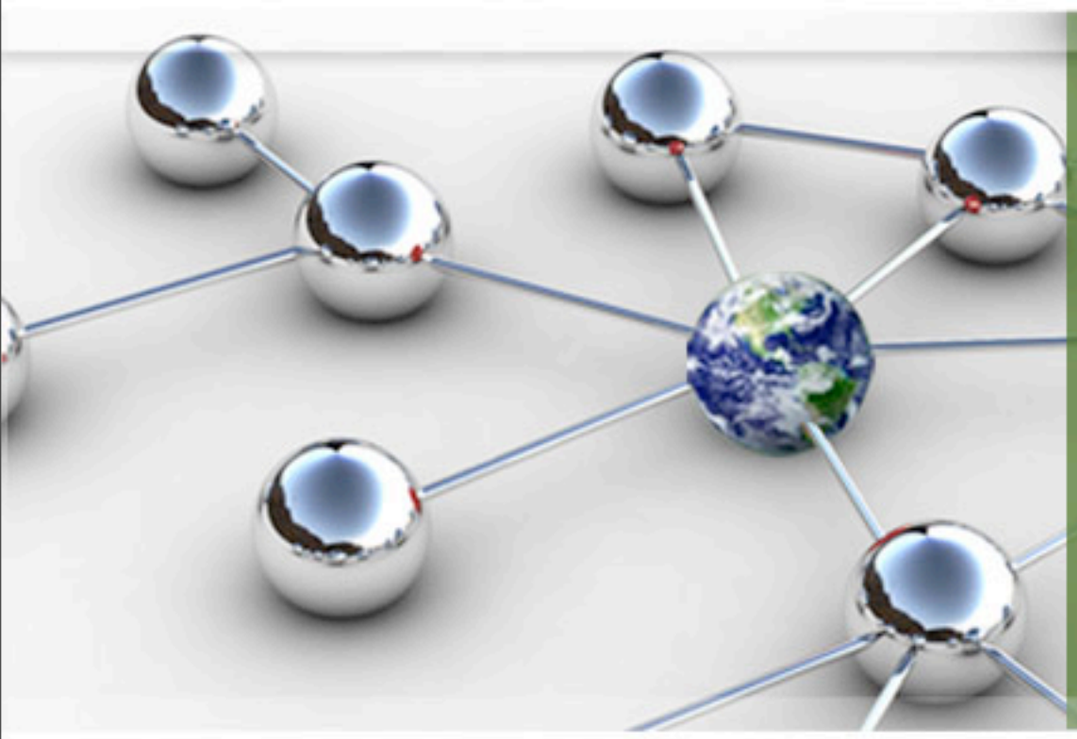


- Multiple components can register an interest in the events
  - We can scale out and add redundancy by replicating the listeners

Scaling Out

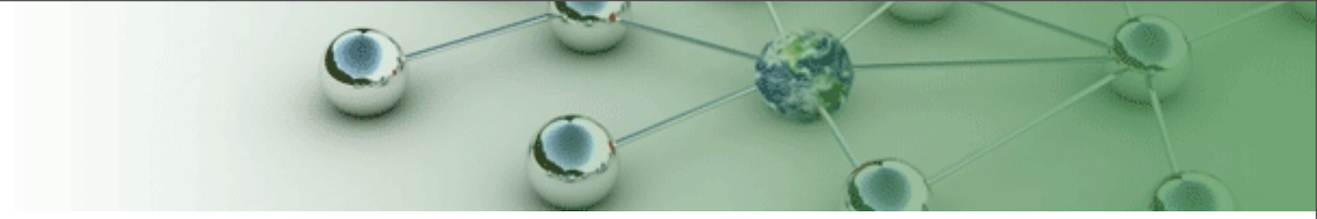
- We can control what is replicated and to where
- The node that parses the message doesn't have to be the same as the one that processes it





Ingesting Data

Demo Time



- GemFire Groovy DSL is still in development

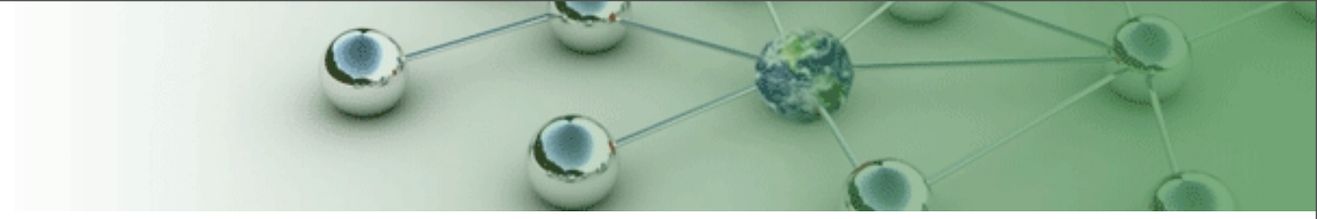
- Allows simple definition of regions and event-based handlers

```
region 'new', shortcut: REPLICATE, {
    listener (
        afterCreate: { EntryEvent e ->
            try {
                // Parse the raw message
                String message = e.getNewValue();
                source.setReader(new StringReader(message));
                parsedRegion.put(e.getKey(), source.readObject(element));
            } catch(Exception ex) {
                exceptionRegion.put(e.getNewValue(), ex);
            } finally {
                newRegion.remove(e.getKey());
            }
        }
    )
}
```

Demo Takeaways

- We had the full message available to us
  - With a meaningful schema
- We can query on and extract the full message or individual values
- Approach is message format neutral
  - We just need to know the hierarchical location of the information we're interested in
- Our persistence code is extremely simple
- We can index on any [derived] attribute

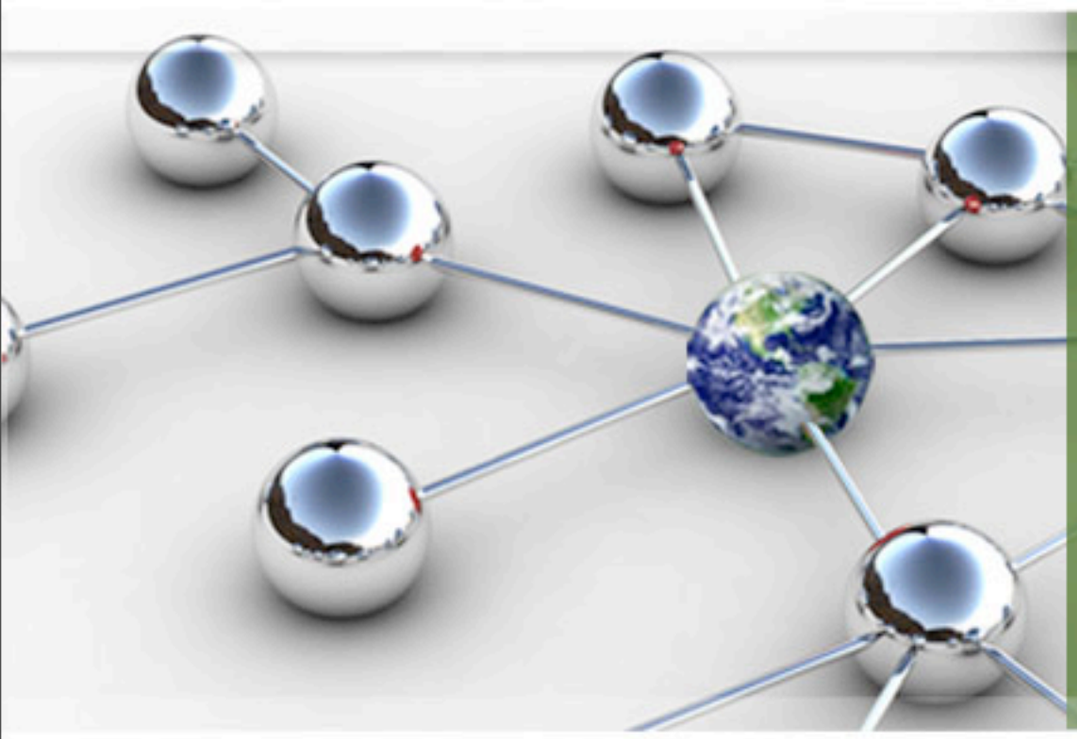
Scaling Out



- Nodes can be replicated and sharded/partitioned
  - Queries are automatically distributed

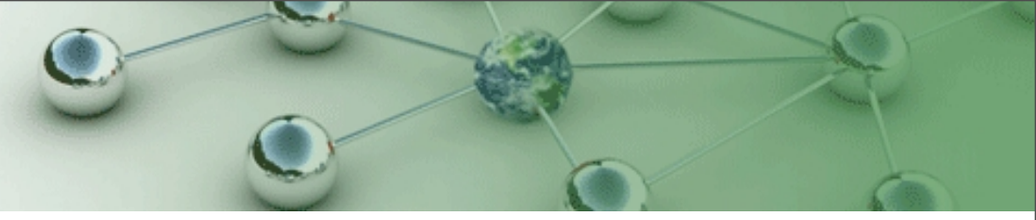
- Data can be ingested on any node

- The same event-based architecture works well for distributed processing & analysis
  - And as we've seen our data grids allow us to embed our logic in them



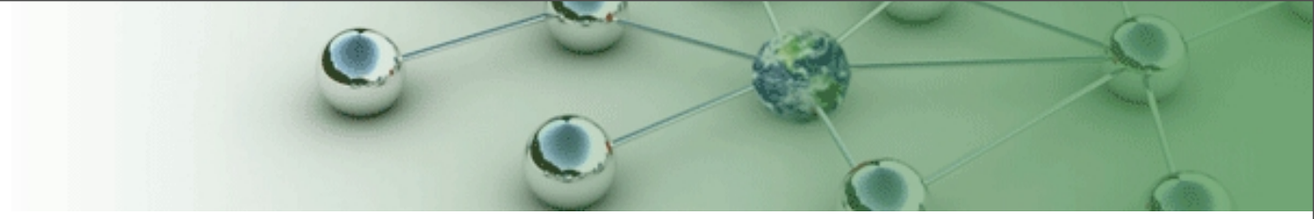
Processing

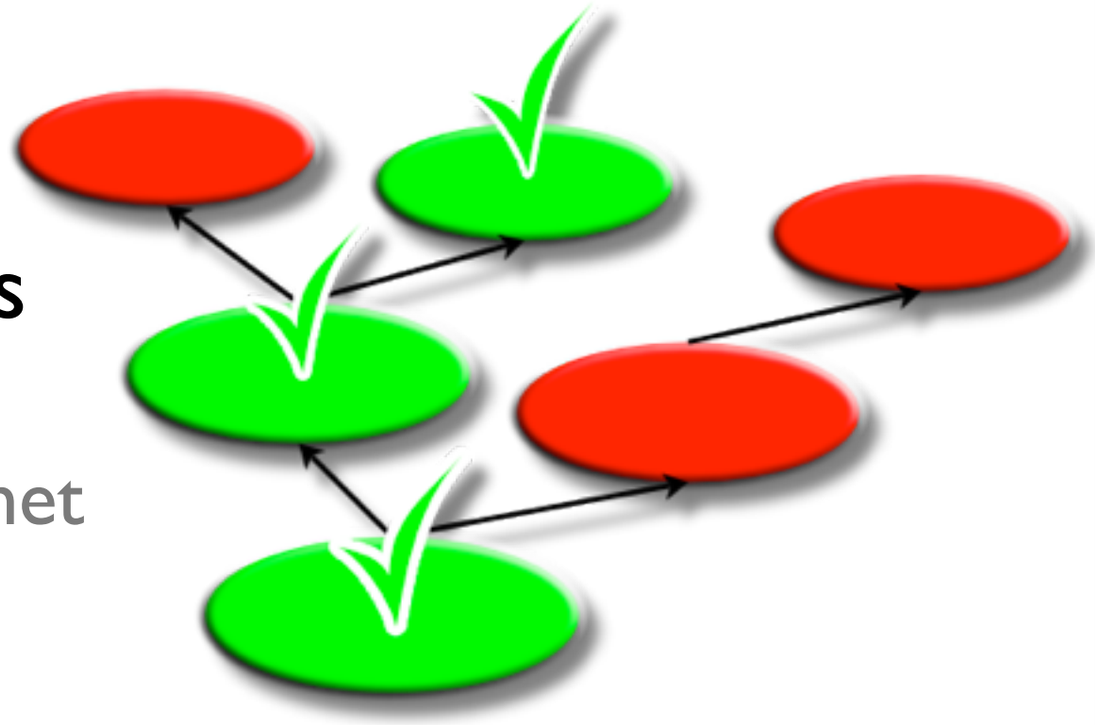
Real-Time Analytics



- Complex Event Processing
- Inter-dependent set of variables & rules
- Typically very high volume & latency critical
- Run on multiple nodes to cope with
  - Load
  - Volume
  - Resilience

Rule Engines





- Track dependencies between rules
  - Recalculate when necessary
  - Trigger behaviour when all rules are met

- Changes typically triggered by time & message state change
  - Fits well with our event-based architecture

- See RETE algorithms & derivates for creating highly performant rule trees

A Simple Architecture

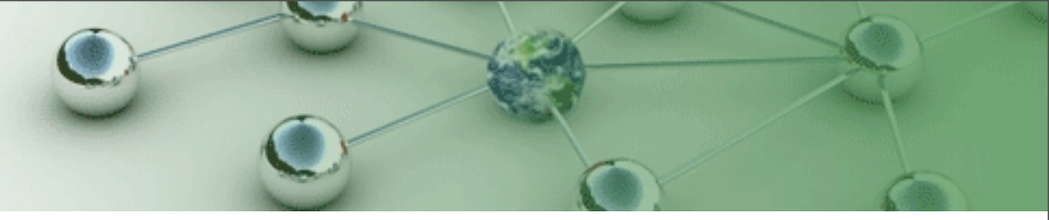
- We'll use Gigaspaces XAP for some variety
  - Distributed in-memory data grid + elastic application platform

- Sample application consists of 3 key areas:
  - 'Space Object' ie what are we going to be storing
  - Feeder - how do we get data into the grid
  - Processors - what happens to data once it's in the grid

```
mvn os:create -DgroupId=my.group -DartifactId=Test1 -Dtemplate=basic
mvn package
mvn os:deploy -Dlocators=localhost:4174
```

- Message Processors
  - ParseProcessor - parse raw messages into domain objects
  - RuleProcessor - trigger rule updates on message state change

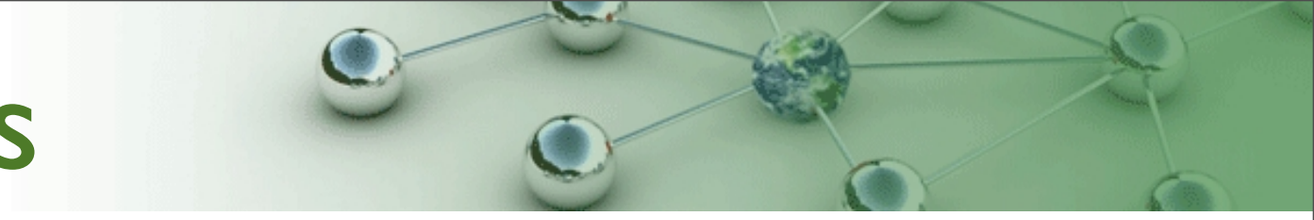
Configuring XAP



- Multiple ways to mark up our code
  - We'll use annotations for simplicity

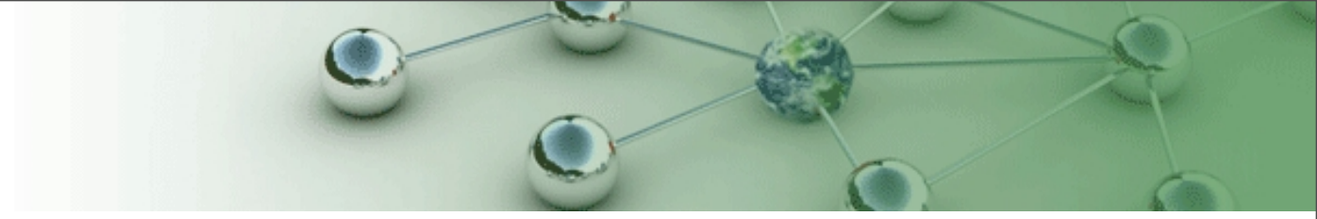
- First we need the type we're going to store in the space

The Space Class



```
@SpaceClass
public class Message<T extends ComplexDataObject> implements Serializable {

    public static enum State {
        NEW,
        VALID,
        PROCESSED,
        INVALID
    };

     private String id = null;

    private State state = null;

    private String rawMessage = null;

    private T message = null;

    private Throwable failure = null;

    @SpaceId(autoGenerate=true)
    public String getId() {
        return id;
    }
```

The Feeder



- Simple implementation to read raw messages from the filesystem
  - In practice we'll be acquiring from multiple sources
  - Use an outbound channel adapater as a feeder?

```
public class Feeder implements InitializingBean, DisposableBean {
    @GigaSpaceContext
    private GigaSpace gigaSpace;

    ...

    public void loadFiles() {
       ...;
       for(File file : dir.listFiles()) {
           Scanner scanner = new Scanner(file);
            String fileContent = scanner.useDelimiter("\\A").next();
            scanner.close();
            gigaSpace.write(new Message(fileContent));
       }
    }
```

ParseProcessor

- Declare the Processor and tell it when to trigger

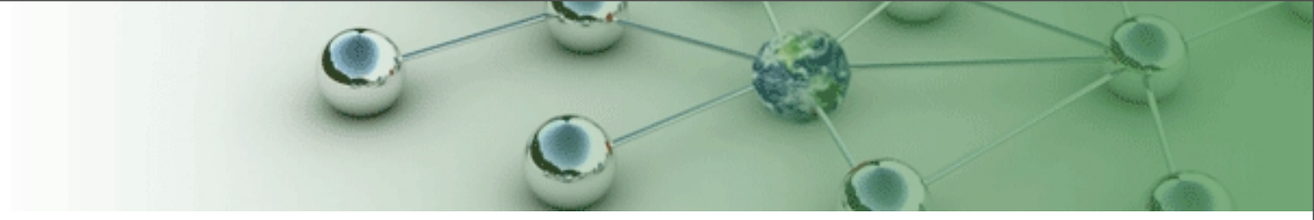
```
@EventDriven @Notify
@NotifyType(write = true, update = true)
public class ParseProcessor<T extends ComplexDataObject> {

    /**
     * We process any messages in a new state
     * @return
     */
    @EventTemplate
    Message templateMessage() {
       Message msg = new Message();
       msg.setState(Message.State.NEW);
       return msg;
    }
```

ParseProcessor

- ...and how to process the messages it receives

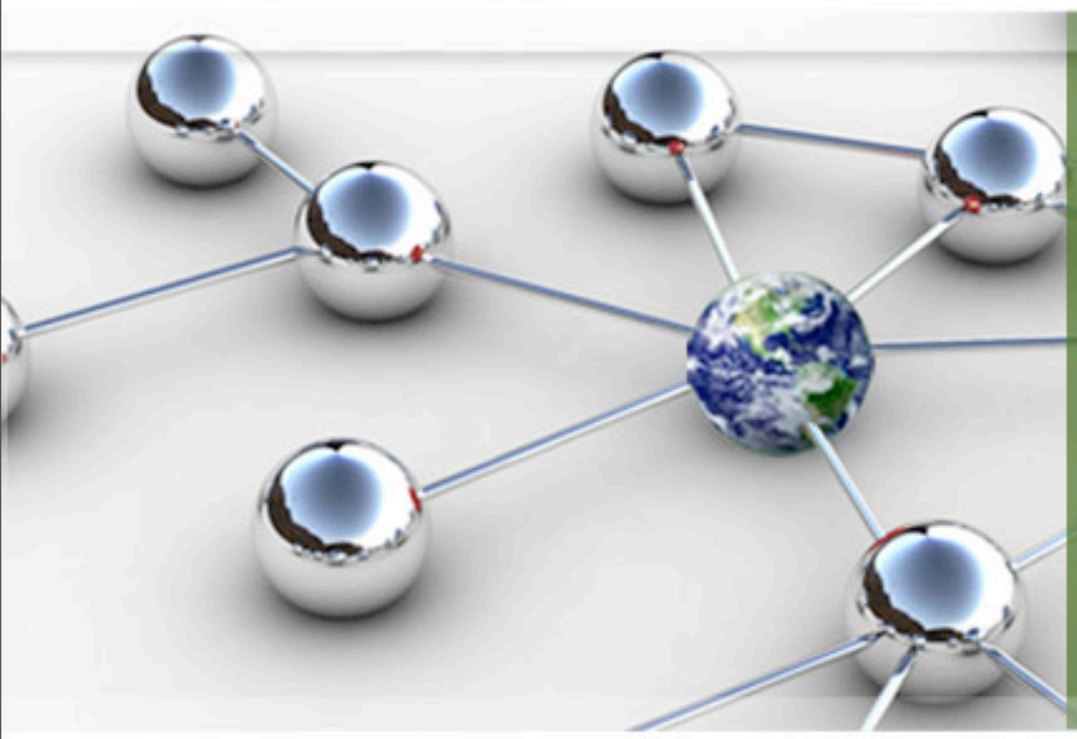
```
@SpaceDataEvent
public Message<T> processData(Message<T> data) {

   try {

      log.info("Parsing " + data.getId());

      source.setReader(new StringReader(data.getRawMessage()));
      data.setMessage((T)source.readObject(element));
      data.setState(State.VALID);

      log.info("Parsed " + data.getId());

   } catch(Throwable ex) {
      log.info("Message invalid " + data.getId());
      data.setFailure(ex);
      data.setState(State.INVALID);
   }

    return data;
}
```

RuleProcessor



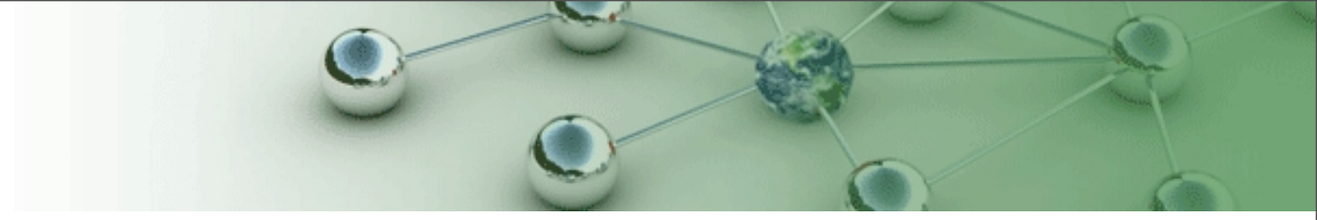
```
/**
 * We process any messages in a valid state
 * @return
 */
@EventTemplate
Message templateMessage() {
   Message msg = new Message();
   msg.setState(Message.State.VALID);
   return msg;
}


@SpaceDataEvent
public Message<T> processData(Message<T> data) {

   log.info("Processing " + data.getId());
   engine.process(data.getMessage());
   data.setState(State.PROCESSED);

   return data;
}
```

Demo



General Concerns

Serialisation



- At some point your objects are likely to be serialized
  - ...and it won't be during your local testing

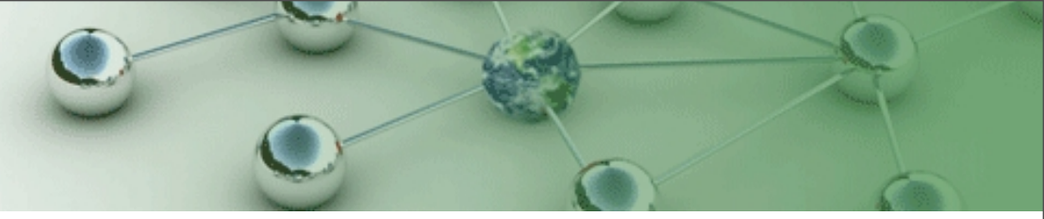
- Create unit tests

```
MyType obj = ...;
ByteArrayOutputStream bos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(bos);
oos.writeObject(obj);
oos.close();
```

- Check what you're actually serialising

```
assertThat(bos.toByteArray().length, is(lessThan(some size)));
```

Interface Semantics



- Exclusive lock vs shared lock vs take
- Who gets notified?
- Callback thread behaviour
  - What happens if we hold on to the thread for a long time?
  - What if we generate an exception?
- None of these technologies make coordination problems disappear
  - They can however make it easier to apply good practice

Thank You!

andrew.elmore@c24.biz

@andrew_elmore