ORACLE

C24
www.C24.biz

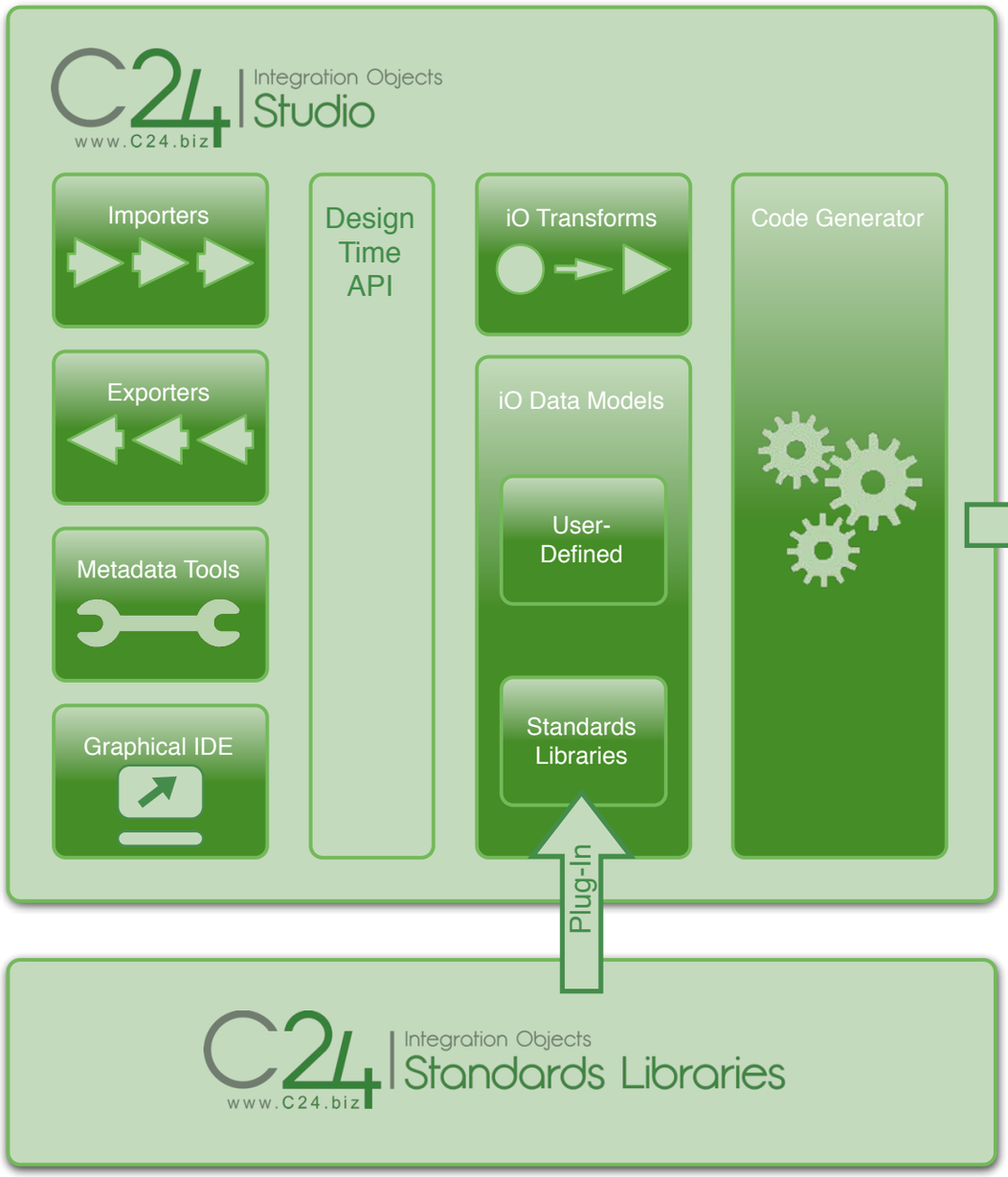
Getting more out of (and specifically IN TO) Oracle Coherence

John Davies | CTO
Steve Miller | Product Director

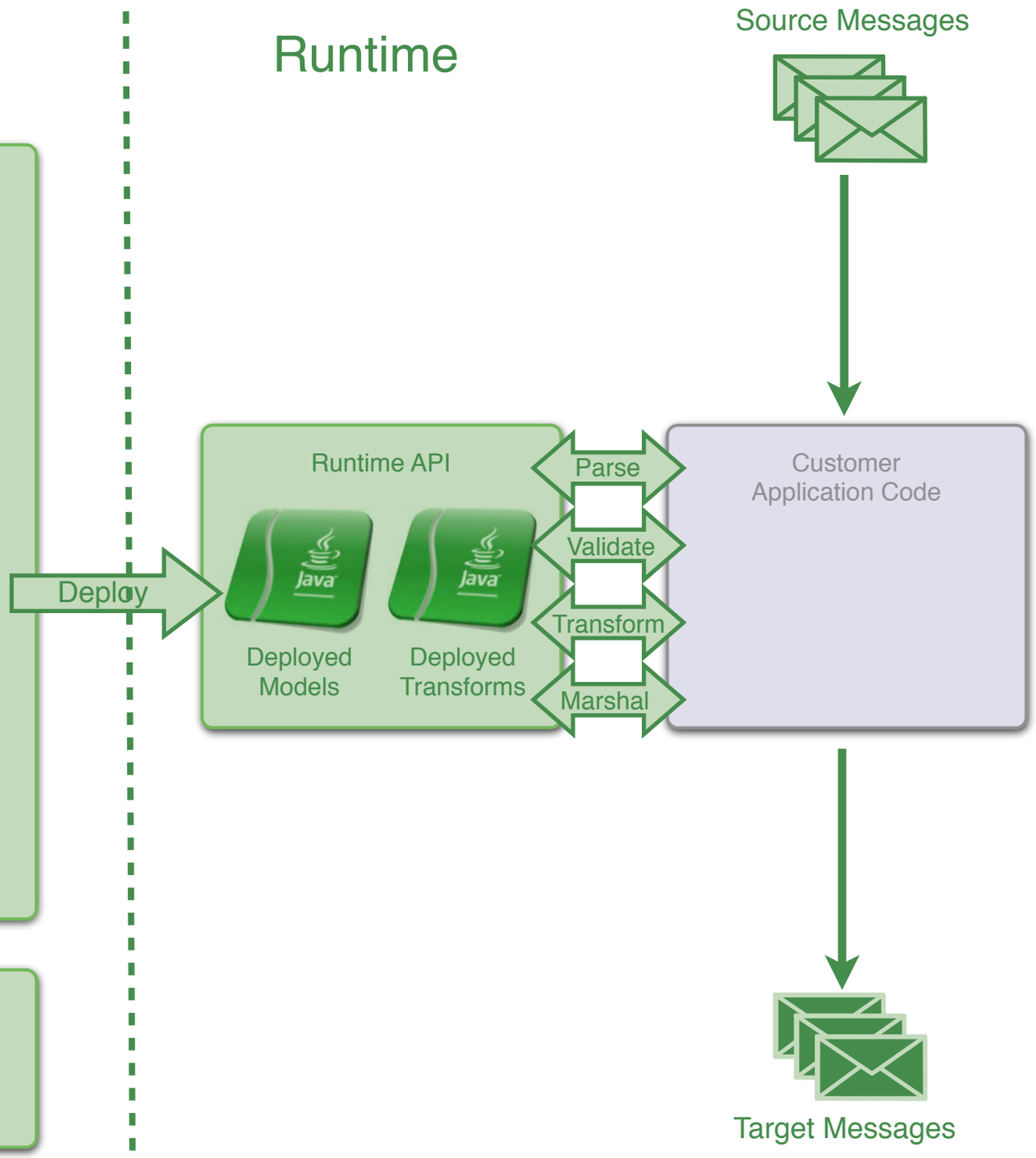
Coherence SIG
Oracle London HQ | 17th July 2014

C24 Integration Objects

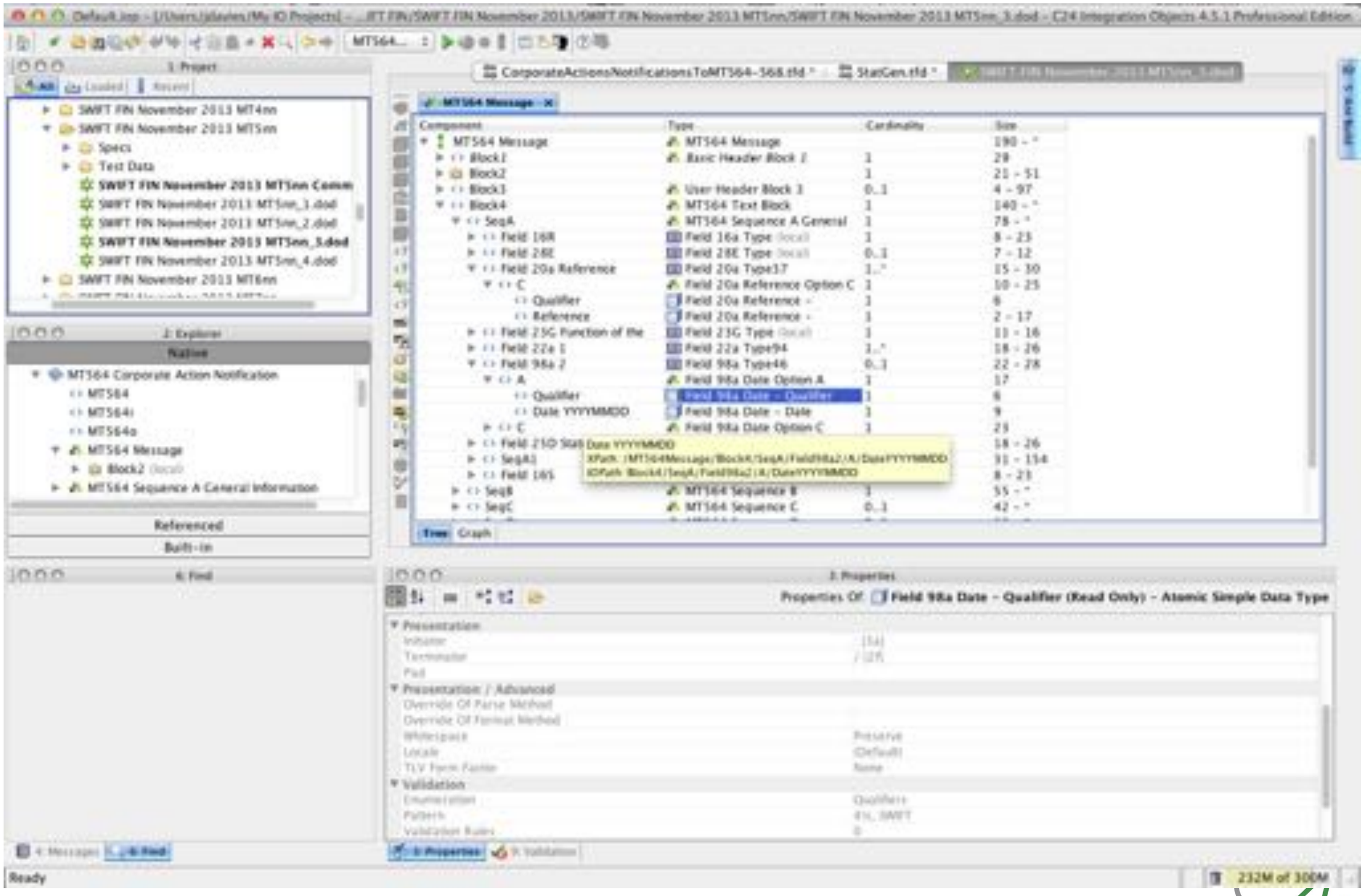
Design Time



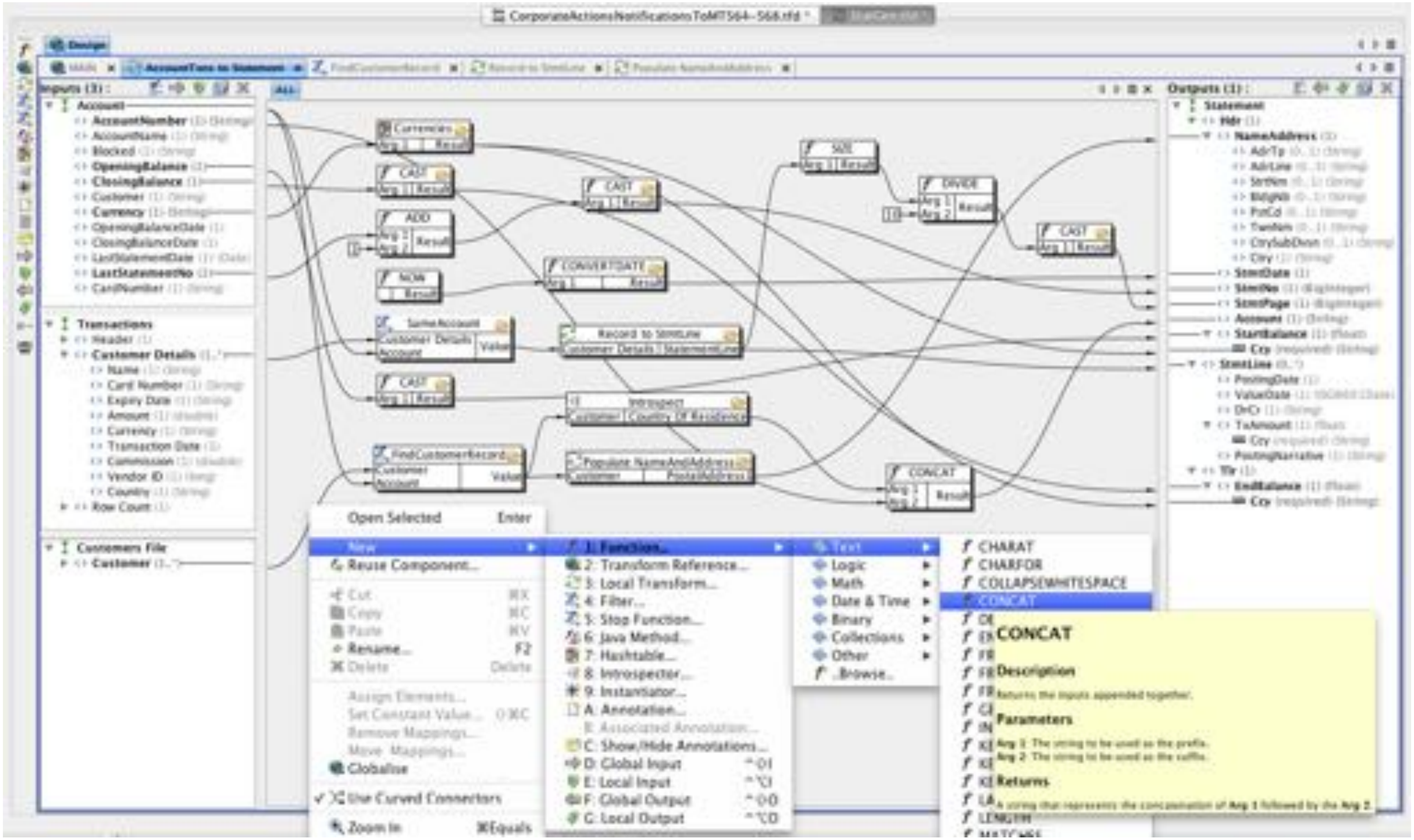
Runtime



SWIFT Example (MT564)



Transformation Example

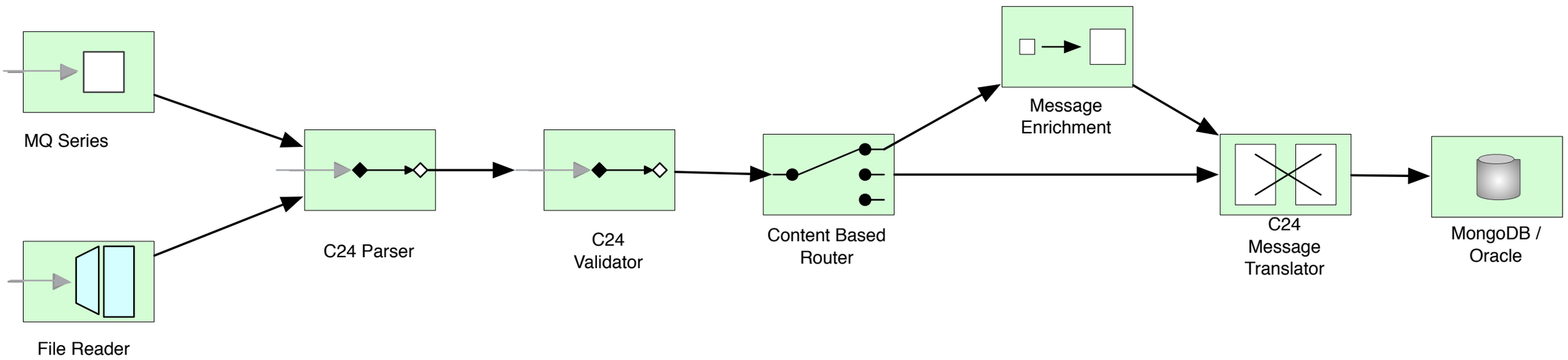


Where we fit

- Where ever you're dealing with events or messages, C24 can help
- The more complex or scale-critical the better the advantage
  - Proprietary formats, industry standards, legacy interfaces

EAI, SOA & ESBs

- C24-iO has deep integration with Spring, Mule, Fuse & Camel etc.
- We generate the Spring config for you so you can use Spring Integration right out of the box
- New performance changes to Spring due out later this summer were driven by C24
- Spring 4.1 will be able to pre-compile the SpEL expressions

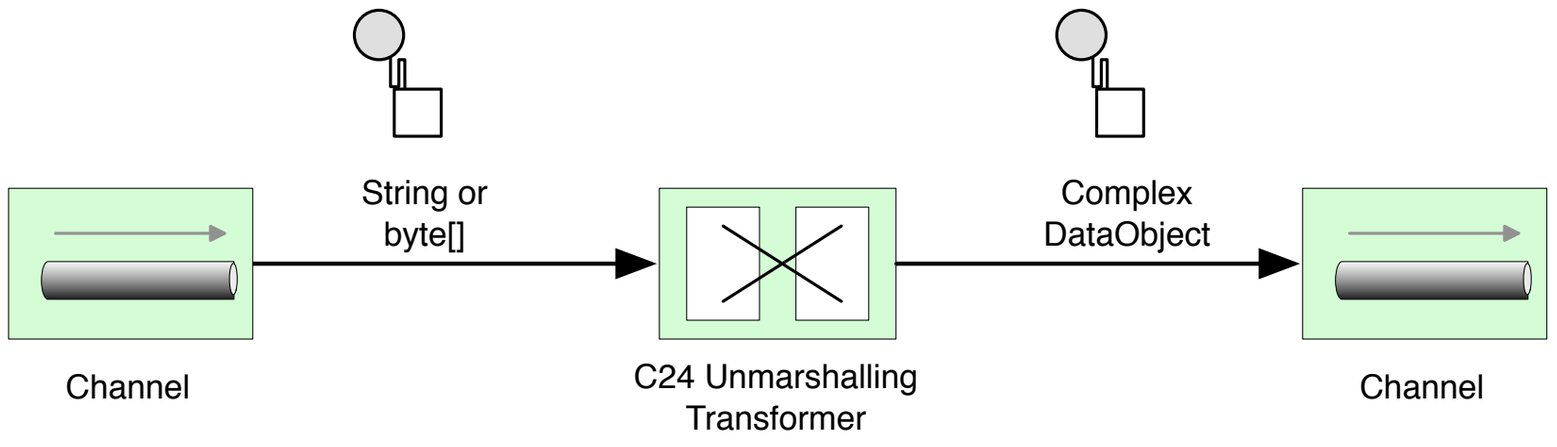


Spring out of the box

- Deploying Fix for example will deploy the config for the parser and model...

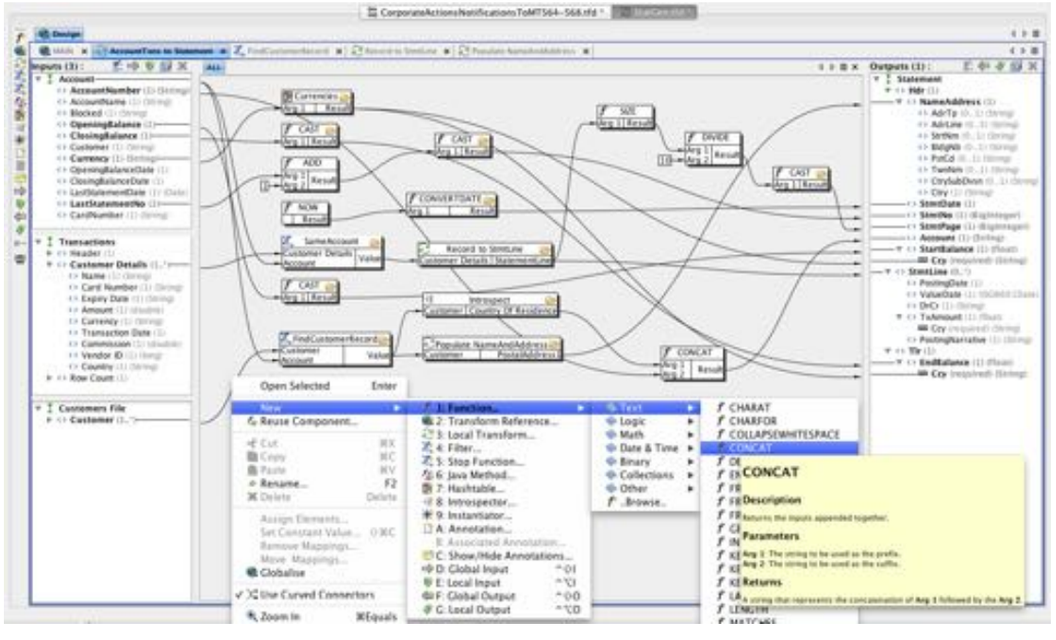
```
<int-c24:unmarshalling-transformer
    model-ref="fixModel"
    source-factory-ref="sourceFactory"
    input-channel="…" output-channel="…" />

<bean id="sourceFactory"
    class="biz.c24...source.FixSourceFactory">
  <property name="encoding" value="UTF-8" />
</bean>
```

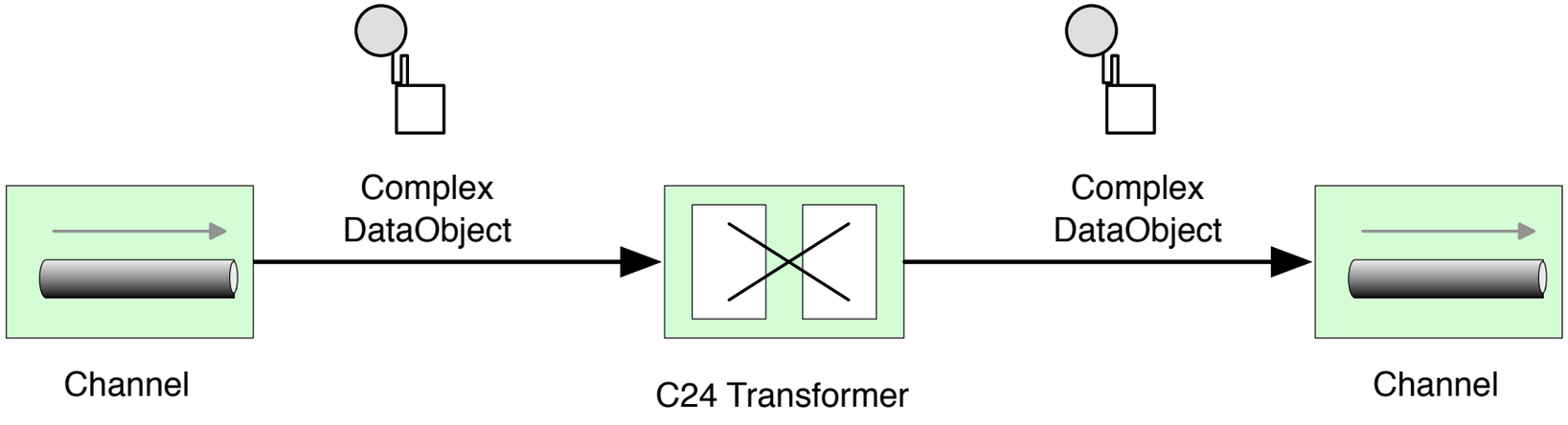


Transformation in Spring

- Real X-to-Y transforms are modelled in the Studio
- The generated transform slots into SI by just specifying the generated class



```
<int-c24:transformer
    transform-class="biz.c24...basic.ExampleTransform"
    input-channel="…" output-channel="…" />
```



Java 8 & Scala

- Change the deploy option to Java 8 and…

```
List<Atom> elements = C24
          .parse(biz.c24.periodic.PeriodicDocumentRoot.class)
          .from(new File("resources/periodic.xml"));

elements.sort((a1, a2) -> Integer.compare(a1.getAtomicNumber(), a2.getAtomicNumber()));

// List all the elements that are solid at room temperature (20C) and boil below 600 deg C
System.out.println("\nSolid at 20°C but boil lower than 800°C...");
elements.stream()
      .filter( a -> a.getMeltingPoint() != null && a.getBoilingPoint() != null )
      .filter(a -> a.getMeltingPoint().getValue() > 293 & a.getBoilingPoint().getValue() < 1073)
      .map(Atom::getNameElement)
      .forEach(System.out::println);
```

- And it works nicely in Scala too…

```
var parser = C24.parse(classOf[CustomersFile]) as C24.Format.XML
var transform = new GenerateContactListTransform
var writer = C24.write() as C24.Format.JSON

new File("/Customers.xml") -> parser -> transform -> writer -> System.out
```

Virtual Getters and Headers

- The XML and therefore generated Java API is usually technically formatted, i.e. it's not easy to extract key business data
  - tradeDate, buySideCurrency, settlementData are all hidden in the message
- For this reason most architectures use a message wrapper and extract the key fields into a header (header enrichment)
  - Even canonicalised messages present the same issue, key business fields are difficult to find
- Using C24 virtual methods…
  - No need for extra message wrappers, no extra memory used
  - Vastly simplifies user-code & maintenance
  - Can be used with ESB/SOA & messaging for filtering & routing etc.
  - Can be used with in-memory cache and database queries/QL etc.
  - Extremely powerful with Spring Integration/Mule, Coherence etc.

Virtual Getters

- We can now generate virtual getters, i.e. getters for fields that don't necessarily represent a real field in the model

```
tradeConfirmed.getTrade().getTradeHeader().getTradeDate().getValue().toDate();
```

- We can now use the much simpler…

```
Date tradeDate = tradeConfirmed.getTradeDate();
```

- Instead of this…

```
MT541SequenceE3Amounts[] seqE3 = mt541Message.getBlock4().getSeqE().getSeqE3();
for (MT541SequenceE3Amounts e3Amount : seqE3) {
  for (Field19aType31 field19 : e3Amount.getField19aAmount()) {
    if (field19.getA().getQualifier().compareTo("SETT") == 0) {
      CurrencyAmount currencyAmount = field19.getA().getSignedCurrencyAmount().getCurrencyAmount();
    }
  }
}
```

- We can use the much simpler…

```
CurrencyAmount getSettlementCurrencyAmount();
```

Lambdas in FS

- Using virtual methods with lambdas we can further simplify the code

```
BigDecimal sum = transactions.stream()
    .filter( t -> t.getBuySideCity().equals("London") )
    .filter( t -> t.getBuySideCurrency().equals("GBP") )
    .map( t -> t.getBuySideAmount() )
    .sum();
```

- This now works across every version of message format and even different message formats
  - FpML
  - FIX
  - ISO 20022
  - Internal canonical format
  - CSV

Compaction

- Everything too large? Why not compress it?
  - It's slow to compress - takes up CPU cycles
  - It's slow to decompress - takes up more CPU cycles
  - The compressed data is relatively useless until it's decompressed
  - Compressing batches is more efficient but you then have to decompress the entire batch too - More CPU cycles again

- Compaction
  - Smaller size but de-compaction is almost free - in some cases better
  - Works at the field level so we can use the data in its compact form
  - Compaction can use many of the features of compression

- Take a trade value… GBP 12,500,000.00
  - We might want to search on GBP values over 10 million
  - With compaction we can do that, compression we need to decompress first

Java binding and memory

- Typically Java Binding tools, like JAXB, JiBX and C24 create Java that looks like the data source

- While this is very convenient for the programmers it creates a lot of Java objects, this slowly consumes memory

- A typical FpML trade is around 8k in size, bind it to Java and it increases to around 25k

- 1 million FpML message in memory is going to cost anything from 8 to 25GB of RAM, add (HA) high availability and we hit 50GB
  - Expensive!
  - In-memory is still fast but 25k message over the network is very slow
  - And 25GB of data over a network or onto disk, even SSD is slow

SDOs

- SDOs or Simple Data Objects are basically Java Binding into a compact binary codec - From any XML format to binary

- We analyse the data model (or XML schema) not just the instance data so can do things like…
  - Reducing the 7 days of the week to just 3 bits
  - Commonly used Strings become lookups into a static table (1 or 2 bytes)
  - Currencies for example only need 1 byte
  - Date/Time with timezone can be stored in 6 bytes

- Bit-fields are compacted resulting in excellent compaction-ratios
  - Getters calculate the offset on the fly, mask and shift the data and return it

- There is NO change to the getter API between standard binding and SDOs

Standard Java Binding

```
<resetFrequency>
    <periodMultiplier>6</periodMultiplier>
    <period>M</period>
</resetFrequency>
```

- JAXB, JIBX, Castor and standard C24 generate something like …

```
public class ResetFrequency {
    private BigInteger periodMultiplier; // Positive Integer
    private Object period;               // Enum of D, W, M, Q, Y

    public BigInteger getPeriodMultiplier() {
        return this.periodMultiplier;
    }
    // constructors & other getters and setters
```

- In memory - 3 objects - at least 144 bytes
  - The parent, a positive integer and an enumeration for Period
  - 3 Java objects at 48 bytes is 144 bytes and it becomes fragmented in memory

Java Binding with SDOs

```
<resetFrequency>
    <periodMultiplier>6</periodMultiplier>
    <period>M</period>
</resetFrequency>
```

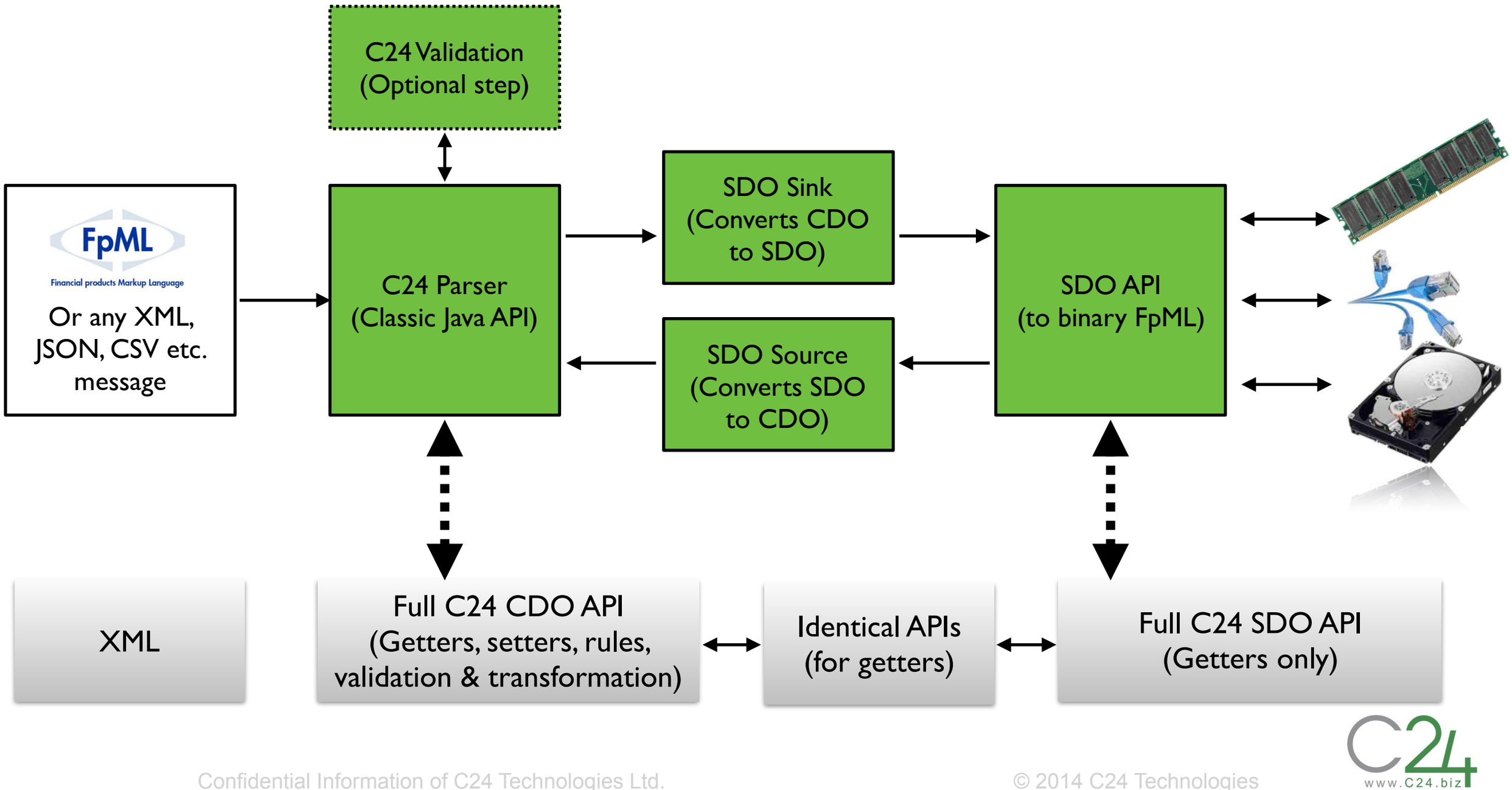
- Using C24 SDO binary codec we generate …

```
ByteBuffer data;    // From the root object
public BigInteger getPeriodMultiplier() {
        int byteOffset = 123;  // Actually a lot more complex
        return BigInteger.valueOf( data.get(byteOffset) & 0x1F );
    }
    // constructors & other getters
```

- In memory - 1 byte for all three fields
  - The root contains one ByteBuffer which is a wrapper for byte[]
  - The getters use bit-fields, Period is just 3 bits for values D, W, M, Q or Y

How it works

~5-8k	10-25k	**Size**	< 500 bytes
10k/sec	~1m/sec	**Performance**	~1m/sec



SDOs - So far*

- ISDA's sample Interest Rate Derivative (vanilla swap) is 7.4k
  - We randomised a few fields and created a few million for testing

- Zipped they are average 1,547 bytes
  - 1 million on disk require 1.5GB and takes 200 seconds to read/decompress
  - Parsing at 20k/sec would add another 50 seconds and need a **lot** of memory

- In memory they are roughly 25k in size (in roughly 400 objects)
  - It was difficult to fit 400k into 10GB of RAM - Lots of full GCs too

- With SDOs the average size was just 442 bytes
  - It took 9 seconds to read and parse 1 million from disk (SSD)
  - It took 415ms to search through all 1 million IRSs in memory (brute force)
  - 20 million fully parsed IRSs comfortably fit in 10GB of RAM

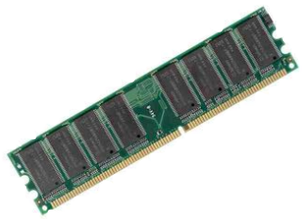
- Total saving on memory with FpML is roughly 50 times

* Tests were run on Java 1.7.0_55 on a MacBook Pro (2.7 GHz Intel i7) on a single core, we continue to improve these figures

SDO selling points

A 5 year leap into the future with Moore's law

- In a nutshell we can compact data by typically over 10 times
- You can get at least 10 times more data into Coherence
- Data takes up a 10th of its usual size
  - On disk or in memory
- Better use of network, memory and disk
  - Massive savings in infrastructure!



C24

www.C24.biz

Info@C24.biz John.Davies@C24.biz
@C24io @jtdavies

SDO landing page: http://sdo.c24.biz
http://ref.c24.biz/whitepapers/C24-SDOs-and-Coherence.pdf
http://ref.c24.biz/whitepapers/C24-SDOs-Big-Data-In-Memory.pdf