CCSDS MO Training

Module 3 - Developer

Overview

- Introduction
- Part 1 - Primer
  - The MO framework and services
  - The MAL and COM
- Part 2 - Defining an MO Service
  - Overview of the MO XML
  - Aspects of an MO service
  - An example service specification
- Part 3 – Java API theory
- Part 4 - Building an MO application
- Part 5 – ESA OSS
  - Available ESA OSS
  - ESA GitHub and Apache Maven
  - ESA Public Licence
- Part 6 – Legacy Systems
  - Integrating with legacy systems
- Summary

Introduction

- CCSDS and MO
- About the presenter
- Aims of the training module

CCSDS and the MO Standards

Images © the relevant Agency

Mission Operations

About the presenter

Aims of the training module

- The developer module focuses on the technologies of MO rather than the uses of MO
- The module will
  - Use the open source MO Java tools and software from ESA to define and build a new MO service
  - This new service will then be used to build small client and server applications
- There will be a discussion on the available Open Source Software
- The final part of the day will cover how to integrate MO M&C services with legacy Mission Control Systems
  - This theoretical exercise will use the example of ESA's SCOS 2000
- The aim of the training is that attendees leaving feeling confident on using the ESA MO tool chain to develop both MO services and applications

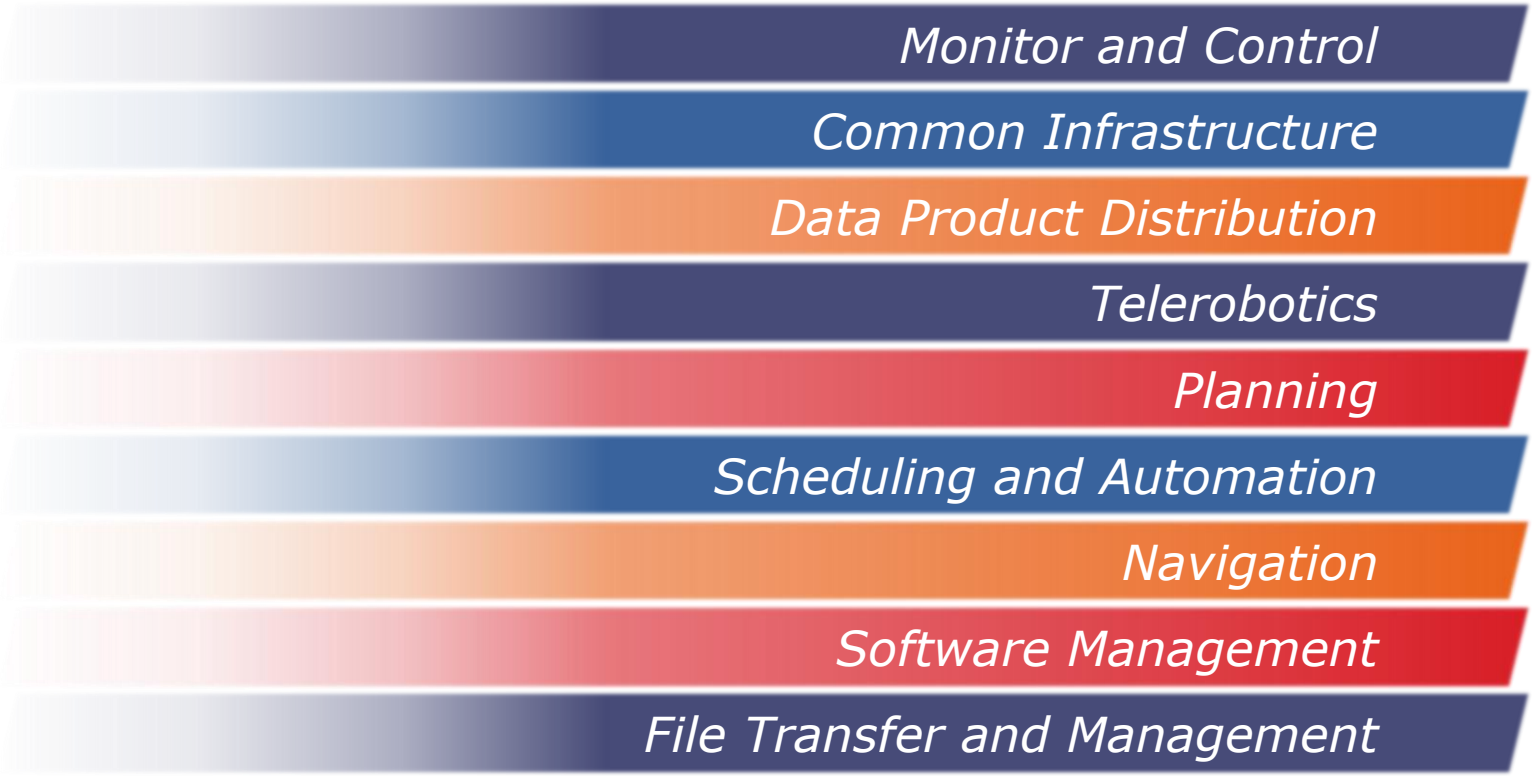
Part 1

- The MO framework and services
- The MAL and COM

CCSDS Mission Operations Services

- To alleviate this, CCSDS is standardising a set of services for Mission Operations
  - These services define a single specification for the exchange of similar information
- To support these standardised services CCSDS has also defined an open architecture and framework that is:
  - Independent from technology
  - Able to integrate new and legacy systems of different organisations
  - Designed to support the long lifetimes of space missions
  - Based on a Service Orientated Architecture (SOA)
- The CCSDS Mission Operations services provide the capability to hide the unavoidable difficulties that characterises the space environment whilst supporting future complexity needs

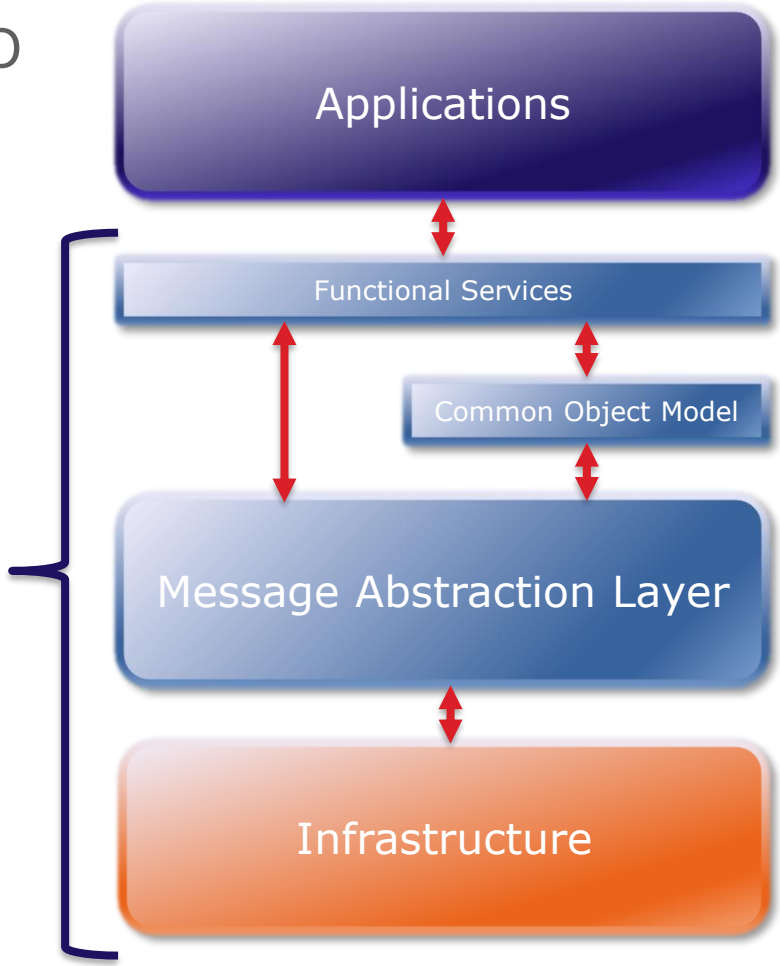
CCSDS Mission Operations Services



The MO Framework

- Central to the MO concept is the MO framework
  - Defines structure of an MO application
  - Provides generic model for data
    - Supports generic facilities such as archiving
  - Provides separation from technology

- It is defined by three specifications
  - A reference model
    - (CCSDS 520.1-M-1)
  - A Message Abstraction Layer (MAL)
    - (CCSDS 521.0-B-2)
  - A Common Object Model for data (COM)
    - (CCSDS 521.1-B-1)



The MO Framework and bespoke services

- MO framework also supports bespoke services
  - These are the services that you define for your:
    - Infrastructure
    - Missions
    - Agency
    - Anything else you identify

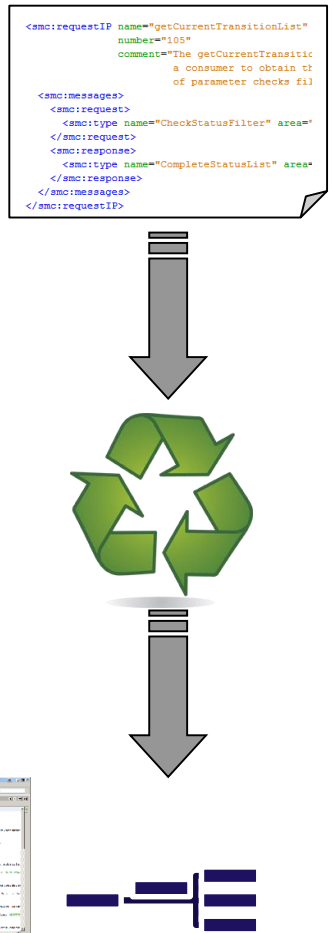
- If you define your bespoke services use the MO framework
  - In the same way the standard services provide future proof technology separation
  - The services specific to your environment can also benefit too

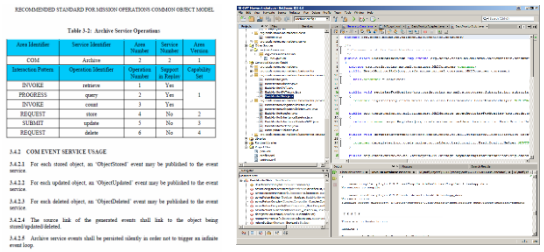
Message Abstraction Layer (MAL)

- The MAL is the building block for all MO services
- Provides the MO framework with its ability to separate from technology
  - Technology independent XML notation used to describe:
    - Service behaviour
    - Available operations
    - Message structures
- Provides the ability to change that technology at any time
  - Mapping from the XML to your chosen technologies
  - Defines the required abilities of communication technologies
- Simplifies the task to develop communicating applications
  - Specifies a fixed behaviour for applications
  - Expected behaviour of service providers
  - Expected behaviour of users of a service

MAL mappings and transformations

- The MAL defines a standard XML notation for service specification
- Mappings define transformations from the XML to:
  - Language mappings for specific programming languages
  - Technology mappings for 'on-the-wire' transports/encodings
  - Bespoke mappings are also supported
- Mappings are not service specific they work for all services
  - Services are defined in terms of the MAL
  - Mappings are defined in terms of the MAL
- So, from this we can automatically generate:
  - Documentation
  - Programming language APIs
  - System databases
  - …





Common Object Model (COM)

- The COM provides, out of the box:
  - Archiving
  - Tracking of remote activities and operations
  - Asynchronous event reporting
- The object model:
  - Provides a standard mechanism for the identification of data
  - Allows the development of standard capabilities such as archiving
- Defined using the MAL notation
  - Technology agnostic

Part 2

- Defining an MO Service

MO XML Overview

- The MO XML notation is central to the specification of an MO service
  - Everything is derived from that
- It can be used to generate:
  - Documentation
  - Databases
  - Programming APIs
- It is split into two XML Schemas:
  - One for basic MAL usage
  - One for the higher COM usage
- It can be downloaded from the ESA GitHub:
  - https://github.com/esa/CCSDS_MO_XML
  - From the MO_XML/src/main/resources folder
- A graphical editor exists to help you develop services
  - See final slide in this section

MO XML: Top level specification

- The top level node of an MO XML document is:

```
<mal:specification xmlns:com="http://www.ccsds.org/schema/COMSchema"
                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                   xmlns:mal="http://www.ccsds.org/schema/ServiceSchema">
  ...
</mal:specification>
```

- It contains a sequence of Area nodes

```
<mal:area name="MyArea" number="22" version="1">
  ...
  <mal:dataTypes>
    ...
  </mal:dataTypes>
  <mal:errors>
    ...
  </mal:errors>
</mal:area>
```

- Each Area can contain a sequence of:
  - Services
  - Data types
  - Error definitions

- Each Area requires a name, number, and version
  - It must be unique inside the XML document. It should be unique outside that too but that is not enforced

MO XML: Services

- Each Service contains a sequence of:
  - Capability Sets containing Operation specifications
  - Data types
  - Error definitions
  - Optionally a COM usage section

```
<mal:service name="MyService" number="1">
  <mal:capabilitySet number="1">
    ...
  </mal:capabilitySet>
  <mal:dataTypes>
    ...
  </mal:dataTypes>
  <mal:errors>
    ...
  </mal:errors>
  <com:features>
    ...
  </com:features>
</mal:service>
```

- Each Service requires a name and number
  - It must be unique inside the containing Area

MO XML: Operations

- Inside a Capability set each Operation is one of the following:

```
<mal:sendIP/>
<mal:submitIP/>
<mal:requestIP/>
<mal:invokeIP/>
<mal:progressIP/>
<mal:pubsubIP/>
```

- Each Operation contains a sequence of:
  - Message specifications
  - Errors returned

```
<mal:messages>
  ...
</mal:messages>
<mal:errors>
  ...
</mal:errors>
```

- Each Operation requires a unique name and number

MO XML: Operation messages

- For each operation, one must define each message
  - The possible messages are dependent on the interaction pattern type
  - The XML Schema enforces that the correct messages are specified

- Each message contains the specification of the contents:
  - The 'field' wrapper allows the message part to be named
  - The name can then be referenced from documentation or in any auto-generated API code

```
<mal:messages>
  <mal:submit>
    <mal:field name="objectType">
      <mal:type name="ObjectType" area="COM"/>
    </mal:field>
    <mal:field name="domain">
      <mal:type list="true" name="Identifier" area="MAL"/>
    </mal:field>
  </mal:submit>
</mal:messages>
```

MO XML: Data type overview

- Inside a Data type section is a sequence of the following:

```
<mal:enumeration/>
<mal:composite/>
```

- The MAL XML specification also contains top level specifications for:
  - Fundamental abstract types: Element, Attribute, and Composite
  - All concrete Attribute types:
    - Blob
    - Boolean
    - Duration
    - Time
    - FineTime
    - Float
    - Double
    - Identifier
    - String
    - Octet
    - UOctet
    - Short
    - UShort
    - Integer
    - UInteger
    - Long
    - ULong
    - URI

MO XML: Enumeration data types

- Enumerations are defined as a sequence of possible values:

```
<mal:enumeration name="QoSLevel" shortFormPart="21">
  <mal:item value="BESTEFFORT" nvalue="1" comment="Used for Best Effort QoS"/>
  <mal:item value="ASSURED"    nvalue="2" comment="Used for Assured QoS"/>
  <mal:item value="QUEUED"     nvalue="3" comment="Used for Queued QoS"/>
  <mal:item value="TIMELY"     nvalue="4" comment="Used for Timely QoS"/>
</mal:enumeration>
```

- Each item of the enumeration has three values:
  - A string value
  - A numeric value
  - An ordinal value (not written but implied)
    - This is the index of the item in the Enumeration

- Most encodings will pass the item ordinal as this is the smallest value
  - For example:
    - If the final item above ("TIMELY") had a numeric value of 10,000,000
    - It would still have an ordinal value of 3
      - That is its position in the sequence of items based on a zero index (so 0, 1, 2, 3 etc)
    - Encoding "3" is more efficient than encoding "10,000,000", or "TIMELY"

MO XML: Composite data types

- Composites are defined as an initial Composite to extend:

```
<mal:composite name="Subscription" shortFormPart="23">
  <mal:extends>
    <mal:type name="Composite" area="MAL"/>
  </mal:extends>
```

- Followed by a list of fields:

```
  <mal:field name="subscriptionId" canBeNull="false">
    <mal:type name="Identifier" area="MAL"/>
  </mal:field>

  <mal:field name="entities" canBeNull="false">
    <mal:type list="true" name="EntityRequest" area="MAL"/>
  </mal:field>
</mal:composite>
```

MO XML: COM usage pt1

- The COM usage section allows a service to specify:
  - The COM Objects it defines
  - The COM Events it defines
  - Its usage of the COM Archive
  - Its usage of COM Activity Tracking
    - The Archive and Activity Tracking section only contain a comment field
    - This currently has no effect on the code generation
    - But defines the required usage for service implementations

```
<com:features>
  <com:objects>
    ...
  </com:objects>
  <com:events>
    ...
  </com:events>
  <com:archiveUsage/>
  <com:activityUsage/>
</com:features>
```

MO XML: COM usage pt2

- The specification of a COM object or event are the same:

```
<com:event name="MyEvent" number="1">
  <com:objectType>
    <mal:type name="UInteger" area="MAL"/>
  </com:objectType>
  <com:relatedObject>
    <com:objectType number="3" service="SomeService" area="SomeArea"/>
  </com:relatedObject>
  <com:sourceObject>
    <com:objectType number="5" service="ActivityTracking" area="COM"/>
  </com:sourceObject>
</com:event>
```

Example service: Overview

- In the next part we will build a small application set to demonstrate MO
- For this we'll need a service specification
  - We could use a existing one
  - But they tend to be rather complex and not immediately interesting
- Our example service will have:
  - A single Area
  - A single Service
  - An example of each interaction pattern
  - Some types specific to our service
  - A service specific error
  - Have a simple use of COM Activity Tracking

Example service: Top level

- First we need to define our top level specification
  - At this stage it will include our single Area and Service:

```
<mal:specification xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                   xmlns:mal="http://www.ccsds.org/schema/ServiceSchema">
  <mal:area name="MOTraining" number="101" version="1">
    <mal:service name="SimSpacecraft" number="1">

    </mal:service>
  </mal:area>
</mal:specification>
```

- We'll also define a service specific error for later use in our operations:

```
      <mal:errors>
        <mal:error name="MODE_CHANGE_FAILED" number="99999"
                                comment="Mode change failed"/>
      </mal:errors>
    </mal:service>
```

Example service: Data types pt1

- Next we'll define our data types
  - There is no fixed ordering on how this should be done
  - It's usually an iterative process
  - By doing this first it makes them available when we come to define the operations

- Inside the service specification we'll add:
  - A simple enumeration to represent our spacecraft operational mode:

```
<mal:dataTypes>
  <mal:enumeration name="SpacecraftMode" shortFormPart="1">
    <mal:item value="SAFE" nvalue="1"/>
    <mal:item value="ERROR" nvalue="2"/>
    <mal:item value="OPERATIONAL" nvalue="3"/>
    <mal:item value="EXPERIMENTAL" nvalue="4"/>
  </mal:enumeration>
```

Example service: Data types pt2

- A simple Composite for housekeeping TM reporting:

```
<mal:composite name="BasicTelemetry" shortFormPart="2">
    <mal:field name="spacecraftMode">
      <mal:type area="MOTraining" service="SimSpacecraft" name="SpacecraftMode"/>
    </mal:field>

    <mal:field name="latitude">
      <mal:type area="MAL" name="Double"/>
    </mal:field>

    <mal:field name="longitude">
      <mal:type area="MAL" name="Double"/>
    </mal:field>

    <mal:field name="counter">
      <mal:type area="MAL" name="Short"/>
    </mal:field>
  </mal:composite>
</mal:dataTypes>
```

Example service: Data types pt3

- Next we'll add two composites that have an abstract base composite:
  - Base composite:

```
<mal:composite name="BaseComposite">
  <mal:field name="summaryInfo">
    <mal:type area="MOTraining" service="SimSpacecraft" name="BasicTelemetry"/>
  </mal:field>
</mal:composite>
```

  - First concrete composite:

```
<mal:composite name="FirstComplexComposite" shortFormPart="3">
  <mal:extends>
    <mal:type area="MOTraining" service="SimSpacecraft" name="BaseComposite"/>
  </mal:extends>

  <mal:field name="temperatureA">
    <mal:type area="MAL" name="UInteger"/>
  </mal:field>
  <mal:field name="temperatureB">
    <mal:type area="MAL" name="UInteger"/>
  </mal:field>
  <mal:field name="temperatureC">
    <mal:type area="MAL" name="UInteger"/>
  </mal:field>
</mal:composite>
```

Example service: Data types pt4

- Now the second concrete composite:

```
<mal:composite name="SecondComplexComposite" shortFormPart="4">
  <mal:extends>
    <mal:type area="MOTraining" service="SimSpacecraft" name="BaseComposite"/>
  </mal:extends>

  <mal:field name="powerA">
    <mal:type area="MAL" name="Short"/>
  </mal:field>

  <mal:field name="powerB">
    <mal:type area="MAL" name="Short"/>
  </mal:field>

  <mal:field name="powerC">
    <mal:type area="MAL" name="Short"/>
  </mal:field>
</mal:composite>
```

Example service: Service pt1

- For our service, we now add the first set of operations
  - These operations should all be put in a single capability set:

```
<mal:capabilitySet number="1">
</mal:capabilitySet >
```

- SEND interaction “testSend” with a single Boolean argument

```
<mal:sendIP name="testSend" number="1" supportInReplay="false">
  <mal:messages>
    <mal:send>
      <mal:field name="sendArgument">
        <mal:type area="MAL" name="Boolean"/>
      </mal:field>
    </mal:send>
  </mal:messages>
</mal:sendIP>
```

Example service: Service pt2

- SUBMIT interaction “enableTelemetry” with a single Boolean argument

```
<mal:submitIP name="enableTelemetry" number="2" supportInReplay="false">
  <mal:messages>
    <mal:submit>
      <mal:field name="enableTM">
        <mal:type area="MAL" name="Boolean"/>
      </mal:field>
    </mal:submit>
  </mal:messages>
</mal:submitIP>
```

Example service: Service pt3

- REQUEST interaction “isTmEnabled” with a single Boolean argument
  - Returning a Boolean and Short

```
<mal:requestIP name="isTmEnabled" number="3" supportInReplay="false">
  <mal:messages>
    <mal:request>
      <mal:field name="returnCount">
        <mal:type area="MAL" name="Boolean"/>
      </mal:field>
    </mal:request>

    <mal:response>
      <mal:field name="generationEnabled">
        <mal:type area="MAL" name="Boolean"/>
      </mal:field>

      <mal:field name="generationCount">
        <mal:type area="MAL" name="Short"/>
      </mal:field>
    </mal:response>
  </mal:messages>
</mal:requestIP>
```

Example service: Service pt4

- INVOKE interaction "invokeMode" with a single SpacecraftMode argument
  - Acknowledging with a SpacecraftMode and Returning a SpacecraftMode
  - That it can return an error, containing a SpacecraftMode as extra information

```
<mal:invokeIP name="invokeMode" number="4" supportInReplay="false">
  <mal:messages>
    <mal:invoke>
      <mal:field name="desiredMode">
        <mal:type area="MOTraining" service="SimSpacecraft" name="SpacecraftMode"/>
      </mal:field>
    </mal:invoke>
    <mal:acknowledgement>
      <mal:field name="currentMode">
        <mal:type area="MOTraining" service="SimSpacecraft" name="SpacecraftMode"/>
      </mal:field>
    </mal:acknowledgement>
    <mal:response>
      <mal:field name="newMode">
        <mal:type area="MOTraining" service="SimSpacecraft" name="SpacecraftMode"/>
      </mal:field>
    </mal:response>
  </mal:messages>

  <mal:errors>
    <mal:errorRef comment="The spacecraft was not able to attain the requested mode.">
      <mal:type area="MOTraining" service="SimSpacecraft" name="MODE_CHANGE_FAILED"/>
      <mal:extraInformation comment="The extra information field holds the desired mode.">
        <mal:type area="MOTraining" service="SimSpacecraft" name="SpacecraftMode"/>
      </mal:extraInformation>
    </mal:errorRef>
  </mal:errors>
</mal:invokeIP>
```

Example service: Service pt5

- PROGRESS interaction “testProgress” with an Integer and a Boolean arguments
  - Acknowledging an Integer, Updating an Integer, Returning the abstract BaseComposite

```
<mal:progressIP name="testProgress" number="5" supportInReplay="false">
  <mal:messages>
    <mal:progress>
      <mal:field name="progressUpdateCount">
        <mal:type area="MAL" name="Integer"/>
      </mal:field>
      <mal:field name="respondWithSecondComplexType">
        <mal:type area="MAL" name="Boolean"/>
      </mal:field>
    </mal:progress>
    <mal:acknowledgement>
      <mal:field name="requestedCount">
        <mal:type area="MAL" name="Integer"/>
      </mal:field>
    </mal:acknowledgement>
    <mal:update>
      <mal:field name="currentCount">
        <mal:type area="MAL" name="Integer"/>
      </mal:field>
    </mal:update>
    <mal:response>
      <mal:field name="finalUpdate">
        <mal:type area="MOTraining" service="SimSpacecraft" name="BaseComposite"/>
      </mal:field>
    </mal:response>
  </mal:messages>
</mal:progressIP>
```

Example service: Service pt6

- We now add the second operation set
  - Add the following operations:
    - PUBSUB interaction "monitorHousekeepingTM"
      - Notify a BasicTelemetry composite

```
<mal:capabilitySet number="2">
  <mal:pubsubIP name="monitorHousekeepingTM" number="6" supportInReplay="false">
    <mal:messages>
      <mal:publishNotify>
        <mal:type area="MOTraining" service="SimSpacecraft" name="BasicTelemetry"/>
      </mal:publishNotify>
    </mal:messages>
  </mal:pubsubIP>
</mal:capabilitySet>
```

- This operation should be put in a second capability set

Example service: COM usage

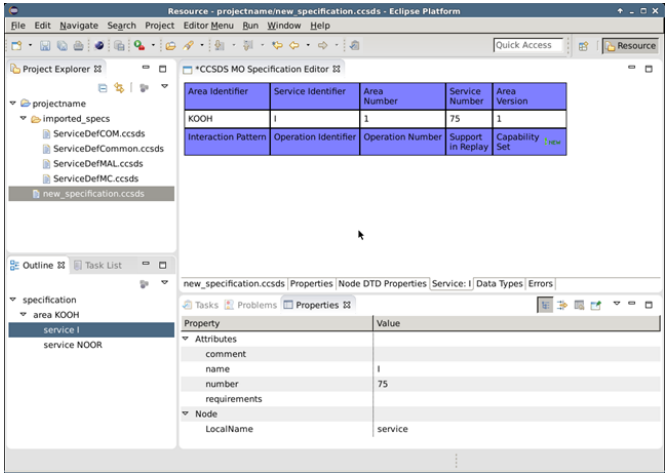
- For our service we are going to use COM Activity Tracking
  - This will be used to report the progress of the operations
  - This is a standard use of Activity Tracking defined in the COM specification
- For the XML we will a section as below:

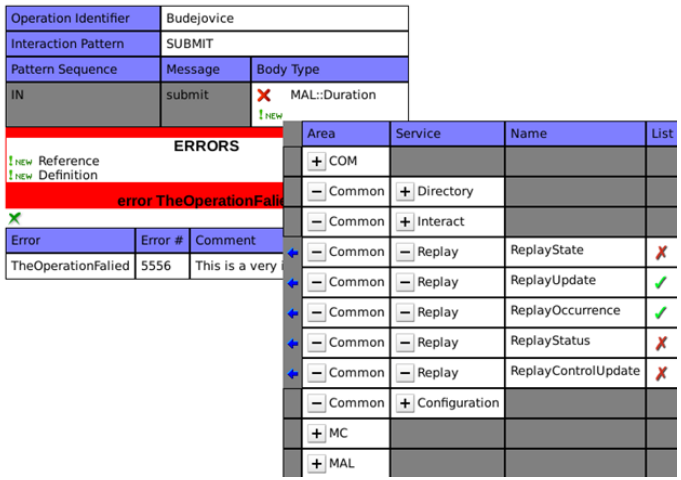
```
<com:features>
  <com:activityUsage comment="The COM Activity service should be used to monitor
                              the transfer and execution of the operations."/>
</com:features>
```

- This will not have an effect on the code generation at this point
  - However, it will appear in any documentation
  - It may also have an effect on some database generation
- Our service specification is now complete!

ESA MO Graphical XML Editor

- ESA have produced a graphic MO service editor
- Provides a table view of the XML
  - similar to that seen in the service specification documents
- Packaged as an Eclipse plug-in
- Can be downloaded from the ESA GitHub
  - Pre-built versions are available:
  - https://github.com/esa/CCSDS_MO_GraphicalEditor/tree/master/ccsds.mo.editor.site/plugins
- There is an installation and user guide available from ESA
  - Ref: CCSDS-MO-GE-SUM





Images © ESA

Part 3

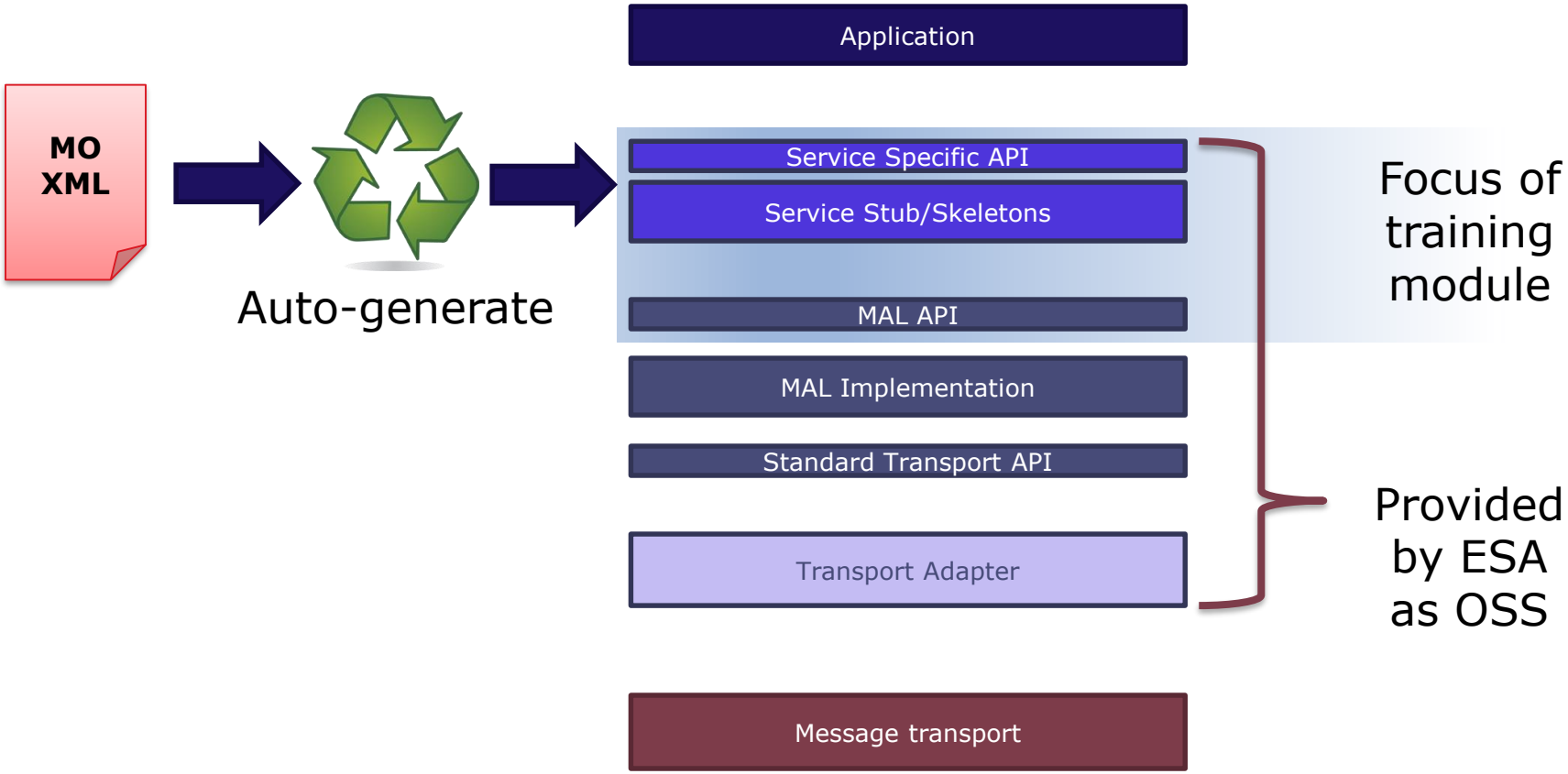
- Java API Theory

Java API Overview

- First standard MAL API created
  - Published in April 2013
- Specification is held on the CCSDS website:
  - http://public.ccsds.org/publications/archive/523x1m1.pdf
  - This is the specification of the API rather than a manual of how to use the API
- Extensively used and tested
  - Split into two parts:
    - Generic service agnostic
    - Auto-generated service specific
- ESA implementation of the MAL API is online
  - Held in Maven Central
  - Also contains the JavaDoc of the API

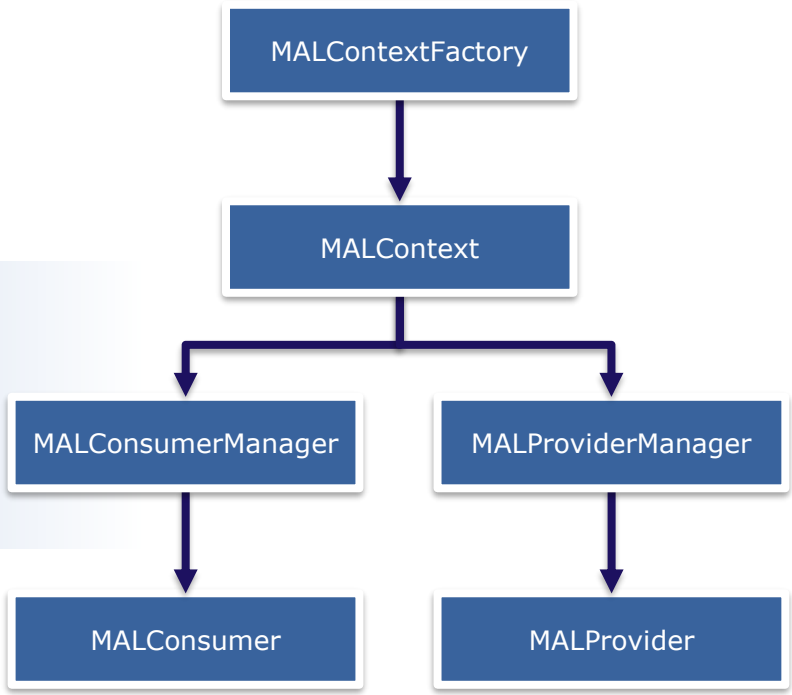
Stack overview

- Components of an MO application:



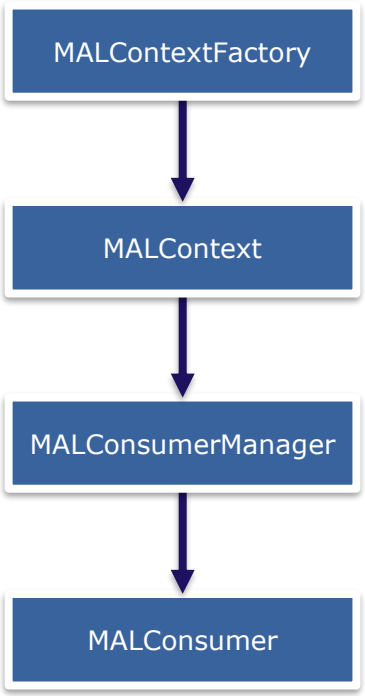
MAL API organisation

- The top level MAL package is:
  - org.ccsds.moims.mo.mal
- Contains the following packages:
  - consumer
    - Contains classes for consumers
  - provider
    - Contains classes for providers
  - structures
    - Contains data type classes
  - broker
    - Contains classes implementing PubSub brokers
  - accesscontrol
    - Contains interfaces for the Access Control component
  - transport
    - Contains classes for implementing message transports
  - encoding
    - Contains classes for implementing message encoders



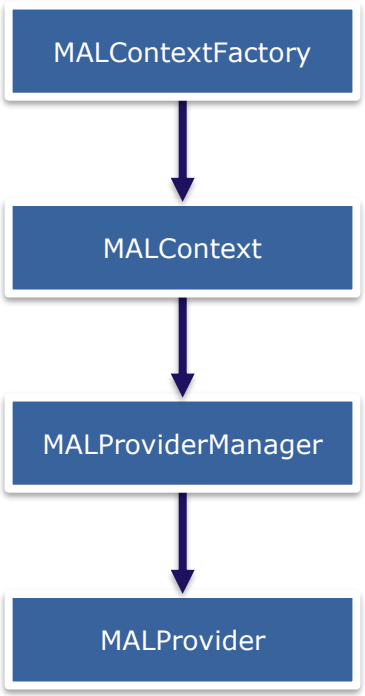
Basic approach for creating consumers

- Create a MAL context factory
- Register services and areas
- Create a MAL context
- Obtain provider URIs
- Create a MALConsumer
  - Holds the consumer connection information
- Wrap it in the auto-generated service stub
- Call some operations!
- When finished close the MAL context
  - Deregister from provider first if using PubSub though



Basic approach for creating providers

- Create a MAL context factory
- Register services and areas
- Implement some services!
  - Note that a provider application can provide more than one service
- Create a MAL context
- Create a MALProvider for each of our services
- Share provider URIs
- Wait for some interaction from consumers
- When finished close the MAL context



Area and Service registration

- Every type, standard and auto-generated, has a numeric short form
  - This is similar to the Java serialisation identifier
  - It is used by the message encoders and transports to correctly decode the incoming messages
- Services contain objects that hold information about the operations
  - Needed for the lookup of the operation details when receiving a message
- All used services and types must be registered before they can be used
  - Done via the MALContextFactory
- Auto-generated areas and services contain a xxxHelper class that contain methods to simplify this:
  - Services have a static init method that registers itself and all operations, types, and error codes
  - Areas have a static init method that registers itself and all types and errors
  - Areas have a static deepInit method that calls init and also init of all contained services
- Getting this wrong is the main cause of encode/decode errors

Consumer operation calling

- The Auto-generated stub class contains the service operations
- Service operations can be called in one of three ways:
  - Synchronous
    - Method call does not return until receive the first response message
  - Asynchronous
    - Method call returns immediately
  - Continue
    - Similar to asynchronous, allows a consumer to continue an existing transaction
- How to receive return messages:
  - For synchronous calls the first returned message is the return value of the method
    - This is the Response message from REQUEST, and the ACK message from INVOKE and PROGRESS
  - For multi-message interactions and asynchronous calls need to supply a callback object
    - Class is auto-generated and service specific
    - Class is made from the concatenation of the service name and the word “Adapter”
    - Override methods to receive the different parts of the interaction response

Implementing a Provider: Skeleton types

- For a service the auto-generator creates a skeleton of the operations of the service
- Two types of service skeleton are auto-generated:
  - An inheritance skeleton
  - A delegation skeleton
- Inheritance skeleton
  - Provides an abstract implementation of the service
  - Operations are defined as abstract and therefore must be implemented
  - Can be passed directly to the MALProviderManager when creating a provider instance
  - Simplest method of creating a provider, just extend and implement methods
- Delegation skeleton
  - Provides a complete implementation of the service interface
  - Delegate class must be passed to constructor of the delegation skeleton
  - Delegation skeleton calls delegate when invoked by a consumer
  - Delegation skeleton passed to the MALProviderManager when creating a provider instance
  - Slightly more complex model, but useful when cannot just extend Inheritance Skeleton
    - For example if implementing multiple services in one class as multiple inheritance is not permitted in Java

Provider operation reception

- Operation implementation signature:
  - Operation arguments are passed as method arguments
  - Returns a void unless it is a REQUEST interaction
  - For multi-message interactions an operation specific class is provided for sending the response messages
    - This is always the final argument of the method
- How to return data/messages:
  - For SUBMIT interactions, just return from the method
    - To return an error just throw a MALInteractionException
  - For REQUEST interactions, response is the return value of the method
    - To return an error just throw a MALInteractionException
  - For INVOKE and PROGRESS, supplied interaction object methods must be used:
    - It's the last argument of the method
    - Provides methods for sending ACK, UPDATE, and RESPONSE messages
    - Also provides a method for returning an error

Publish and Subscribe: Consumer

- Follows normal procedure for initialising a consumer service stub
- From service specific stub use the xxxRegister method to register for PubSub updates
  - The ‘xxx’ part is the PubSub operation name
  - Takes two arguments
    - First is the subscription details
    - Second is the callback adapter used to receive the updates
- To Deregister use the xxxDeregister method in the service specific stub
  - The ‘xxx’ part is the PubSub operation name
  - Takes a single argument of the list of active subscriptions to deregister

Publish and Subscribe: Provider

- Decision to be a publisher is decided when creating the MALProvider object:
  - Second to last Boolean argument determines whether PubSub is supported by the provider
  - Last argument determines whether an internal Broker is to created or an external broker is to be used
    - Passing NULL requests that an internal broker be created
- Both skeletons (Inheritance and Delegation) provide a createXxxPublisher method
  - The 'xxx' part is the PubSub operation name
  - This creates the object used to interact with the broker by the provider
- Providers must register with the broker before they are allowed to publish
  - Must supply a list of entity keys of the objects they are going to publish
    - Wildcards are permitted
- Publisher object is then used to perform the actual publish of updates
- Providers should deregister with the broker once they have finished all publishing

Part 4

- Building an MO application

Example applications: Overview

- In this section we will gain some hands-on experience
  - We take the simple MO service from the XML section
  - Using this we build some Consumer and Provider applications
- Build the applications in several stages
  - Each stage will be executable
- Requirements:
  - Java 8 update 40 minimum
    - Consumer application uses JavaFX for its MMI
  - Apache Maven 3.2 or higher
- Optionally
  - A Maven aware IDE
    - Netbeans is preferred due to its strong Maven support
    - It is assumed that the attendee is proficient with their IDE choice and how it interacts with Apache Maven

Example applications: Stages

- This section builds the application is stages
  - Each stage will be executable and builds incrementally on the previous stage
- Stage 0
  - In this stage we take our service XML and use it to auto-generate Java code
- Stage 1
  - In this stage we start development of our applications
    - At the end our consumer will be able to use the SEND interaction with the provider
- Stage 2
  - In this stage we add in an MMI to our consumer
    - At the end we will have implemented the SUBMIT and REQUEST interactions
- Stage 3
  - In this stage we implement the INVOKE and PROGRESS interactions
- Stage 4
  - In this stage we implement the PubSub interaction
- Stage 5
  - In this stage we add in some COM Activity Tracking

Running the Example applications

- To test each stages there are two applications
  - A Consumer
  - A Provider

- To start each application there are three possible options

  1) Launch directly from your IDE
     - For the Provider the main is called esa.mo.training.provider.MyProviderMain
     - For the Consumer the main is called esa.mo.training.consumer.MyConsumerMain

  2) Launch from Windows Explorer
     - In the root directory of each stage there are two Windows batch files:
     - For the Provider the file is called runProvider.bat
     - For the Consumer the file is called runConsumer.bat

  3) Launch from a command prompt
     - In the root directory of each stage there are two Windows batch files:
     - For the Provider the file is called runProvider.bat
     - For the Consumer the file is called runConsumer.bat

- The Provider application should always be started first in this example

Stage 0: Auto-generating from the service XML

- Open Stage 0 project
  - The XML from the previous section is already included
- Build the project as this will trigger the auto-generation from the XML
  - Our XML file is held in:
    - src/main/resources
  - Auto-generation is done by ESA's StubGenerator application
    - This is called from the Apache Maven build file (pom.xml)
    - StubGenerator application can be invoked as a Maven plugin
  - The auto-generated Java is held in:
    - target/generated-sources/stub

Stage 1 : The basic applications

- Open “Stage 1 – begin” project
  - Build the project as this will trigger the auto-generation from the XML
  - Our Java source code is held in:
    - src/main/java
  - The auto-generated Java is held in:
    - target/generated-sources/stub

- The project contains the following files of interest:
  - In the provider package of esa.mo.training.provider:
    - MyProviderMain
      - Contains the provider application Java main method
    - MyProvider
      - Contains the MO application logic
    - MySimSpacecraftServiceImpl
      - Outline implementation of our SimSpacecraft service using inheritance skeleton
  - In the consumer package of esa.mo.training.consumer:
    - MyConsumer
      - Contains the consumer application Java main method

Stage 1 : Implementing the service

- Open the provider MySimSpacecraftServiceImpl file
  - At the moment this class is empty
- First we must extend the auto-generated service inheritance skeleton:

```
public class MySimSpacecraftServiceImpl extends SimSpacecraftInheritanceSkeleton
```

- This will not compile, so import the correct class and get the IDE to auto-generate the missing methods
  - These methods match the operations of our service
  - For each method add in a simple System.out message
  - For now, return null from the "isTmEnabled" method
- Our service implementation is complete for this stage

Stage 1 : Implementing the provider

- Open the provider MyProvider file
  - At the moment this class is empty
  - This class will be used to manage our MAL connection and also hold any services will provide
- The ESA OSS contains support libraries for simplifying the development of service providers
  - We will use these libraries as they simplify the process
- First we must extend the ESA OSS BaseMalServer class:

```
public class MyProvider extends BaseMalServer
```

  - This will not compile, so import the correct class and get the IDE to auto-generate the missing methods
- For the constructor super arguments supply:

```
super(new IdentifierList(), new Identifier("SPACE"));
```

- In the subInitHelpers generated method add:

```
MOTrainingHelper.deepInit(malefr);
```

- In the subInit generated method add:

```
serviceImpl = new MySimSpacecraftServiceImpl();
createProvider(SimSpacecraftHelper.SIMSPACECRAFT_SERVICE, serviceImpl, false);
StructureHelper.storeURIs("trainingServiceURI.properties", ep.getURI(), ep.getURI());
```

  - You will also need to create a private member variable to hold the service implementation

Stage 1 : Implementing the provider main

- Open the provider MyProviderMain file
  - At the moment this class just loads in some properties and then waits for a keypress
  - There are three 'ToDo' comments that need to be replaced
- The first one is where we create an instance of our provider class:

```
MyProvider provider = new MyProvider();
```

- The second one is where we initialise it and start it:

```
provider.init("SimSpacecraft", null);
provider.start();
```

- The final one, after the wait for a key press, is where we close things down and clean up:

```
provider.stop();
```

- And that's it for the provider
  - The provider should compile fully now

Stage 1 : Implementing the consumer pt1

- Open the consumer MyConsumer file
  - At the moment this class just loads in some properties
  - There are five 'ToDo' comments that need to be replaced
- The first one is where we initialise our services and areas

```
MALHelper.init(MALContextFactory.getElementFactoryRegistry());
MOTrainingHelper.deepInit(MALContextFactory.getElementFactoryRegistry());
```

- The second one is where we create our consumer MAL context:

```
MALContext mal = MALContextFactory.newFactory().createMALContext(System.getProperties());
MALConsumerManager consumerMgr = mal.createConsumerManager();
```

Stage 1 : Implementing the consumer pt2

- The third one is where we create our consumer connection to the provider:

```
SimSpacecraftStub simSpacecraftStub = new
     SimSpacecraftStub(consumerMgr.createConsumer((String)null,
            new URI(tpuri), new URI(tburi),
            SimSpacecraftHelper.SIMSPACECRAFT_SERVICE,
            new Blob("".getBytes()),
            domain, new Identifier("GROUND"),
            SessionType.LIVE, new Identifier("LIVE"),
            QoSLevel.BESTEFFORT, System.getProperties(), new UInteger(0)));
```

- The fourth one is where we make our actual call to the provider:

```
simSpacecraftStub.testSend(true);
```

- The final one is where we close things down and clean up:

```
mal.close();
```

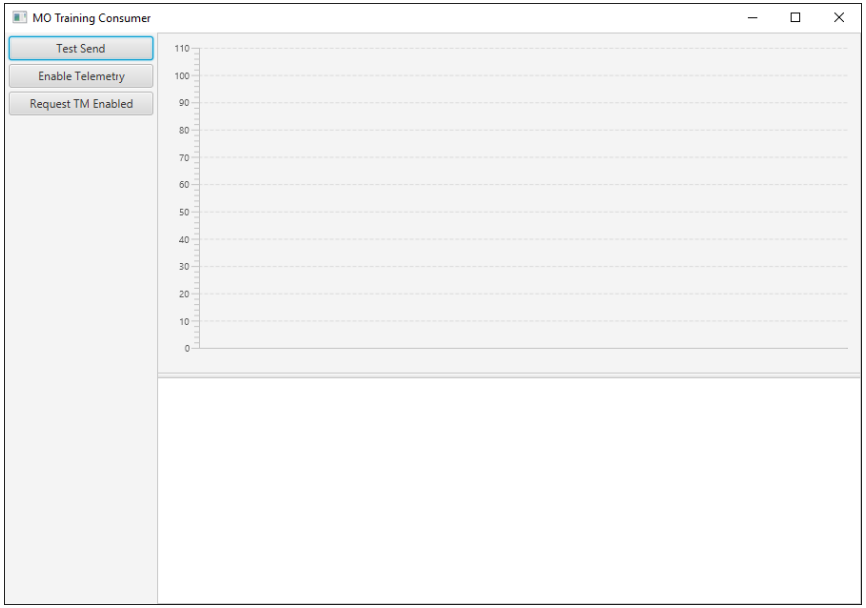
- And that's it for the consumer, it should all now compile

Stage 1: Running the applications

- To test this stage, first start the provider application
  - From either the IDE, from Windows Explorer, or a command prompt
- This should start up, create the provider, and then wait for a key press
  - Entering a ‘q’ will shut the provider down
- Once the provider has started you can run the consumer
  - From either the IDE, from Windows Explorer, or a command prompt
- You should see the message you added to the provider for the testSend operation being displayed on the provider console
- If this is not working you can compare your code to that in the Stage “1 – complete” project
  - This is a completed version of this stage that is to be used to verify correct behaviour

Stage 2: Implementing SUBMIT and REQUEST

- Open Stage 2 project
  - Build the project as this will trigger the auto-generation from the XML
- The main change from the previous stage is that we now have an MMI for the consumer
  - The previous code is still there but the main is now part of the JavaFX part
- The JavaFX MMI code is outside of the scope of the training module
  - It should be noted that the MMI code does not contain any MO code
  - It is just there to make the consumer code clearer and more interesting



Stage 2 : Continuing the service

- No changes are required to the MyProviderMain or MyProvider classes
- Open the provider MySimSpacecraftServiceImpl file
  - We are going to implement the "enableTelemetry" and "isTmEnabled" operations
- For the "enableTelemetry" operation, replace the ToDo with:

```
this.generating = enableTM;
```

- For the "isTmEnabled" operation:
  - We must return whether the TM generation is enabled or not
  - The supplied Boolean argument indicates whether we should also return the current TM report counter
  - Replace the ToDo with:

```
if (returnCount)
{
  return new IsTmEnabledResponse(generating, currentTmReportCounter);
}
else
{
  return new IsTmEnabledResponse(generating, null);
}
```

- Our service implementation is complete for this stage

Stage 2 : Continuing the consumer

- Open the consumer MyConsumer file
  - The code has been reworked into a class rather than just a single method
    - The functionality is identical
    - However there is now a method for each of the consumer calls
      - These are called from the MMI code
  - There are two 'ToDo' comments that need to be replaced
- The first one is for the SUBMIT operation:

```
simSpacecraftStub.enableTelemetry(value);
```

- The second one is for the REQUEST operation:

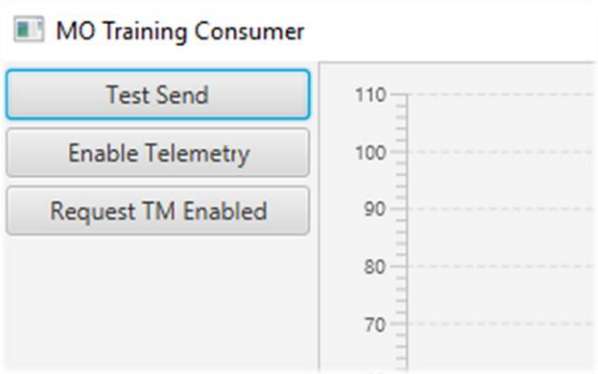
```
IsTmEnabledResponse rspn = simSpacecraftStub.isTmEnabled(value);
```

- The response can be displayed on the MMI using the "display" callback object:

```
display.appendLogValue("Request response: TM enabled is "
            + rspn.getBodyElement0()
            + " : Current TM counter is "
            + rspn.getBodyElement1());
```

Stage 2: Running the applications

- To test this stage, first start the provider application as before
  - From either the IDE, from Windows Explorer, or a command prompt
- This should start up, create the provider, and then wait for a key press
  - Entering a ‘q’ will shut the provider down
- Once the provider has started you can run the consumer as before
  - From either the IDE, from Windows Explorer, or a command prompt
- The JavaFX MMI should start
  - There are three buttons
    - one for each of the implemented operations
  - Pressing them should cause the appropriate operation to be called on the service provider



- If this is not working you can compare your code to that in the Stage 3 project

Stage 3: Implementing INVOKE and PROGRESS

- Open Stage 3 project
  - Build the project as this will trigger the auto-generation from the XML
- In this stage we implement the more complex INVOKE and PROGRESS patterns
- The implementation at this stage will be very simple
  - The next stage (4) implements these operations in a more interesting way
  - Fundamentally it is the same as we will do now but more interactive

Stage 3 : Continuing the service

- Open the provider MySimSpacecraftServiceImpl file
  - We are going to implement the "invokeMode" and "testProgress" operations
- For the "invokeMode" operation, replace the ToDo with:

```
interaction.sendAcknowledgement(currentSpacecraftMode);
currentSpacecraftMode = desiredMode;
interaction.sendResponse(desiredMode);
```

- For the "testProgress" operation, replace the ToDo with:

```
interaction.sendAcknowledgement(progressUpdateCount);
interaction.sendUpdate(1);
interaction.sendResponse(new SecondComplexComposite(null,
                         (short)2, (short)20, (short)200));
```

- Our service implementation is complete for this stage
  - These are very trivial implementations of these operations but they are fine for this stage

Stage 3 : Continuing the consumer

- Open the consumer MyConsumer file
  - There are two 'ToDo' comments that need to be replaced
- The first one is for the INVOKE operation:

```
simSpacecraftStub.invokeMode(desiredMode, new MySimSpacecraftServiceAdapter(display));
```

- The second one is for the PROGRESS operation:

```
simSpacecraftStub.asyncTestProgress(count, second,
                                    new MySimSpacecraftServiceAdapter(display));
```

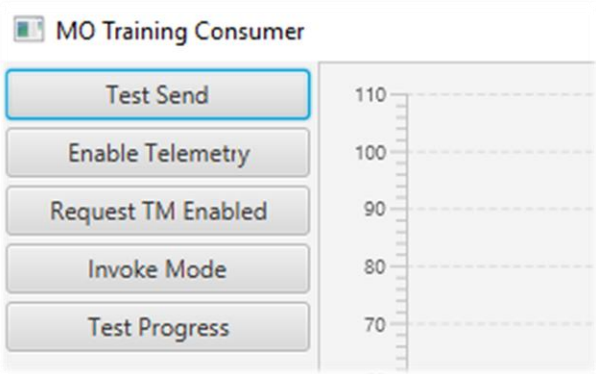
- Both require a new callback class called MySimSpacecraftServiceAdapter:
  - Create this class that extends SimSpacecraftAdapter
    - Pass in the display object from the consumer class so that it can display log messages on the MMI
- Override the following methods:

```
invokeModeAckReceived
invokeModeResponseReceived
testProgressAckReceived
testProgressUpdateReceived
testProgressResponseReceived
```

  - In each one just display a log message on the MMI

Stage 3: Running the applications

- To test this stage, first start the provider application as before:
  - From either the IDE, from Windows Explorer, or a command prompt
- This should start up, create the provider, and then wait for a key press
  - Entering a ‘q’ will shut the provider down
- Once the provider has started you can run the consumer as before:
  - From either the IDE, from Windows Explorer, or a command prompt
- The JavaFX MMI should start
  - There are two more buttons
    - one for each of the extra operations
  - Pressing them should cause the appropriate operation to be called on the service provider



- If this is not working you can compare your code to that in the Stage 4 project

Stage 4: Implementing PUB-SUB

- Open Stage 4 project
  - Build the project as this will trigger the auto-generation from the XML
- In this stage we implement the Publish Subscribe pattern

Stage 4 : Continuing the service pt1

- Open the provider MyProvider file
  - We are going to modify how we create our provider
  - We need to inform the MAL that our provider is now a publisher

- It is the final argument to MALProviderManager.createProvider
  - We don’t use that method directly though
  - We use the BaseMalServer.createProvider helper method
    - The helper method calls MALProviderManager.createProvider
    - The final argument of the helper method is passed to the provider manager call

- In the subInit method
  - Change the final argument of the createProvider call to true:

```
createProvider(SimSpacecraftHelper.SIMSPACECRAFT_SERVICE, serviceImpl, true);
```

Stage 4 : Continuing the service pt2

- Open the provider MySimSpacecraftServiceImpl file
  - We are going to implement the "monitorHousekeepingTM" operation
- There are no extra, auto-generated, methods to populate
  - The PubSub pattern does not generate any methods in the provider
  - The provider must use the auto-generated PubSub classes to publish updates
- Our service implementation has gained two extra methods
  - An "init" method which creates a periodically executing task
    - This is what we will use to publish our TM update every second
  - A "close" method which is used to shut down the periodically executing task
- The extra methods have been added for you as they are not specific to MO
    - They are just one way of adding a cyclic task to generate the TM

Stage 4 : Continuing the service pt3

- First we need to create our publisher object
  - In the "init" method replace the ToDo comment with:

```
publisher = createMonitorHousekeepingTMPublisher(domain, network,
            SessionType.LIVE, new Identifier("LIVE"),
            QoSLevel.BESTEFFORT, null, new UInteger(0));

EntityKeyList lst = new EntityKeyList();
lst.add(new EntityKey(new Identifier("*"), 0L, 0L, 0L));
publisher.register(lst, new MonitorHousekeepingTMPublishInteractionListener());
```

  - The first part creates the publisher and the second part registers it with the broker
    - You will also need to create a private member variable to hold the publisher

- We now need to create the callback class given in the PubSub register call:

```
private static final class MonitorHousekeepingTMPublishInteractionListener
                           implements MALPublishInteractionListener
```

  - This can be created anywhere, but an inner class is as good as anywhere
  - Use the IDE to fix the missing imports and auto generate the missing methods
    - Replace the default contents with a simple System.out

Stage 4 : Continuing the service pt4

- Now we need to actually implement our publish task
  - In the "run" method of the GenerateHousekeepingTmTask class, replace the ToDo with:

```
try
{
  UpdateHeaderList hdrLst = new UpdateHeaderList();
  BasicTelemetryList lst = new BasicTelemetryList();

  hdrLst.add(new UpdateHeader(new Time(System.currentTimeMillis()),
                              new URI("SomeURI"), UpdateType.UPDATE,
                              new EntityKey(new Identifier("TM"), null, null, null)));
  lst.add(generateTmReport(true));

  publisher.publish(hdrLst, lst);
}
catch (IllegalArgumentException | MALException | MALInteractionException ex)
{
  ex.printStackTrace();
}
```

  - This creates the required update header and body and then publishes it

- Our service implementation is complete for this stage

Stage 4 : Continuing the consumer pt1

- Open the consumer MyConsumer file
  - There are three 'ToDo' comments that need to be replaced
- The first one is for registering for the PubSub operation:

```
// set up the wildcard subscription
EntityKeyList entityKeys = new EntityKeyList();
entityKeys.add(new EntityKey(new Identifier("*"), 0L, 0L, 0L));

// match all domains, including sub-domains, of our domain
IdentifierList subDomain = new IdentifierList();
subDomain.add(new Identifier("*"));

// match all subdomains, but of the specific service and operation, for wildcard
entity key
EntityRequestList entities = new EntityRequestList();
entities.add(new EntityRequest(subDomain, false, false, false, false, entityKeys));

// register for the housekeeping TM reports
simSpacecraftStub.monitorHousekeepingTMRegister(new Subscription(new Identifier("SUB"),
                                                                 entities), adapter);
```

  - Note that the consumer callback adapter has been moved into a member variable

Stage 4 : Continuing the consumer pt2

- The second one is for de-registering for the PubSub operation:

```
IdentifierList subLst = new IdentifierList();
subLst.add(new Identifier("SUB"));

// deregister from the housekeeping TM service
simSpacecraftStub.monitorHousekeepingTMDeregister(subLst);
```

- The third one is for de-registering for the PubSub operation in the close method:

```
performPSderegisterInteraction();
```

Stage 4 : Continuing the consumer pt3

- The PubSub operation uses the callback adapter to pass the update to the consumer
  - We need to update our service adapter to process those new incoming updates

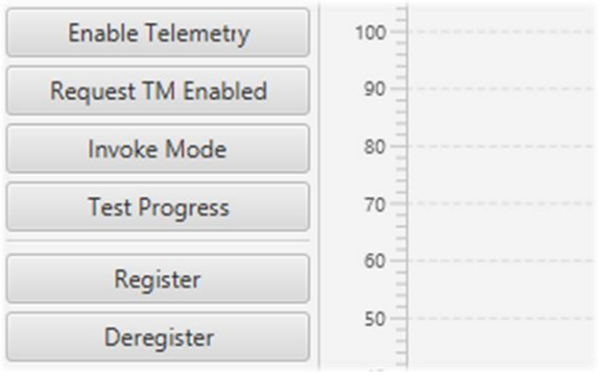
- Open the consumer MySimSpacecraftServiceAdapter file
  - Override the "monitorHousekeepingTMNotifyReceived" method
  - Replace the default contents with:

```
for (BasicTelemetry updateValue : basicTelemetryList)
{
  display.appendTmValue(updateValue);
}
```

  - This displays the TM update on the MMI

Stage 4: Running the applications

- To test this stage, first start the provider application as before:
  - From either the IDE, from Windows Explorer, or a command prompt
- This should start up, create the provider, and then wait for a key press
  - Entering a ‘q’ will shut the provider down
- Once the provider has started you can run the consumer as before:
  - From either the IDE, from Windows Explorer, or a command prompt
- The JavaFX MMI should start
  - There are two more buttons
    - one for registering and deregistering from the TM
  - Don’t forget that TM will not be generated until the enabledTM operation has been called passing TRUE



- If this is not working you can compare your code to that in the Stage 5 project

Stage 5: Adding COM Activity Tracking

- Open Stage 5 project
  - Build the project as this will trigger the auto-generation from the XML
- In this stage we implement the COM Activity Tracking for the INVOKE operation
- The Consumer will register for the COM Event service
  - The Activity Tracking service uses the COM Event service to publish its tracking events
  - Therefore the Consumer should filter for these events
- The Provider will need to implement the COM Event service
  - It will use this to publish the Activity Tracking events

Stage 5 : Continuing the service pt1

- Open the provider MyProvider file
  - We are going to add in an extra service to the provider
  - The COM Event service is a separate service from our SimSpacecraft service
  - We can do this manually however the ESA OSS library has a helper for this
- Change the base class of our MyProvider class to BaseComServer:

```
public class MyProvider extends BaseComServer
```

  - This class is held in another ESA OSS library, the Maven dependency has already been added for you
- We must also change slightly our subInitHelpers and subInit methods to call super
  - Change the ToDo comments to call the relevant super class methods
  - The BaseComServer uses this to initialise the COM service and area helpers
- A wrapper class is held in our new base class that helps with publishing Activity Tracking events
  - The new base class also creates an instance of the COM Event service for us to use
  - The Activity Tracking member variable needs to be passed to our SimSpacecraft service implementation for it to use

Stage 5 : Continuing the service pt2

- Open the provider MySimSpacecraftServiceImpl file
  - We will only implement Activity Tracking for the "invokeMode" operation for simplicity
  - We are going to pass the ActivityTrackingPublisher object via the constructor
  - The service implementation will use this to publish its COM Activity Tracking events
- First, create a constructor passing in the ActivityTrackingPublisher object:

```
public MySimSpacecraftServiceImpl(ActivityTrackingPublisher activityService)
{
  this.activityService = activityService;
}
```

  - You will need to create a member variable to hold the object
  - You will also need to update the MyProvider class to pass it into the constructor
- Next we will add in the calls to publish the events in the "invokeMode" operation
  - Change the five ToDo comments to call the relevant publishXXX method of the ActivityTrackingPublisher object
  - There is a single method for generating the acceptance event for an operation
  - There are specific methods for generating INVOKE execution events
  - The ToDo comments should help indicate what method to use
- Our service implementation is now complete

Stage 5 : Continuing the consumer pt1

- Open the consumer MyConsumer file
  - There are four 'ToDo' comments that need to be replaced
- The first one is for initialising the COM services and area:

```
COMHelper.deepInit(MALContextFactory.getElementFactoryRegistry());
```

- The second one is for creating the consumer stub for the COM Event service:

```
eventServiceStub = new EventStub(consumerMgr.createConsumer((String)null,
            new URI(tpuri), new URI(tburi),
            EventHelper.EVENT_SERVICE, new Blob("".getBytes()),
            domain, new Identifier("GROUND"),
            SessionType.LIVE, new Identifier("LIVE"),
            QoSLevel.BESTEFFORT, System.getProperties(), new UInteger(0)));
```

  - You will need to create a member variable to hold this for later use

Stage 5 : Continuing the consumer pt2

- The third ToDo is for registering for the COM PubSub operation:

```
// set up the subscription just for Activity Tracking events
EntityKeyList entityKeys = new EntityKeyList();
entityKeys.add(new EntityKey(new Identifier("*"),
            ComStructureHelper.generateSubKey(COMHelper._COM_AREA_NUMBER,
                    ActivityTrackingHelper._ACTIVITYTRACKING_SERVICE_NUMBER,
                    COMHelper._COM_AREA_VERSION, 0), 0L, 0L));

// match all domains, including sub-domains, of our domain
IdentifierList subDomain = new IdentifierList();
subDomain.add(new Identifier("*"));

// match all subdomains, but of the specific service and operation, for our entity key
EntityRequestList entities = new EntityRequestList();
entities.add(new EntityRequest(subDomain, false, false, false, false, entityKeys));

// register for COM events
eventServiceStub.monitorEventRegister(new Subscription(new Identifier("SUB"), entities),
                                                    new MyEventServiceAdapter(display));
```

  - Note that the subscription is not for everything
    - We have refined the subscription to only match Activity Tracking events
    - This reduces the traffic to us and makes out callback adapter simpler

Stage 5 : Continuing the consumer pt3

- The fourth one is for de-registering for the COM PubSub operation:

```
IdentifierList subLst = new IdentifierList();
subLst.add(new Identifier("SUB"));

// deregister from the COM event service
eventServiceStub.monitorEventDeregister(subLst);
```

- The COM event subscription requires a new callback class
  - Create a class MyEventServiceAdapter and extend EventAdapter:

```
public class MyEventServiceAdapter extends EventAdapter
{
  private final MyConsumerDisplayController display;

  public MyEventServiceAdapter(final MyConsumerDisplayController display)
  {
    this.display = display;
  }
```

    - Pass in the display object from the consumer class so that it can display log messages on the MMI

Stage 5 : Continuing the consumer pt4

- Override the monitorEventNotifyReceived method
  - Replace the default contents with:

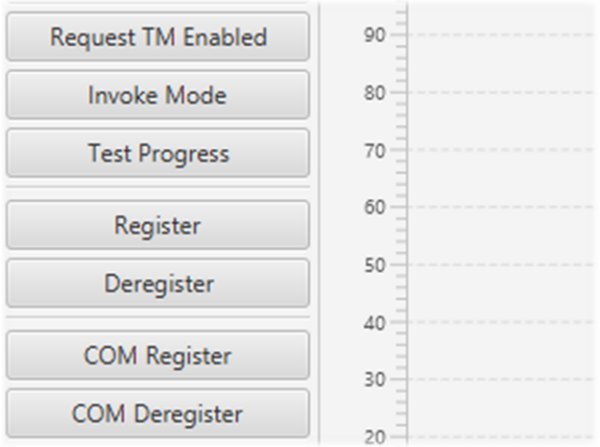
```
for (int i = 0; i < _UpdateHeaderList1.size(); i++)
{
  Identifier eventType = _UpdateHeaderList1.get(i).getKey().getFirstSubKey();
  Identifier acpt = new Identifier(ActivityTrackingPublisher.OBJ_NO_ASE_ACCEPTANCE_STR);

  // check to see what type of activity tracking event it is
  if (eventType.equals(acpt))
  {
    display.appendLogValue("Activity Tracking ACCEPTANCE Event with result "
                                    + _ElementList3.get(i).toString());
  }
  else
  {
    display.appendLogValue("Activity Tracking EXECUTION Event with result "
                                    + _ElementList3.get(i).toString());
  }
}
```

  - This displays the COM events on the MMI

Stage 5: Running the applications

- To test this stage, first start the provider application as before:
  - From either the IDE, from Windows Explorer, or a command prompt
- This should start up, create the provider, and then wait for a key press
  - Entering a 'q' will shut the provider down
- Once the provider has started you can run the consumer as before:
  - From either the IDE, from Windows Explorer, or a command prompt
- The JavaFX MMI should start
  - There are two more buttons
    - one for registering and deregistering for COM events
  - Don't forget Activity Tracking events will only be generated by the invokeMode operation
- If this is not working you can compare your code to that in the Stage 6 (final) project
  - Stage 6 (final) is only there for comparison purposes
- The training applications are now complete!



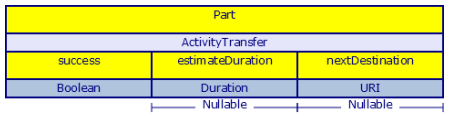
Part 5

- ESA OSS
  ESA GitHub and Apache Maven
  ESA Public Licence

ESA Open Source library

- Auto generator
  - Java
  - Docx
  - SVG
- MO Standards as XML/APIs
  - Blue Book specifications MAL/COM
  - Red Book draft specifications Common/M&C
- MAL Implementation
- Transport adaptors
  - Generic
  - RMI
  - TCP/IP
  - JMS
- Encoders
  - String
  - Binary
- Support libraries
  - MAL
  - COM
- Demo and example applications

HTML PUS style documentation



Others still Work In Progress…
FILE
HTTP
SPP (coming soon)

Fixed Length
Variable Length
Split

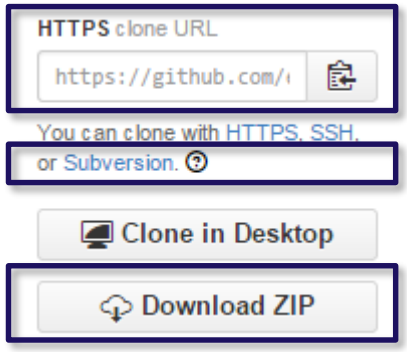
Developed during service prototyping – more to come

Where is it?

- Located in GitHub:
  - http://github.com/esa
- Split into several repositories:
  - CCSDS_MO — Documentation/Wiki
  - CCSDS_MO_POM — Top level Maven POM
  - CCSDS_MO_XML — MO Spec XML
  - CCSDS_MO_StubGenerator — API Stub Auto Generator
  - CCSDS_MO_APIS — Pre-built APIs
  - CCSDS_MO_MAL_IMPL — Java MAL Implementation
  - CCSDS_MO_TRANS — Transports/Encoders
  - CCSDS_MO_APPS — Demo/Example applications
  - CCSDS_MO_SUPPORT_LIBS — Support Libraries
  - CCSDS_MO_TESTBEDS — MO Spec Testbeds
    - Contains code under different licences from CNES/DLR
- Each repository may hold several components

Getting the code and building

- GitHub offers a few ways to get the code:
  - Clone using Git version control
    - Either HTTPS (Recommended) or SSH
  - Checkout using Subversion
  - Download a Zip file



- GitHub provides us with:
  - Visual browsing of the code
  - Issue reporting
  - Very easy but powerful Git version management
    - Simple to "fork and merge" using Git

- Build using Apache Maven
  - More on this in the next slide
  - You can use other build systems
    - But you may need to create your own build file in this case

The wonderful world of Maven (and friends)

- Why do we use Maven?

- Reason 1: Much superior build system than Ant
  - Uses convention rather than configuration
    - Only specify what is different rather than everything!
  - Plugin system – ESA MO Stub generator is a Maven plugin
  - Dependency system
    - Extremely easy component build system
    - Maven Central….

- Reason 2: Maven Central
  - Cloud based hosting of pre-built components
    - ESA MO in Maven Central
  - Can be accessed from multiple build systems:
    - Maven (obviously)
    - Apache Buildr
    - Apache Ant
    - Groovy Grape
    - Gradle
    - Grails
    - Leiningen
    - SBT
    - …

Licence and Contributor's Agreement

Firstly "I Am Not A Lawyer"!

Disclaimer: This is only a summary. No information is legal advice.

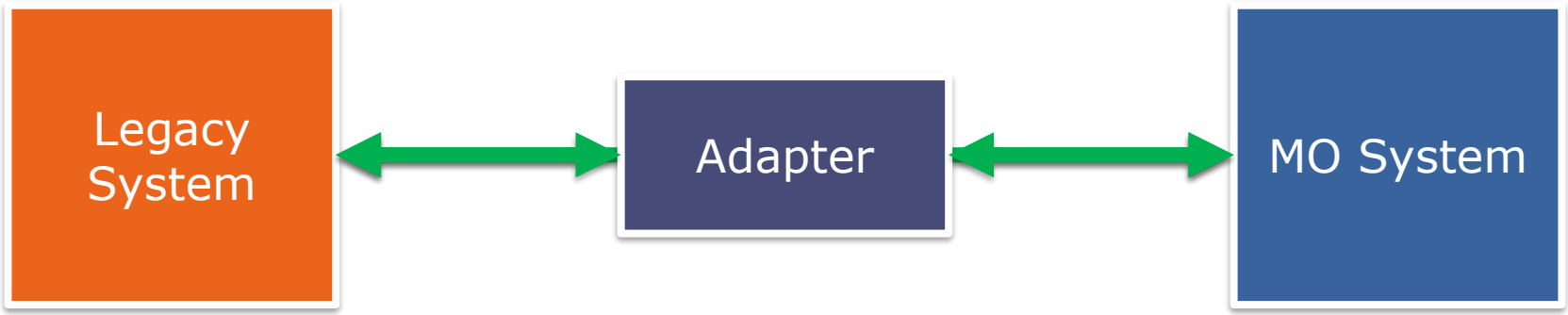
- Licenced under ESA Public License version 2.0 weak copyleft
  - Basically it is similar to a hybrid of LGPL and Apache 2
  - Permits Commercial and Non-Commercial use
  - You can Distribute, sublicense, lend and rent the Software
- There is broadly no copyleft infection of derivative works
  - However, check the details if you are concerned
    - See section 3 (specifically section 3.2) of the license
- Any submission back to ESA of your wonderful additions/fixes/corrections/etc will need you to sign a Contributor's Agreement
  - Gives ESA rights to use your work
  - IPR remains with you
  - … does not exist yet … ☹
  - We have a stop gap solution though so don't hold back! ☺

Part 6

- Integrating with legacy systems

Integrating with legacy systems

- The basic approach when looking to integrate MO with legacy systems:

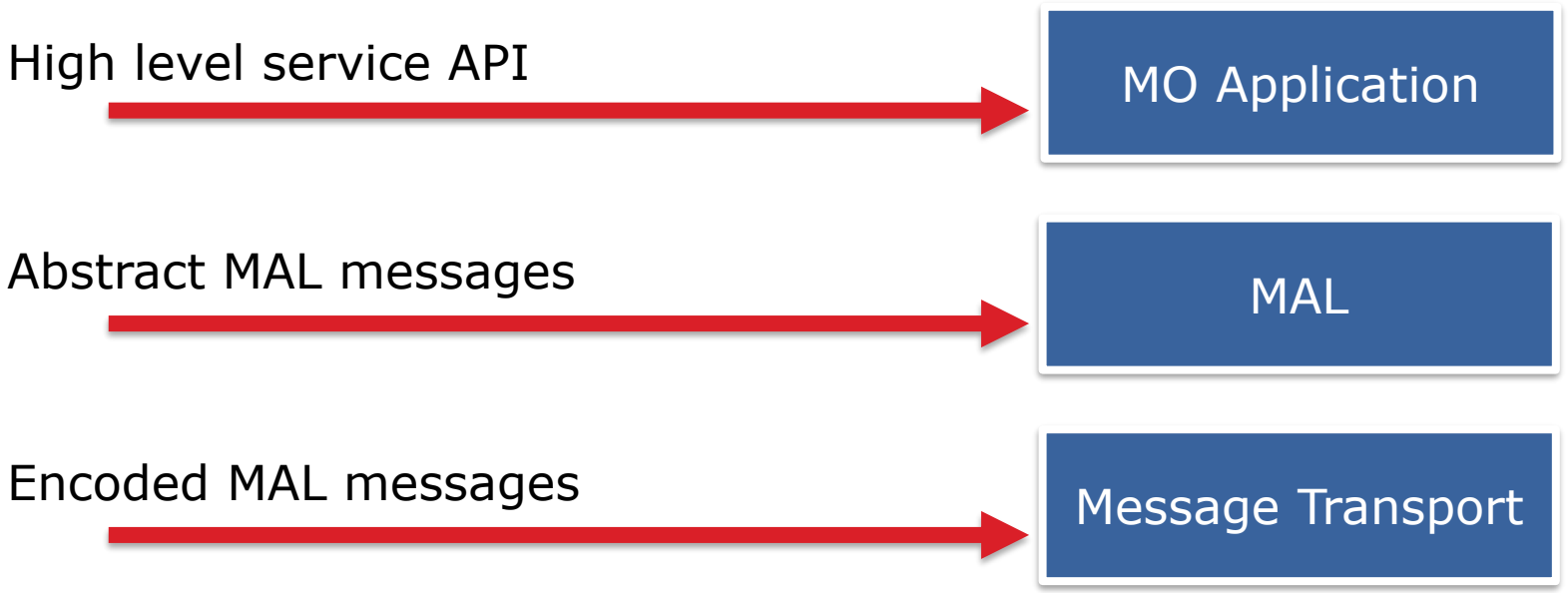


  - is the concept of an adapter

- What the Adapter is responsible for is completely dependent on the capabilities of the legacy system

Integrating with legacy systems

- There are many levels at which integration can occur:

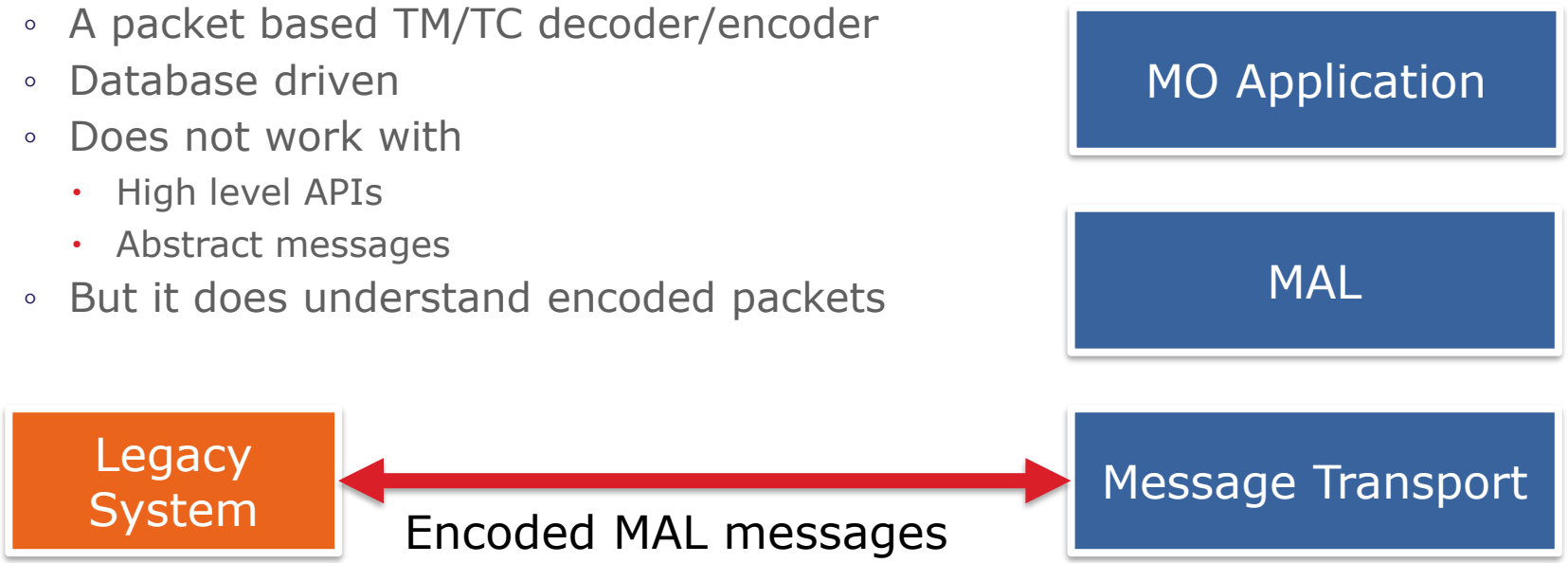


- What level you choose depends on your legacy system

Integrating with ESA SCOS 2000

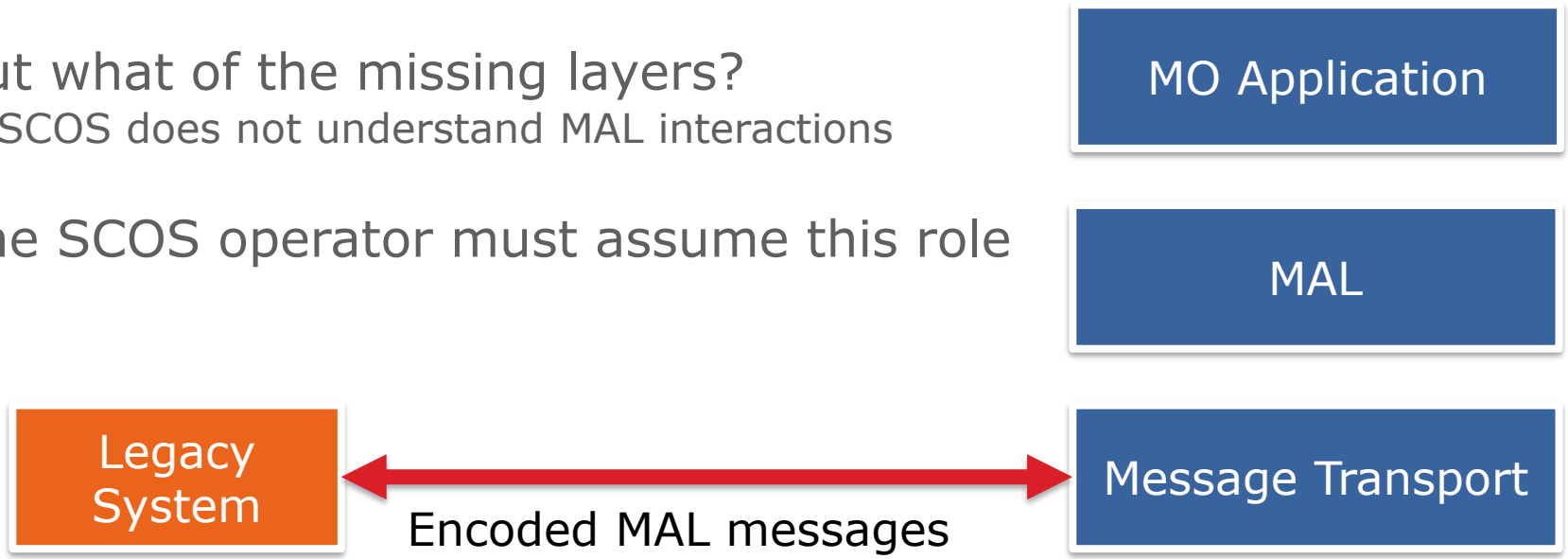
- For an example we will look briefly at ESA's SCOS 2000 system

- SCOS 2000 is:
  - A packet based TM/TC decoder/encoder
  - Database driven
  - Does not work with
    - High level APIs
    - Abstract messages
  - But it does understand encoded packets



Integrating with ESA SCOS 2000

- For SCOS to be able to encode/decode MO messages
  - The MO XML must be converted into the database format of SCOS
    - The MO XML is abstract
    - We must choose an encoding to use to talk to SCOS
      - Fixed length binary with CCSDS Space Packet Protocol works
  - We need a database containing
    - the definitions of the MO messages in the chosen encoding
- But what of the missing layers?
  - SCOS does not understand MAL interactions
- The SCOS operator must assume this role



Integrating with ESA SCOS 2000

- The SCOS operator must represent the consumer MAL
  - This sounds weird
  - But its not as weird as you might think
  - For an MO consumer this just means
    - Populating the messages correctly
    - Building up the sequences of encoded messages
    - Matching response messages to the correct request
- This is just normal SCOS operator responsibilities
- So, this should be possible for SCOS
  - It has already been demonstrated
  - ESA's OPS-SAT mission has demonstrated using SCOS to talk to an MO spacecraft
    - In fact OPS-SAT (an MO spacecraft) will fly using SCOS 2000

Summary

Summary

- This is only scratching the surface
  - Look at the ESA OSS code for what else can be done
  - More is being added all the time
- Use the ESA OSS and please contribute!
  - Bug reports
  - Enhancements
  - Ideas of what to look at next
  - Even just an email to say that you are use it
  - Anything is always gratefully received
- Other agencies are creating MO software too
  - CNES have their own line of OSS for MO
  - NASA are developing a C++ API
- There are many paths to follow from here
  - Some are listed on the final slide

Links

General

CCSDS Website

http://www.ccsds.org/

MO in Wikipedia

http://en.wikipedia.org/wiki/CCSDS_Mission_Operations

Open Source Software

ESA on GitHub

https://github.com/esa

ESA MO OSS Wiki

https://github.com/esa/CCSDS_MO/wiki