SCA Java Architecture Guide

Unable to render {include} The included page could not be found.

Unable to render {include} The included page could not be found.

Apache Tuscany SCA Java Architecture Guide

Overview

The SCA Java runtime is composed of core and extensions. The core is essentially a multi-VM wiring engine that connects components together using the principles of Dependency Injection, or Inversion of Control.

Core

The Core is designed to be simple and limited in its capabilities. It wires functional units together and provides SPIs that extensions can interact with. Capabilities such as service discovery, reliability, support for transport protocols, etc. are provided through extensions.

Extension

Extensions enhance SCA runtime functionality. Extension types are not fixed and core is designed to be as flexible as possible by providing an open-ended extension model. However, there are a number of known extension types defined including:

- **Component implementation types**, e.g. Spring, Groovy, and JavaScript
- **Binding types**, e.g. Axis, CXF, AMQP, ActiveMQ, JXTA
- **DataBinding types**, e.g. JAXB, SDO, XmlBeans
- **Interface Binding types**, e.g. WSDL, Java

Details of how to implement an extension can be can be found in the Extensions Guide.

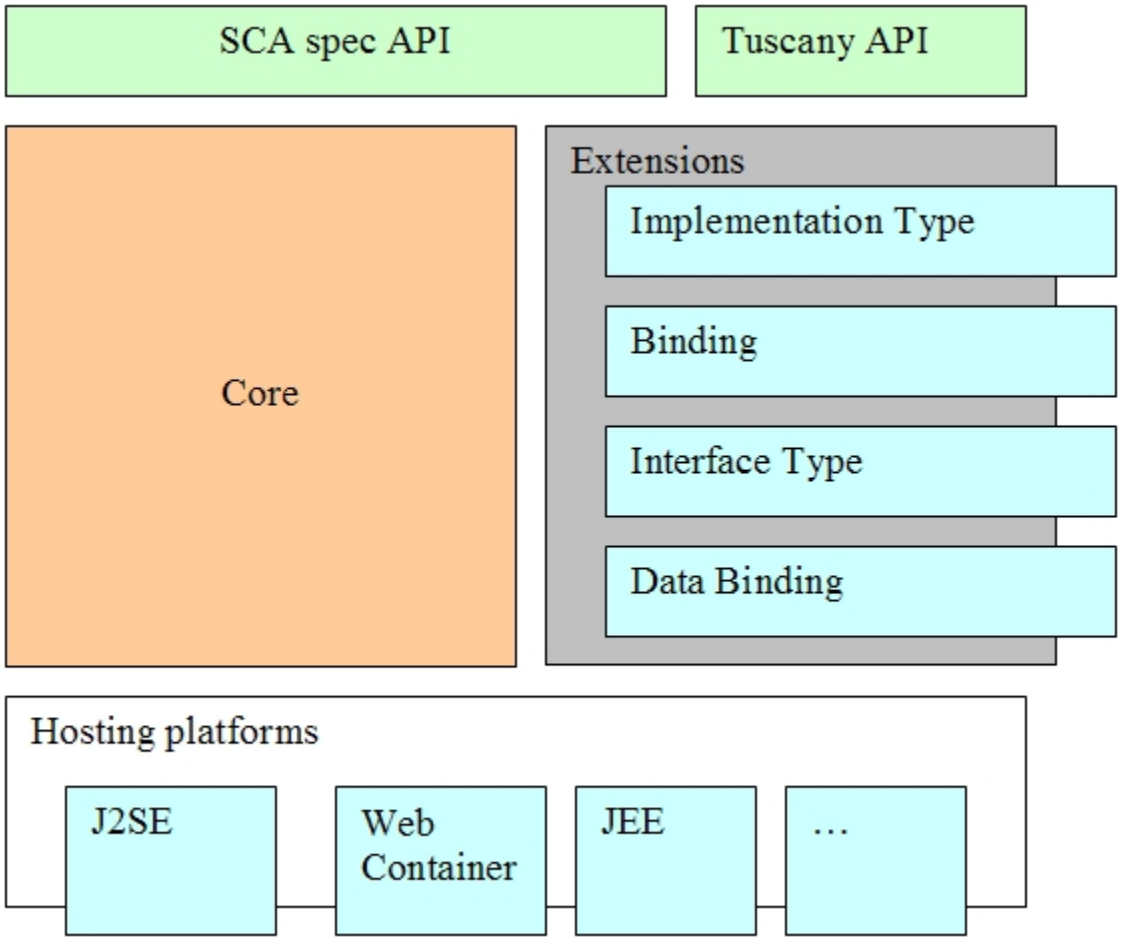
Runtime

The core is designed to be embedded in, or provisioned to, a number of different host environments. For example, the core may be provisioned to an OSGi container, a standalone runtime, a servlet engine, or J2EE application server. Runtime capabilities may vary based on the host environment.

High level overview of Java SCA runtime

The diagram shown below is a high level view of the SCA runtime which consists of the following key modules/packages:

1. SCA Spec API: The APIs defined by the SCA Java Client and Implementation Spec
2. API: Tuscany APIs which extend the SCA Spec APIs
3. Core: The runtime implementation and the SPIs to extend it
4. Extensions:
   a. Container implementation - For extending language support, for example BPEL, Python, C++, Ruby,..
   b. Binding - for extending protocol support, for example Axis2, CXF,..
   c. Interface Binding - for extending types of service definition, for example WSDL, Java, ...
   d. Databinding - for extending data support, for example SDO, JAXB, ...
5. Host Platforms: The environments that host the Tuscany runtime.



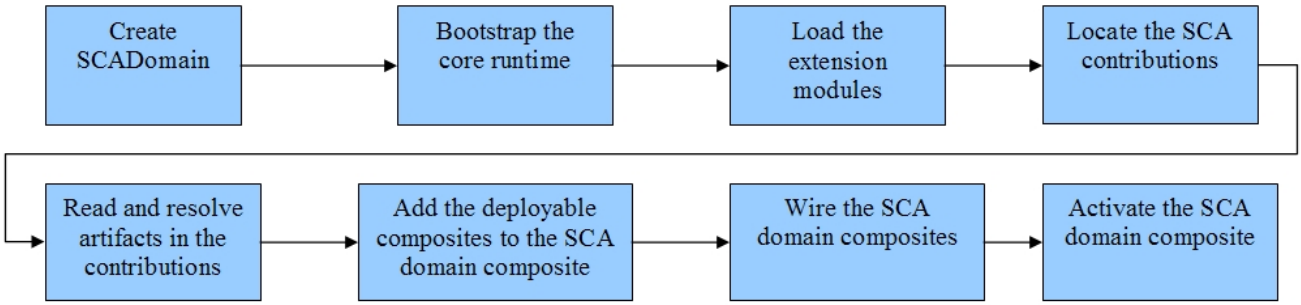
Internals High Level View

[Note: Do we want to link to Kernel-Structure]?

Bootsrap

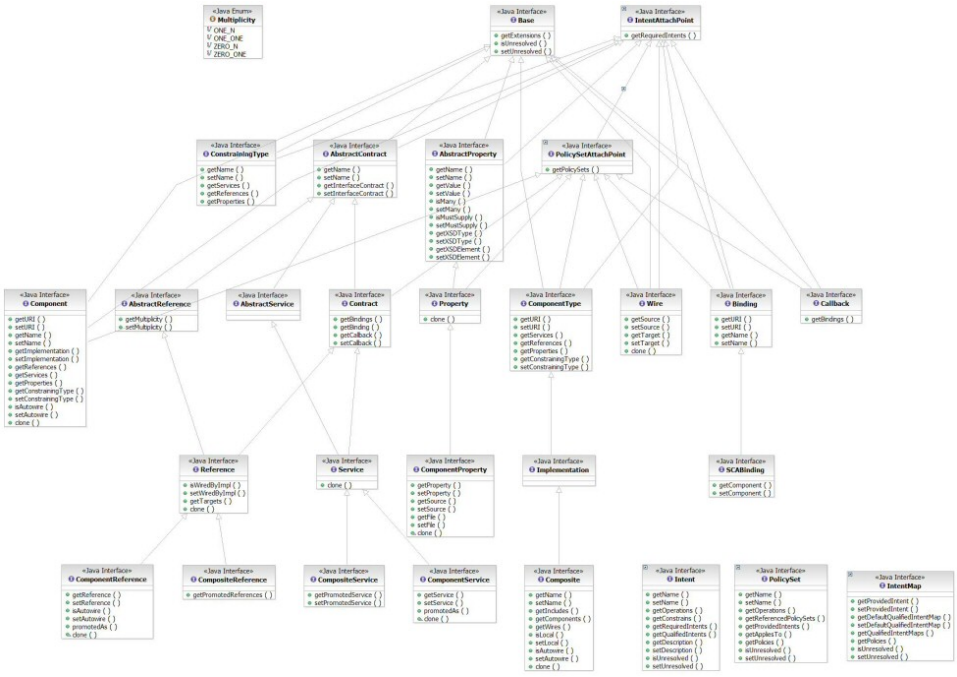
Bootstrap process is controlled by Host environment. The default process is implemented in DefaultBootstrapper.
The runtime processes service assemblies serialized using SCA Assembly XML which can take other forms.

- The load phase processes SCDL and creates an in-memory model and produces corresponding runtime artifacts (e.g. components, services, references)
- The connect/wire phase wires references to services



Assembly Model

The SCA assembly model is represented as a set of interfaces in Tuscany. The following are some key elements.

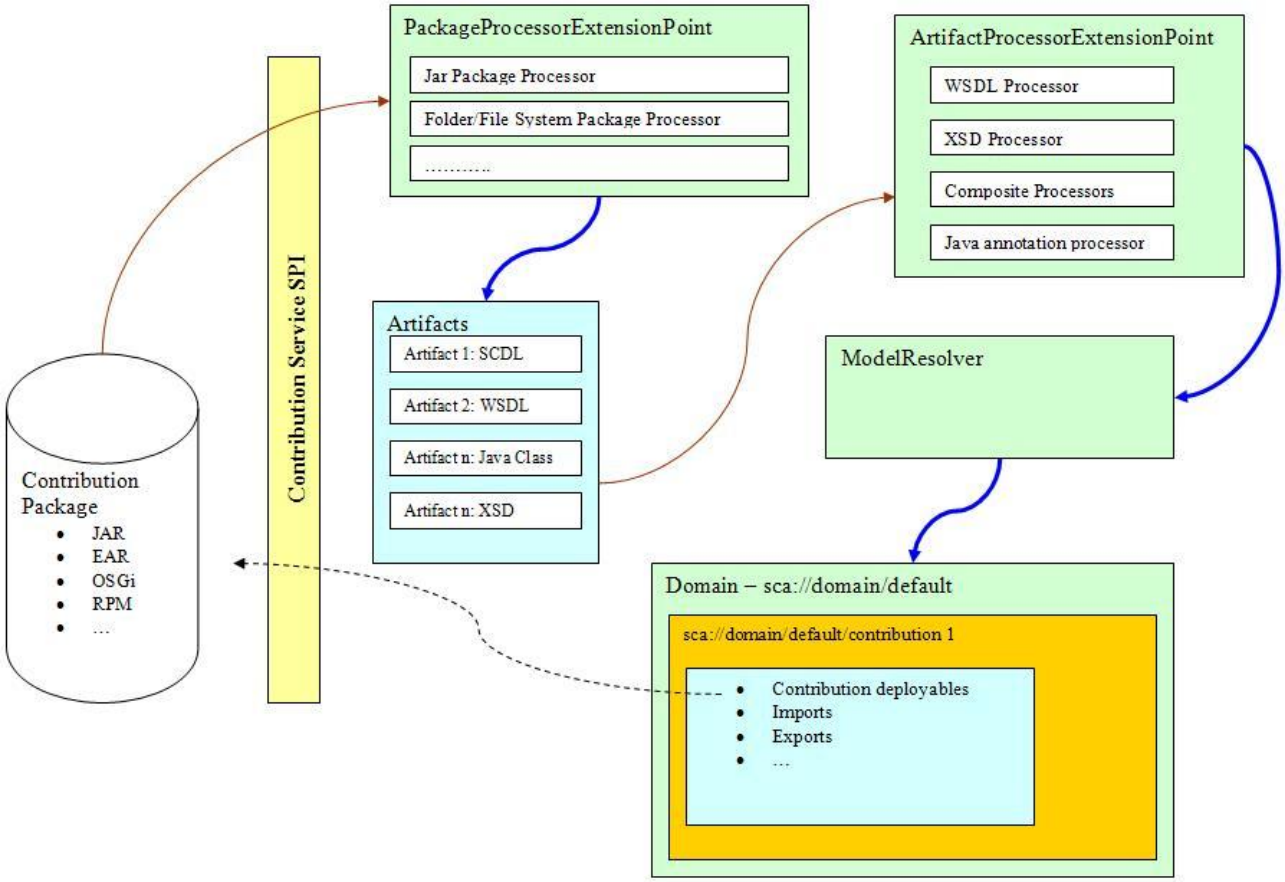
- SCA **components** are configured instances of SCA implementations, which provide and consume services.
- SCA **services** are used to declare the externally accessible services of an implementation.
- SCA **references** represent a dependency that an implementation has on a service that is supplied by some other implementation, where the service to be used is specified through configuration.
- An **implementation** is concept that is used to describe a piece of software technology such as a Java class, BPEL process, XSLT transform, or C++ class that is used to implement one or more services in a service-oriented application. An SCA composite is also an implementation.
- **ComponentType** refers to the configurable aspects of an implementation.
- **Interfaces** define one or more business functions. These business functions are provided by Services and are used by components through References. Services are defined by the Interface they implement. SCA currently supports two interface type systems:
    - Java interfaces
    - WSDL portTypes
- An SCA **composite** is the basic unit of composition within an SCA Domain. An ***SCA Composite*** is an assembly of Components, Services, References, and the Wires that interconnect them.
- *SCA **wires*** connect ***service references*** to ***services***.
- ***Bindings*** are used by services and references. References use bindings to describe the access mechanism used to call the service to which they are wired. Services use bindings to describe the access mechanism(s) that clients should use to call the service.
- **Properties** allow for the configuration of an implementation with externally set data values. The data value is provided through a Component, possibly sourced from the property of a containing composite.
- 

Contribution

The Tuscany runtime provides a framework to support SCA contributions. The framework can be extended against the following two extension points:

PackageProcessorExtensionPoint: It accepts extensions that can handle different packaging format/archives such as a directory, a JAR, an OSGi bundle, an EAR, a WAR and a ZIP.

ArtifactProcessorExtensionPoint: It accepts extensions that can handle specific types of artifacts such as WSDL, XSD, composite, java class, BPEL.



- Package processors scans the contribution being installed and generate a list of artifacts that needs to be processed. Currently there is support for folder/file system and jar contribution packages. In order to be available to the contribution service, a package processors needs to register itself with the package processor extension.
- Artifact processors are used to process each artifact available on the contribution. In order to be available to the contribution service, a artifact processor needs to register itself with the artifact processor extension. An artifact processor will be called for each artifact in two phases :
-
  - read phase : This is where you read an artifact (a document, an XML element, classes etc.), populate a model representing the artifact and return it. The SCA contribution service calls ArtifactProcessor.read() on all artifacts that have an ArtifactProcessor registered for them. If your model points to other models, instead of trying to load these other models right away, you should just keep the information representing that reference, which you'll turn into a pointer to the referenced model in the resolve() phase. Note that you don't necessarily need to fully read and populate your model at this point, you can choose to complete it later.
-
  - resolve phase : This phase gives you the opportunity to resolve references to other models (a WSDL, a class, another composite, a componentType). At this point, all models representing the artifacts in your SCA contribution have been read, registered in the contribution's ArtifactResolver, and are ready to be resolved.
- All deployable composites should be now ready to get deployed to the SCA Domain

Implementation Extension

Implementation extension is responsible for supporting an implementation type, such as Java, Script, and BPEL

Binding Extension

Binding extension is responsible for supporting a binding type such as Web Service, JMS, JSON-RPC and RMI

Interface Extension

Interface extension is responsible for supporting an interface type such as Java interface and WSDL 1.1 portType

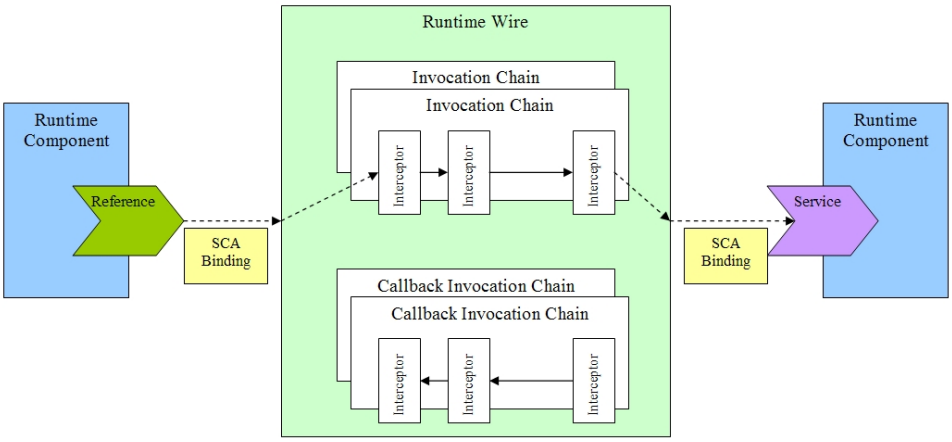
Data Binding Extension

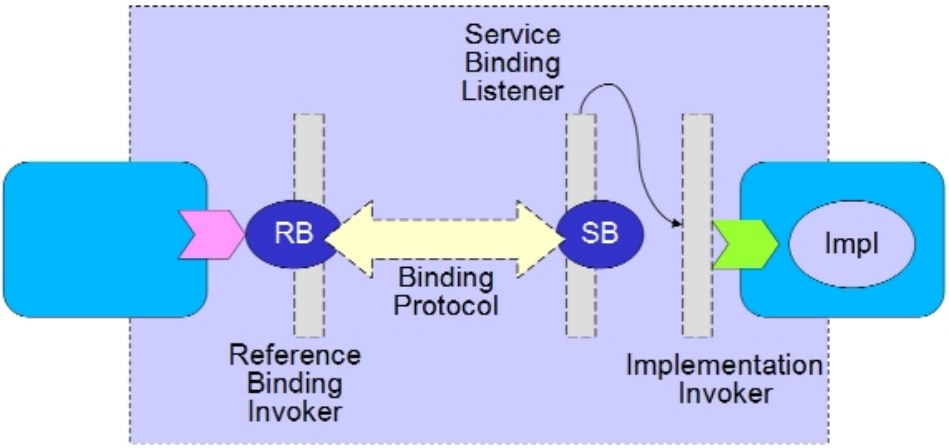
Please refer to: Tuscany Databinding Guide

Composite Activation

After a composite is fully configured, it can be activated in the SCA domain. The Tuscany runtime activates a composite in the following steps:

1. **Build the composite**: In this phase, the composite model is further normalized to facilitate the runtime interactions. The metadata is consolidated following the service/reference promotions. With the flattened model, we can get all the information at component level.
2. **Configure the composite**: In this phase, the composite hierarchy is navigated to configure the component implementation, reference binding and service binding with provider factories which will be used in later steps to create runtime wires among components and external services.
3. **Create the runtime wires**: In this phase, runtime wires are created for component references and component services over selected bindings. The runtime wire is a collection of invocation chains that are patitioned by operations. Each invocation chain consists of a set of invokers and interceptors s. Invokers provides the invocation logic to binding protocols and implementation technologies. Interceptors are speical kind of invokers that provide additional logic for the invocation, such as data transformation and transaction control. For a component reference, we create runtime wires to represent the outbound invocation over the selected binding. For a component service, we create runtime wires to represent the inbound invocation to the implementation type. Callback wires can be attached to component services to represent the callback invocations from the service.
4. **Start the composite**: In this phase, the start() callback methods defined by the ImplementationProvider, ReferenceBindingProvider and ServiceBindingProvider will be invoked. As a result, components, component references and component services are initialized to serve the component interactions. Service listeners are started to accept inbound requests from the binding layer.





Invocation Overview

- An invocation is dispatched to the WireInvocationHandler
- The WireInvocationHandler looks up the correct InvocationChain
- It then creates a message, sets the payload, sets the TargetInvoker, and passes the message down the chain
- When the message reaches the end of the chain, the TargetInvoker is called, which in turn is responsible for dispatching to the target
- Having the TargetInvoker stored on the outbound side allows it to cache the target instance when the wire source has a scope of equal or lesser value than the target (e.g. request-->composite

The runtime provides components with InboundWires and OutboundWires. InvocationChains are held in component wires and are therefore stateless which allows for dynamic behavior such as introduction of new interceptors or re-wiring.

Loading SCA assemblies

Artifact Processor

Artifact processors are responsible for processing each artifact available in a contribution package, these processors are going to be invoked in two phases

- read phase : This is where you read an artifact (a document, an XML element, classes etc.), populate a model representing the artifact and return it. The SCA contribution service calls ArtifactProcessor.read() on all artifacts that have an ArtifactProcessor registered for them. If your model points to other models, instead of trying to load these other models right away, you should just keep the information representing that reference, which you'll turn into a pointer to the referenced model in the resolve() phase. Note that you don't necessarily need to fully read and populate your model at this point, you can choose to complete it later.

- resolve phase : This phase gives you the opportunity to resolve references to other models (a WSDL, a class, another composite, a componentType). At this point, all models representing the artifacts in your SCA contribution have been read, registered in the contribution's ArtifactResolver, and are ready to be resolved.

Loading Java SCA

SCA service assemblies are deployed to the SCA domain in the format of SCDL files. Tuscany runtime artifact processor loads the SCDLs into model objects which are a set of java beans to hold the metadata.

There are two types of loaders:

- StAXElementLoader: Load the XML element from the StAX events
- ComponentTypeLoader: Load the Component Type for an implementation either by introspection or paring a side file

Loading Component Type

Loads the component type definition for a specific implementation

- How it does this is implementation-specific
- May load XML sidefile (location set by implementation)
- May introspect an implementation artifact (e.g. Java annotations)
- ... or anything else

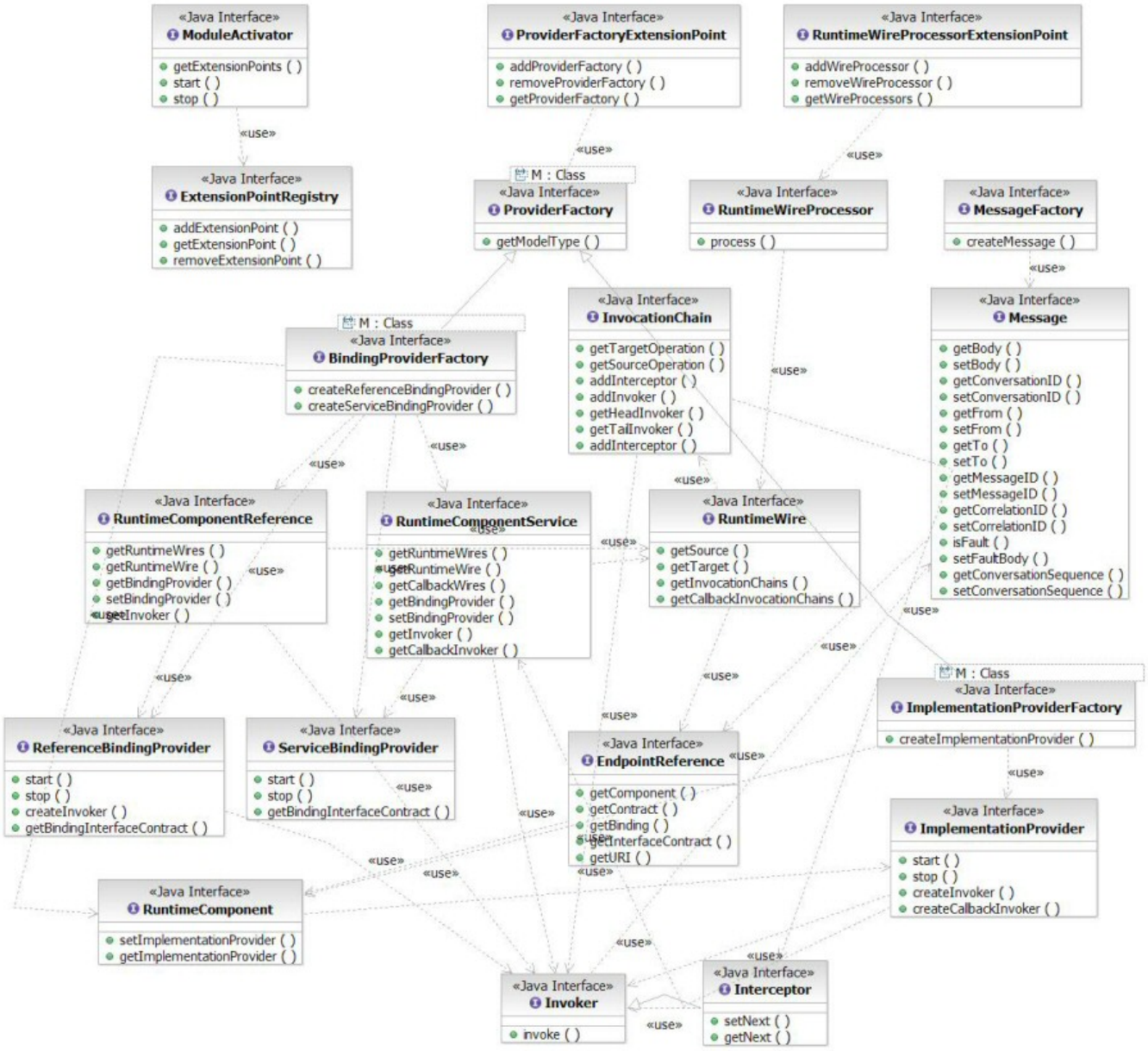
Loading Composite ComponentType Loader

- Load SCDL from supplied URL
- Extract and load SCDL from composite package

POJO ComponentType Loader

- Introspect Java annotations
- Uses a pluggable "annotation processing" framework to introspect Java classes

Class diagram for the runtime artifacts



Spring Integration