

**ISO 21090 DATA TYPE**

**Design Document**

Version: 0.1

Last Modified: 8/31/2010

Team : caCORE SDK

Client : National Cancer Institute -   
 Center for Bioinformatics and Information Technology,

National Institutes of Health,

US Department of Health and Human Services

Document History

Revision History

| **Version Number** | **Revision Date** | **Author** | **Summary of Changes** |
| --- | --- | --- | --- |
| 0.1 | 08/31/2010 | Satish Patel | Initial Draft |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Review

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **Team/Role** | **Version** | **Date Reviewed** | **Reviewer Comments** |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Related Documents

More information can be found in the following related documents:

|  |
| --- |
| **Document Name** |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

Table of Contents

1. Background 6

1.1 General 6

1.2 Current State of Service Development Tooling 6

1.2.1 Service Modes 6

1.2.2 ISO 21090 Enabled Service Development Tooling 7

1.3 Problem statement 8

2. Goal and Objectives 9

3. Project Description and Requirements 9

3.1 Requirements 9

4. Tooling Implementation Governance 12

5. Technical Approach 13

5.1 ISO 21090 Library 13

5.2 ISO 21090 Enabled Service 14

5.3 ISO 21090 Enabled Service Development Tool 15

5.3.1 Graphical User Interface (GUI) 15

5.3.2 Code Generator 16

5.3.1 Mapping and Transformation 16

5.3.2 Semantic Metadata Management 17

5.3.1 Deployment 18

6. Technical Design - Service 19

6.1 Annotations 20

6.2 Interceptors 22

6.3 Error Handling 26

6.4 Data bindings 27

7. Technical Design – Service Generation Process 29

7.1 Implementation Principles 31

7.2 Service Assembler 31

7.3 Service Synchronizer 32

8. Technical Design – Service Development Tooling 34

8.1 Graphical User Interface 34

8.2 Code Generator 40

8.2.1 Model Representation 40

8.2.2 Challenges 43

8.2.3 Code Generator 45

8.3 Mapping and Transformation 54

8.4 Semantic Metadata Management 54

8.5 Deployment 54

Table of Figures

[Figure 1: Data Service Architecture 7](#_Toc271102182)

[Figure 2: Analytical Service Architecture 7](#_Toc271102183)

[Figure 3: ISO 21090 Tooling Governance Structure 12](#_Toc271102184)

[Figure 4: Service Development Tool Architecture 15](#_Toc271102185)

[Figure 5: ISO 21090 Enabled Service Components 17](#_Toc271102186)

[Figure 6: CXF Architecture 19](#_Toc271102187)

[Figure 7: Service Interceptors 23](#_Toc271102188)

[Figure 8: Interceptors sequence diagram 24](#_Toc271102189)

[Figure 9: Service Generation Process 29](#_Toc271102190)

[Figure 10: Service Implementation Principles 31](#_Toc271102191)

[Figure 11: Service Assembler 32](#_Toc271102192)

[Figure 12: Service Synchronizer 33](#_Toc271102193)

[Figure 13: Model Conversion Interface 35](#_Toc271102194)

[Figure 14: Model Explorer Interface 36](#_Toc271102195)

[Figure 15: Model Property Browser Interface 37](#_Toc271102196)

[Figure 16: Model Relationship Browser Interface 37](#_Toc271102197)

[Figure 17: Model Operations Browser Interface 38](#_Toc271102198)

[Figure 18: Data Model Browser Interface 38](#_Toc271102199)

[Figure 19: Model Properties Browser Interface 39](#_Toc271102200)

[Figure 20: Generator Interface 39](#_Toc271102201)

[Figure 21: An intermediate form capable of representing the fundamental parts of modeling can serve as a universal translator for other meta-models. 42](#_Toc271102202)

[Figure 22: Ecore as a universal intermediate form for meta model translation. 43](#_Toc271102203)

[Figure 23: Sample development workflow involving model translation and artifact generation. 44](#_Toc271102204)

[Figure 24: High level candidate architecture for the new SDK. 45](#_Toc271102205)

[Figure 25: Template approach 46](#_Toc271102206)

[Figure 26: Model Validation Approach 47](#_Toc271102207)

[Figure 27: Code Generator Framework Architecture 48](#_Toc271102208)

[Figure 28: Code Generator Component Class Diagram 50](#_Toc271102209)

[Figure 29: Code Generator Workflow 51](#_Toc271102210)

[Figure 30: JSR 223 Based Scripting Architecture 53](#_Toc271102211)

# Background

## General

One goal of caBIG® is to provide interoperability between systems. The focus of that interoperability has been facilitating information sharing between data systems. The infrastructure tools and APIs developed by the National Cancer Institute Center for Bioinformatics and Information Technology (NCI CBIIT) provide the building blocks for development of interoperable information management systems. As the Cancer Biomedical Informatics Grid (caBIG®) community continues to expand along with evolution of technologies, standards and policies – we are examining the need for the infrastructure tools and processes to enable enterprise-grade and standards based syntactically and semantically interoperable applications in a federated collaborative environment.

The ISO 21090 standard datatypes is the central building block to create semantically interoperable services. The standard enables the services to carry additional information and metadata about the domain elements. In order to lower the barriers to entry for the users and increase the adoption of the ISO 21090 standard, we need to examine the needs for effective tooling.

## Current State of Service Development Tooling

The NCI CBIIT and caBIG provides several tools which are used in specific combination to develop the required service capabilities. In the last release of the software bundle, we had provided support for the NCI localization of ISO 21090 in these tools. Given below is the list of primary tools in the infrastructure portfolio responsible for service development

* NCI localization ISO 21090 – Common reusable library with NCI localizations
* caCORE SDK – Code generator and caCORE-like middleware system
* caCORE Workbench – Manages the entire code generation process
* caAdapter – Model mapping capability
* Introduce – Grid service authoring tool
* Semantic Integration Workbench (SIW) – Semantic metadata annotation tool

### Service Modes

Current form of the infrastructure tools allows service authoring in one of the two modes.

**Data Service**

A data sharing service allows query of the data source in a structured fashion. The data service tooling is the most extensive tooling that we have in the caBIG to create a service. The tools are capable of creating the entire service from the UML model without writing single line of code. The tooling takes a UML model with the class diagrams and tables as an input, and through a guided process, the user creates a data service with many options including search capabilities, security, and various interfaces to access the data etc.



Figure : Data Service Architecture

The above diagram shows the components of the data services that are generated using the infrastructure tooling. The primary advantage of this tooling is that it allows users to quickly build a service from a UML model and thereby lowering the barriers to entry in the caBIG world. The data service provides a unified interface to access the data which further enables the end consumers to navigate through a large number of data sources.

**Analytical Service**

Analytical service is any service that is not a data service. The tooling allows the service developers to create the service skeleton and other service layer functions right out of the box using the service creation wizard. The service developer is responsible for supplementing the service with the business logic to complete the implementation.



Figure : Analytical Service Architecture

The above diagram shows the components of the analytical services that are generated using the infrastructure tooling. The primary advantage of this tooling is that it allows users to quickly build a service from an existing API and thereby reducing the requirements of managing and developing the service wrapper.

### ISO 21090 Enabled Service Development Tooling

The service development tools were capable of recognizing the domain structure elements in the form of primitive datatypes. In order to support the ISO 21090 datatypes in all of the tools mentioned above (with exception of SIW), several major enhancements were implemented. After the enhancements, the service development tools recognized the NCI localization as valid datatypes and allowed users to create domain specific analytical and data services with ISO 21090 datatypes at a great ease. The specific implemented enhancements are:

* NCI localization of ISO 21090 library
* ISO 21090 based domain model to the database mapping capability
* Code generation capability to generate persistence layer and related artifacts
* Query support for the ISO 21090 datatypes based domain models
* Service authoring tool that recognizes the ISO 21090 datatypes

## 

## Problem statement

The ISO 21090 implementation in the current generation of the service development tool provides an easier way to create the analytical and data service. The enhancements implemented in the tools as part of the last release were limited as the existing tools were not built on the ISO 21090 foundation. We believe that there are additional functionalities that are needed in the tooling. These capabilities are:

* Complete implementation of ISO 21090 library
* Integrated service development platform to create ISO 21090 enabled services
* Integration with the semantic infrastructure
* Query support for the ISO 21090 based data structure
* Service deployment capability
* Support for the non-grid services and non-java services

# Goal and Objectives

The long-term goal of the service development tooling is to provide support for automated creation and deployment of services using ISO 21090 data types compliant with SAIF and ECCF to ease adoption of ISO 21090 data types. To meet this long-term goal, following short-term goals are identified:

**Complete ISO 21090 Implementation:** We need to provide a full implementation of the ISO 21090 datatype library. As the complete implementation of the ISO 21090 is made available to the users, we need to provide a migration path for users of the NCI localization to upgrade to the complete implementation of the ISO 21090 library.

**Integrated Service Development Tooling:** A service development tool is needed that is capable of supporting the complete implementation of the ISO 21090 data types. The service development tool will have features that will enable the developers to create and deploy the services.

**Service Migration:** Currently, we have many users who have developed the services with the existing tools and localized ISO 21090 data type implementation. We need to provide a clear migration path to bring these users to the new infrastructure tools.

# Project Description and Requirements

The project involves developing effective service development tooling which allows creation of the ISO 21090 data type enabled services in the user friendly fashion. The tooling is targeted toward the system developers and architects who are primarily involved in the implementation of the services.

## Requirements

* Complete implementation of ISO 21090 library – The ISO 21090 standard contains 110 datatypes. The standard specifies the structure of the datatypes as well as conformance requirements of the datatypes. A complete implementation of the ISO 21090 datatype library should to be developed which supports both.
* Migrate current implementation to the full ISO 21090 library – NCI CBIIT currently has localized version of the ISO 21090 datatypes. As the complete datatype library is developed, the existing users’ implementation should be migrated to the complete version.
* Integrated service development platform – The service development tool should be an integrated platform where all the capabilities of the service development are available as modules. Users should be able to create the complete service by using a “guided” process which assists the users in using the capabilities through these modules.
* Model Driven service development tooling – The UML model should be used to represent the structure of the information. The service development tooling should generate the artifacts from these UML model. The artifacts include functional code, documentation, test cases, reports etc. The same model will be used to load the semantic metadata in the semantic infrastructure repository.
* Model mapping capabilities – Model mapping capability refers to the ability to map different model structures with each other. There are many different ways the user can create the models for the information structures which includes following
  + Primitive based domain models
  + ISO 21090 data types based domain model
  + Data model
  + XML schema
* Domain Converter – The service will contain information in various data structure in different layer of the system. The service development tool should provide appropriate APIs to convert the information from one format to other. E.g information in the ISO 21090 based domain model to primitives based domain model.
* Query support for the ISO 21090 based data structure – The ISO 21090 datatype is a complex data structure. In case of the data service, the service should provide capability to query the data from the data source based on the ISO 21090 data structure.
* Service deployment capabilities – The developer of the service should be able to deploy the created service in different environment including local machines, remote machines and cloud environments. The service development tool should generate appropriate scripts, configuration files, and tools to fulfill the requirement.
* Support for the non-grid services and non-java services – The service development tool architecture should support the extension points which can be used to build support for the non-grid and non-java services.
* Support for generating the client – The service consumer should be able to create the client application from the service end point and WSDL. This will allow the user to rapidly develop the client application in different programming languages.
* Integration with the semantic infrastructure – The semantic infrastructure provides access to the metadata that the service is supposed to use. At the time of service development, the tool should provide a way for annotating the models with metadata from the semantic metadata registry as well as with the custom metadata supplied by the user. The annotated model should be used to synchronize information with the semantic metadata registry.
* Security and Audit trail implementation – The service generated with the service development tool should have dedicated support for the security and audit trail management through corresponding services. The service developer should be able to plug-in custom authorization implementation to implement appropriate levels of security in the service.

# Tooling Implementation Governance



Figure : ISO 21090 Tooling Governance Structure

The ISO 21090 tooling development will be done by the development team based on the input received from the user community. The user community will provide feedback to the ISO 21090 governance team and the ISO 21090 product management team (which includes SAIC-F COTR and product manager). The ISO 21090 governance team will work with the users and provide change in the ISO 21090 policy to the product management team. The product management team will drive the implementation based on the policy and the features input received. This user driven tool development process will enhance the usability of the product and yield in greater adoption with better user experience.

# Technical Approach

Based on the identified requirements and use cases, we believe that the ISO 21090 tooling should be developed in four major areas: 1) ISO 21090 library, 2) Service development tool, 3) Service. The details of these areas are provided below.

## ISO 21090 Library

ISO 21090 library is the critical piece of the service development environment and the service. The library provides the foundation block upon which other blocks of the service development tool operates. Currently the NCI has developed a localized version of the ISO 21090 library. This specific implementation provides sufficient capabilities to the domain specific applications to leverage the ISO 21090 features. As the current implementation is a constrained version of the full implementation, it restricts the users from accessing the full capabilities of the ISO 21090 standard. In the next round of development of service development tooling, it is imperative that the users are provided with a complete library of the ISO 21090 datatypes.

There are several options available to provide a complete library of ISO 21090 data types to the users. The options include 1) using off-shelf library, 2) extend the NCI localization library, or 3) build new library from scratch. Each of these options presents its advantages and disadvantages.

|  |  |  |
| --- | --- | --- |
|  | **Advantages** | **Disadvantages** |
| **Off-shelf library (e.g. OHT, Tolven, JavaSIG etc)** | 1. Minimize effort in understanding the ISO 21090 standard 2. Minimize the development effort and time | 1. Any extensions to the library cannot be built easily 2. We cannot change the implementation to suit our environment 3. Existing users needs to be migrated to newer version |
| **Extend NCI localization** | 1. Provides us complete control over the implementation 2. We can implement extensions as needed | 1. We need to spend time and effort to understand ISO 21090 standard 2. Increases overall development effort and time |
| **Build from scratch** | 1. Provides us complete control over the implementation 2. We can implement extensions as needed 3. We can take the new requirements into consideration which were omitted in the NCI localization implementation | 1. We need to spend time and effort to understand ISO 21090 standard 2. Increases overall development effort and time 3. Existing users needs to be migrated to newer version |

As part of the implementation, we will evaluate all of these options with inputs from the standard experts and the domain specific teams. Once the analysis is complete and approved by the management, we will implement the approved option.

## ISO 21090 Enabled Service

The service is the finished output of the service development tool. The service provides a specific functionality that the end user or service developer intended to build with the tool. Currently there are two types of services being supported by the infrastructure tools: 1) Analytical service and 2) Data service.

Both the types of mentioned services and in general other services share many common characteristics. Each service needs the service representation using a specific technology, message processing logic, security, audit trails, query support, deployment capability etc. As we start supporting the ISO 21090 datatypes, these common components of the services will need to recognize and provide support for the same. For example, we currently have CQL as a query language in which we can specify the query based on the domain objects using the primitive data types. This language was enhanced to provide support for the ISO 21090 datatypes using specialized operators and specialized operations. Similarly, the fine grained authorization implementation would require us to recognize attributes of the ISO 21090 datatypes so that we can construct security policy based on that.

We will need to provide following modules for the service to be functional; 1) the service skeleton, 2) common components like security and audit integration points 3) the code generated from the model and 4) data transformation functions and 5) querying capabilities. With exception of the artifacts generated from the model, all modules will be pre-developed. The service development tool will generate the code from the model and combine other modules to produce a complete working system. If the service developer provides custom code then that will be integrated with the finished service as well.

The PST team will determine the architecture and the messaging standard for the service. We believe that regardless of the architecture style and technology, we will need to build the mentioned service functionalities. For example, the service style could be SOAP based or RESTful style. The technology of choice could be Axis or CXF. We will be able to align our implementation with the target service environment in a relatively easy manner because the service development tool produces the artifacts, and these artifacts can be combined into any service.

## ISO 21090 Enabled Service Development Tool



Figure : Service Development Tool Architecture

Service development tool is the infrastructure component which allows the users to build the ISO 21090 compliant services. It provides all the functionalities necessary to meet needs of a wide array of users. The requirements identified in the earlier section provide us the overview of basic needs to adopt ISO 21090 datatypes. Based on these requirements, we believe that an integrated environment is needed. Given below are the major areas of the service development tool components:

### Graphical User Interface (GUI)

The GUI is the critical component of the service development tool. The user of the tool interacts with the functionality using the GUI. Using the GUI, the user can control the service development life cycle. Given below are some of the specific functions.

1. Create new service and manage service settings
2. Manage code generation settings
3. Manage the semantic metadata
4. Map the model components with other models
5. Drive the deployment scenarios
6. Synchronize the code with the model

As evident by the commercial software (like windows, unix, and mac), the effective GUI can determine how the software will be utilized by the user. Poor user interfaces leads to the frustration from the user. We will pay very careful attention to the development of this component and make sure that the service developers are taken into the consideration while implementing the GUI controls.

### Code Generator

The core of the service development tool is a Model Driven Architecture (MDA) oriented component. This component provides following features

* Code generation capability from UML models to support consistent language neutral representation of the system components and rapid development of the services
* Code synchronization capability with the UML models to support iterative development

The core engine of the service development tool will allow the users to build various types of artifacts. When needed, the user will be able to develop and plug-in additional generators to produce the artifacts that are not supported out of the box by our service development tool. The examples of the artifacts are Java beans, Object relational mapping files, XSDs, WSDL files, configuration files, ECCF templates etc. The artifacts produced by the core engine will be used to build and support service layers.

### Mapping and Transformation

The ISO 21090 datatypes introduces a complex problem of managing the data in a specific format with its constraints. From the user’s perspective, the ISO 21090 datatypes can be implemented in more than one ways. Given below are some of the possible options

1. Implement ISO 21090 datatypes in the domain model and map that to relational database. Convert this domain model in the XML to exchange information with the clients.
2. Implement the ISO 21090 datatypes in the domain model and map that to the domain model based on the primitives. Convert the domain model based on the ISO 21090 datatypes to XML to communicate with the clients
3. Implement the domain model with the primitives and convert it in the XML. Using a separate utility, map the XML into the ISO 21090 XML structures.
4. Implement the domain model in primitives and convert it directly in the ISO 21090 XML format.



Figure : ISO 21090 Enabled Service Components

The diagram above shows all possible combinations in which the information can be converted into ISO 21090 datatype representation format. Current implementation of the NCI localization and tooling supports the option 1 and option 2. The tooling provides end to end support for the option 1 while the user is required to implement a large amount of code to use option 2. The transformation functions can be used outside the service as well. For example, the client may need to convert the ISO 21090 message into their domain model so that they do not have to change their business logic. As we move in the direction of providing comprehensive tooling, we need to provide support for multiple options for the users to choose from.

The mapping of the models requires very sophisticated GUI development. Using a robust interface, the user will be able to load the model and map the model components with another model. The user will choose the appropriate options of mapping and transformation based on their needs. The mapping information is saved into a separate mapping configuration file. This file is then used by the runtime APIs to transform the data from one data structure to another data structure.

### Semantic Metadata Management

The semantic metadata provides the back bone to the interoperable systems. Major problems with the semantic metadata management are how the models are created and how the metadata is managed by the end user. When the users create the models at that time they do not annotate the models with the semantic metadata. As a result, when the users try to publish their service, they tend to have missing metadata information. To improve this process, we believe that the developers should be enabled with a user friendly way of annotating the model and the source code with the semantic metadata. We can achieve this function by providing following functionalities in the service development area

* Provide ability to retrieve semantic metadata for the new model elements from the semantic infrastructure repository
* Provide ability to create new metadata whenever the semantic metadata do not exist in the public repository and store this in the “local-metadata” repository
* Provide an ability to push the newly created from local metadata repository to semantic metadata to the public repository
* Get updates on the already used semantic metadata

Using these features, the developers will be able to annotate the models when they are in the process of creating and updating the models. In case of applications like “Dynamic Extensions”, the same process can be extended and the local-metadata can be hosted along with the service.

### Deployment

The developed service is going to be used by developers in various environments. At NCI CBIIT we have a system of DEV, QA, STAGE, and PROD tiers. At the same time, many teams are looking into using the cloud based deployment capabilities. As the service development tool produces the service, it can enable it with the deployment capabilities out of the box. Leveraging the BDA scripts is one of the available options. We can provide additional GUI on top of the scripts so that the users of the developed service can easily deploy these services in their environment using few simple clicks. For the everyday deployment on the target environment, we will provide an additional GUI area in the service development tool to deploy the service.

# Technical Design - Service

The Apache CXF framework helps to develop web services using standards based programming model and also provides a flexible deployment model for deploying web services. Following diagram show the architecture of CXF.

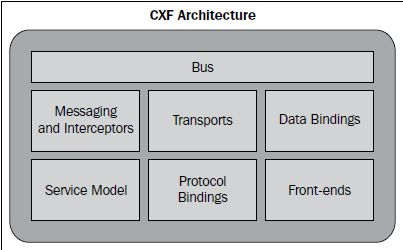


Figure : CXF Architecture

Busis the backbone of the CXF architecture. The CXF bus is comprised of a Spring-based configuration file, namely, cxf.xml which is loaded upon servlet initialization through SpringBusFactory. It defines a common context for all the endpoints. It wires all the runtime infrastructure components and provides a common application context.

CXF provides the concept of frontend modeling, which allows creating web services using different frontend APIs. The APIs let users create a web service using simple factory beans and JAX-WS implementation. It also lets users create dynamic web service clients. The primary frontend supported by CXF is JAX-WS.

JAX-WS is a specification that establishes the semantics to develop, publish, and consume web services. JAX-WS simplifies web service development. JAX-WS defines Java-based APIs that ease the development and deployment of web services. The specification supports WS-Basic Profile 1.1 that addresses web service interoperability. It effectively means a web service can be invoked or consumed by a client written in any language. JAX-WS also speeds up web service development by providing a library of annotations to turn Plain Old Java classes into web services and specifies a detailed mapping from a service defined in WSDL to the Java classes that will implement that service. Any complex types defined in WSDL are mapped into Java classes following the mapping defined by the JAXB specification.

CXF supports two approaches for web service development: Code-First and Contract-First. With the Code-first approach, it starts by developing a Java class and interface and annotating the same as a web service. The approach is particularly useful where Java implementations are already available and you need to expose implementations as services. With the contract-first approach, CXF provides tools to generate various service artifacts.

Service development framework will make use of open-source CXF and JAX-WS capabilities and tools to jump start with its requirements. CXF supports automated code generation of the following:

* Java to WSDL
* WSDL to Java
* XSD to WSDL
* WSDL to XML
* WSDL to SOAP
* WSDL to service

## Annotations

JAX-WS 2.0 uses several annotations defined by JSR-181. Annotations play a critical role in JAX-WS 2.0. First, annotations are used in mapping Java to WSDL and schema. Second, annotations are used a runtime to control how the JAX-WS runtime processes and responds to web service invocations.

Following is an example of annotating a java interface to be a service interface.

|  |
| --- |
| package com.nih.nci.sdk.demo;  @WebService  public interface HelloWorld {  String sayHi(@WebParam(name="text") String text);  } |

The @WebService annotation on the implementation class lets CXF know about the interface.

The @WebParam annotation is necessary as java interfaces do not store the Parameter name in the .class file.

Following example annotates implementation class as a service implementation.

|  |
| --- |
| package com.nih.nci.sdk.demo;  import javax.jws.WebService;  @WebService(endpointInterface = " com.nih.nci.sdk.demo.HelloWorld",  serviceName = "HelloWorld")  public class HelloWorldImpl implements HelloWorld {  public String sayHi(String text) {  System.out.println("sayHi called");  return "Hello " + text;  }  } |

Following example shows a way to publish the service.

|  |
| --- |
| HelloWorldImpl implementor = new HelloWorldImpl();  String address = "http://localhost:9000/helloWorld";  Endpoint.publish(address, implementor); |

Following example is another way to publish the service with more control over its behavior.

|  |
| --- |
| HelloWorldImpl implementor = new HelloWorldImpl();  JaxWsServerFactoryBean svrFactory = new JaxWsServerFactoryBean();  svrFactory.setServiceClass(HelloWorld.class);  svrFactory.setAddress("http://localhost:9000/helloWorld");  svrFactory.setServiceBean(implementor);  svrFactory.getInInterceptors().add(new LoggingInInterceptor());  svrFactory.getOutInterceptors().add(new LoggingOutInterceptor());  svrFactory.create(); |

Following example shows client code used to access the service.

|  |
| --- |
| public final class Client {  private static final QName SERVICE\_NAME  = new QName("http://server.nci.demo/", "HelloWorld");  private static final QName PORT\_NAME  = new QName("http://server.nci.demo/", "HelloWorldPort");  private Client() { }  public static void main(String args[]) throws Exception {  Service service = Service.create(SERVICE\_NAME);  // Endpoint Address  String endpointAddress = "http://localhost:9000/helloWorld";  // Add a port to the Service  service.addPort(PORT\_NAME, SOAPBinding.SOAP11HTTP\_BINDING, endpointAddress);    HelloWorld hw = service.getPort(HelloWorld.class);  System.out.println(hw.sayHi("World"));  }  } |

This is another client example that provides more control with interceptors.

|  |
| --- |
| JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean();  factory.getInInterceptors().add(new LoggingInInterceptor());  factory.getOutInterceptors().add(new LoggingOutInterceptor());  factory.setServiceClass(HelloWorld.class);  factory.setAddress("http://localhost:9000/helloWorld");  HelloWorld client = (HelloWorld) factory.create();  String reply = client.sayHi("HI");  System.out.println("Server said: " + reply);  System.exit(0); |

All the examples shown above can be created by Service development framework automatically using CXF, JAX-WS tools and EMF templates.

## Interceptors

The interceptors are responsible for transforming messages between the raw data transported across the wire and the Java objects handled by the endpoint's implementation code. The interceptors are organized into phases to ensure that processing happens on the proper order. Apache CXF provides powerful runtime implementation to implement and execute interceptors. Following description is focused around CXF implementation. A similar implementation can be used in place of CXF via loosely coupled nature of this enterprise service framework. These interceptors can implement non-functional logic such as access control, logging, monitoring, etc. leaving only the core business functionality to the Web service developer

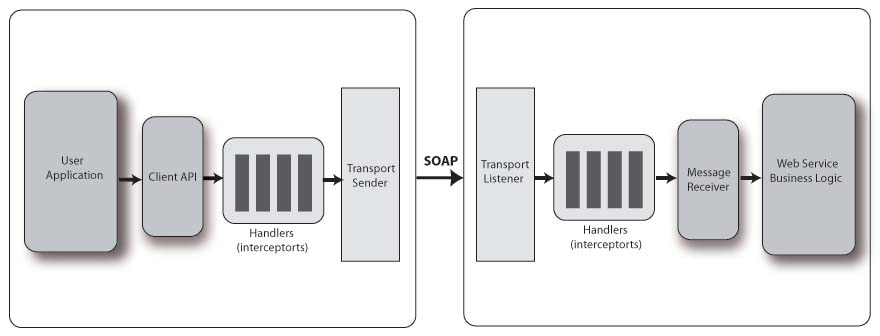


Figure : Service Interceptors

Interceptors are used with both CXF clients and CXF servers. When a CXF client invokes a CXF server, there is an outgoing interceptor chain for the client and an incoming chain for the server. When the server sends the response back to the client, there is an outgoing chain for the server and an incoming one for the client. Additionally, in the case of SOAPFaults, a CXF web service will create a separate outbound error handling chain and the client will create an inbound error handling chain.

Some examples of interceptors inside CXF include:

* SoapActionInterceptor - Processes the SOAPAction header and selects an operation if it's set.
* StaxInInterceptor - Creates a Stax XMLStreamReader from the transport input stream.
* Attachment(In/Out)Interceptor - Turns a multipart/related message into a series of attachments.

Following sequence diagram shows the interaction between different components invoking an interceptor.

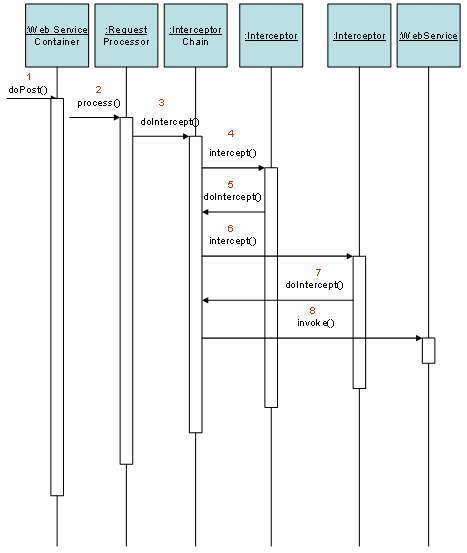


Figure : Interceptors sequence diagram

InterceptorChains are divided up into Phases. The phase that each interceptor runs in is declared in the interceptor's constructor. Each phase may contain many interceptors. On the incoming chains, you'll have the following phases:

On the incoming chains, following are the phases:

|  |  |
| --- | --- |
| **Phase** | **Functions** |
| RECEIVE | Transport level processing |
| (PRE/USER/POST)\_STREAM | Stream level processing/transformations |
| READ | This is where header reading typically occurs. |
| (PRE/USER/POST)\_PROTOCOL | Protocol processing, such as JAX-WS SOAP handlers |
| UNMARSHAL | Unmarshalling of the request |
| (PRE/USER/POST)\_LOGICAL | Processing of the umarshalled request |
| PRE\_INVOKE | Pre invocation actions |
| INVOKE | Invocation of the service |
| POST\_INVOKE | Invocation of the outgoing chain if there is one |

On the outgoing chain there are the following phases:

|  |  |
| --- | --- |
| **Phase** | **Functions** |
| SETUP | Any set up for the following phases |
| (PRE/USER/POST)\_LOGICAL | Processing of objects about to marshalled |
| PREPARE\_SEND | Opening of the connection |
| PRE\_STREAM |  |
| PRE\_PROTOCOL | Misc protocol actions. |
| WRITE | Writing of the protocol message, such as the SOAP Envelope. |
| MARSHAL | Marshalling of the objects |
| (USER/POST)\_PROTOCOL | Processing of the protocol message. |
| (USER/POST)\_STREAM | Processing of the byte level message |
| SEND | Final sending of message and closing of transport stream |

Following is an example of SOAP message interceptor that runs before/after certain other interceptors defined in the same phase.

|  |
| --- |
| public class MyInterceptor extends AbstractSoapInterceptor {  public MyInterceptor() {  super(Phase.USER\_PROTOCOL);  // MyInterceptor needs to run after SomeOtherInterceptor  getAfter().add(SomeOtherInterceptor.class.getName());  // MyInterceptor needs to run before YetAnotherInterceptor  getBefore().add(YetAnotherInterceptor.class.getName());  }  ...  } |

Interceptors can be added to a service either through annotations or CXF bus configuration. Following is an example of adding interceptors through annotations.

|  |
| --- |
| @org.apache.cxf.interceptor.InInterceptors (interceptors = {"com.example.Test1Interceptor" })  @org.apache.cxf.interceptor.InFaultInterceptors (interceptors = {"com.example.Test2Interceptor" })  @org.apache.cxf.interceptor.OutInterceptors (interceptors = {"com.example.Test1Interceptor" })  @org.apache.cxf.interceptor.InFaultInterceptors (interceptors = {"com.example.Test2Interceptor","com.example.Test3Intercetpor" })  @WebService(endpointInterface = "org.apache.cxf.javascript.fortest.SimpleDocLitBare",  targetNamespace = "uri:org.apache.cxf.javascript.fortest")  public class SayHiImplementation implements SayHi {  public long sayHi(long arg) {  return arg;  }  ...  } |

Service generation framework will read interceptor configuration from the model through tag values and generate annotated service interface code as shown above.

## Error Handling

An exception is any error condition or unexpected behavior encountered by an executing program. When there is an exception during the execution of the Web service, the Web service should not only capture the exceptions, but also communicate the exception back to the consumers of the Web service. Because Web services provide a platform-independent of way of leveraging a specific functionality, the exceptions that occur in the Web Services must also be communicated in a platform-independent manner. To accomplish this, SoapException class can be used that abstracts the complexities of the SOAP fault creation process. The SoapException class consists of the following properties that need to be populated before throwing the exception to the consumers.

* Message—Contents of the exception
* Code—Enum constant that specifies the type of Fault code (e.g. ClientFaultCode, ServerFaultCode)
* Actor—URL of the Web service method where the exception has occurred
* Detail—Detail element can be used to communicate more information about the exception to the callers

One can also register a custom CXF out fault interceptor which can handle all the exceptions by writing directly to the HttpServletResponse stream or XMLStreamWriter (as XMLFaultOutInterceptor does).

The WebFault annotation is used when mappingWSDL faults to Java exceptions, see section 2.5. It is used to capture the name of the fault element used when marshalling the JAXB type generated from the global element referenced by the WSDL fault message. It can also be used to customize the mapping of service specific exceptions to WSDL faults.

Following is an example of defining an exception:

|  |
| --- |
| package com.example.customerservice;  @WebFault(name="NoSuchCustomer")  @XmlAccessorType( XmlAccessType.FIELD )  public class NoSuchCustomerException extends RuntimeException {  /\*\*  \* We only define the fault details here. Additionally each fault has a message  \* that should not be defined separately  \*/  String customerName;  } |

Following is an example of using the exception:

|  |
| --- |
| package com.example.customerservice;  @WebService  public interface CustomerService {  public Customer[] getCustomersByName(@WebParam(name="name") String name) throws NoSuchCustomerException;  } |

## Data bindings

Data binding components are responsible for mapping between XML on the wire and Java objects. Each data binding implements a particular discipline for mapping, such as JAXB or XML Beans.

There are three parts to a data binding:

* Mapping the live data as it comes into and out of services.
* Providing XML schema based on Java objects for dynamic wsdl URLs and java2ws.
* Generating Java code from WSDL for wsdl2java (and, theoretically, dynamic clients).

All data bindings provide the live data mapping. Each data binding supports one or more formats for the data in transit. By default, CXF supports data binding by JAXB. The architecture of CXF supports pluggable nature of data binding framework to replace its default easily.

# Technical Design – Service Generation Process

Following diagram shows high level service generation process. A service generator is responsible to generate following service artifacts:

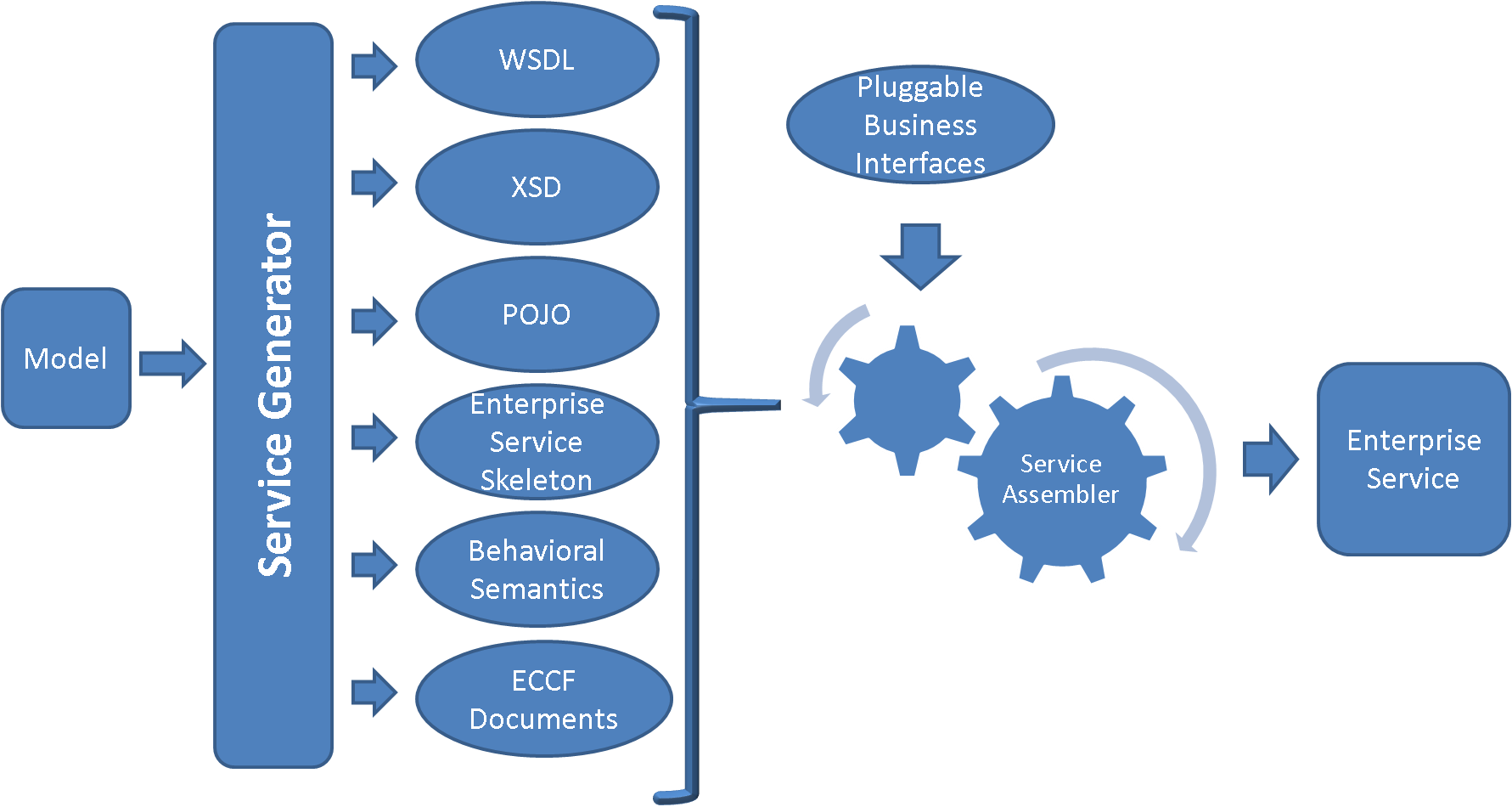


Figure : Service Generation Process

1. **WSDL:** It is used to describe the message syntax associated with the invocation and response of a web service. A WSDL description describes three fundamental properties of a web service:
   1. *What* a service does – The operations (methods) the service provides, and the data (arguments and returns) needed to invoke them
   2. *How* a service is accessed – Details of the data formats and protocols necessary to access the service operations
   3. *Where* the service is located – Details of the protocol-specific network address, such as a URL
2. **XSD:** It is used to describe schema of data structures used in a data exchange. This schema will be referred by the service to validate input and output types of operations.
3. **POJO:** These are the java beans generated confirming the format defined by the schema. These java beans built with standardized data types are exchanged between the interfacing services to enable interoperability.
4. **Enterprise service skeleton:** Service skeleton is the collection of necessary service artifacts to implement and deploy a service including:

* Ant processes for build, deploy, and test operations
* Standard interface for both client and service to implement
* Fully implemented client APIs
* Stub implemented service
* configuration to support service metadata and the registration of metadata and properties

1. **Behavioral semantics:** This enables standard WSDL operate at the syntactic level needed to represent the requirements and capabilities of a Web Services. This allows Web service developers to annotate their Web services with their choice of ontology language such as UML or OWL. This is significant because the ability to reuse existing domain models expressed in modeling languages like UML can greatly alleviate the need to separately model semantics. The Semantics provides a process level description of the service which, in addition to functional information, models the preconditions and postconditions of the process so that the evolution of the domain can be logically inferred. It relies on ontologies to formalize domain concepts which are shared among services.
2. **ECCF documents:**  ECCF is “a grammar for navigating and quantitatively evaluating a layered collection of artifacts that explicitly express the totality of the static and behavioral semantics necessary to achieve WI in a loosely coupled, technology-diverse, distributed deployment environment.” Following ECCF guidelines, the conceptual, platform independent and platform dependant specifications will be generated automatically driven a model.

Service generator is pluggable Ecore component following “Title of Bediako document” design specification. Service generator is a composite component orchestrating individual Ecore components responsible to generate each artifact. The flexible nature of this architecture enables adding new generators as needed without disturbing a predefined process. Conversion between each of these artifacts is also supported. For example, service generation framework would support generating WSDL from XSD or POJOs, XSD from POJOs etc.

As shown in the diagram above, the service generator is responsible in generating service artifacts from a given information or data model. These service artifacts by themselves don’t serve business motivations unless the service implementation is coupled with business implementation. The service assembler is responsible to:

* Plug in business implementation with the service implementation. This would enable loose coupling between the service implementation and the business implementation. This is important when a service is regenerated from the model. This loose coupling makes service regeneration easy by focusing on service artifacts only.
* Generate directory structure and package all artifacts into a deployable enterprise service.

## Implementation Principles

Following diagrams shows the overall approach of artifact generation. With the loose coupling nature service generation framework coupled with CXF, an enterprise deployable service can be generated from a model in two approaches.

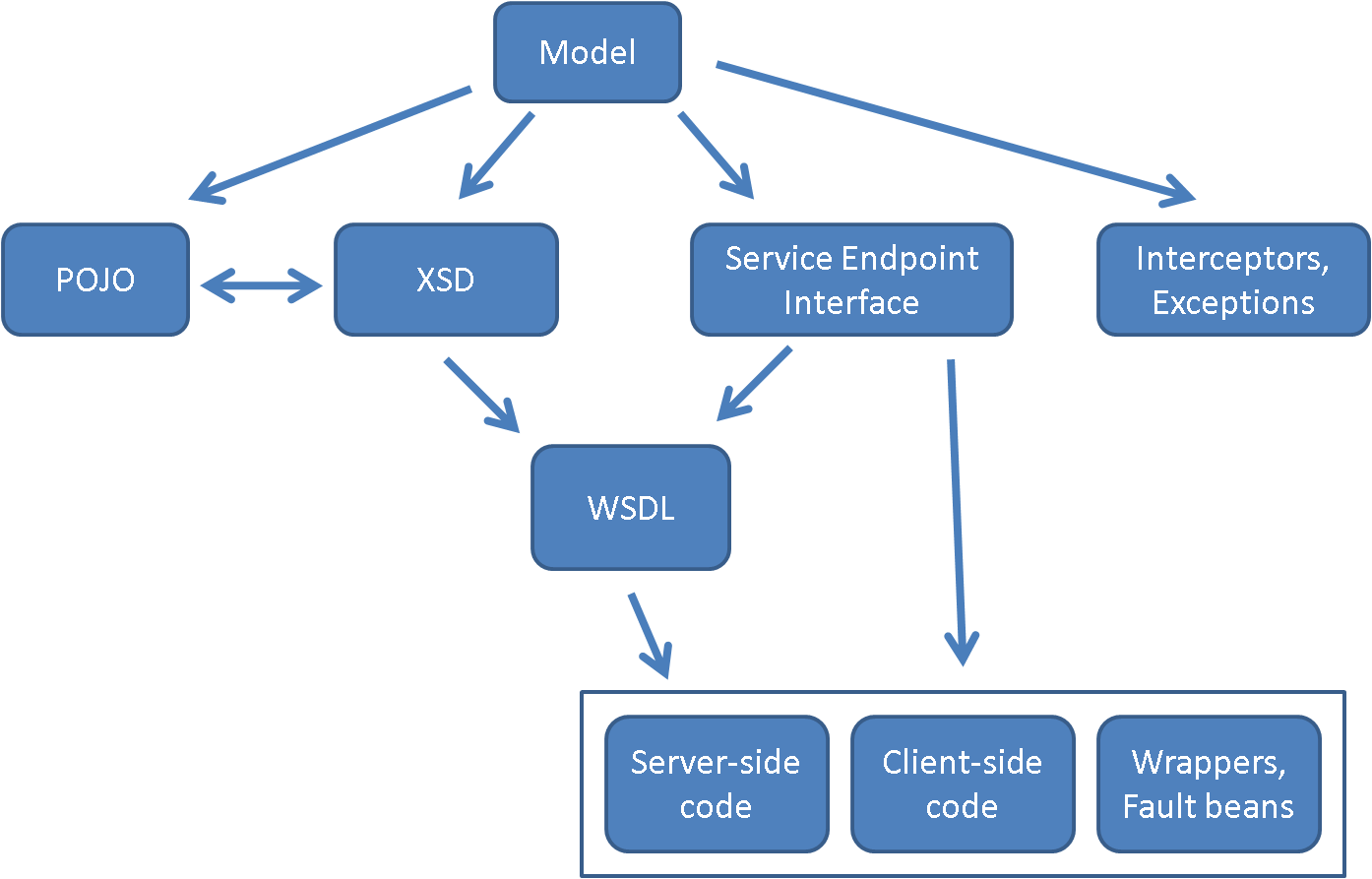


Figure : Service Implementation Principles

1. Code first: With this approach, service generator component will generate a service endpoint interface with all standard, user opted annotations from the model. Service development framework will use CXF tools to generate WSDL or service artifacts including interceptors, exceptions, service implementation, stubs, client-side code, wrappers, fault handlers. This would also generate configuration files necessary to deploy the service.
2. Contract first: With this approach, service generator component will generate XSD from the model. Service development framework will use CXF tools to generate WSDL and to further generate all above mentioned artifacts.

## Service Assembler

When designing an application, developers develop a logical model of what an enterprise does in terms of business objects (such as product, customer, order, bill, etc) and the services the business requires from these business objects (what is the stock level, what is the delivery schedule and so on). The developer may implement these concepts as a blend of service interfaces and components (the business objects). Components are normally used to implement (realize) the service functionality. It is important to distinguish between these two elements to enable loose coupling and separation of responsibilities.

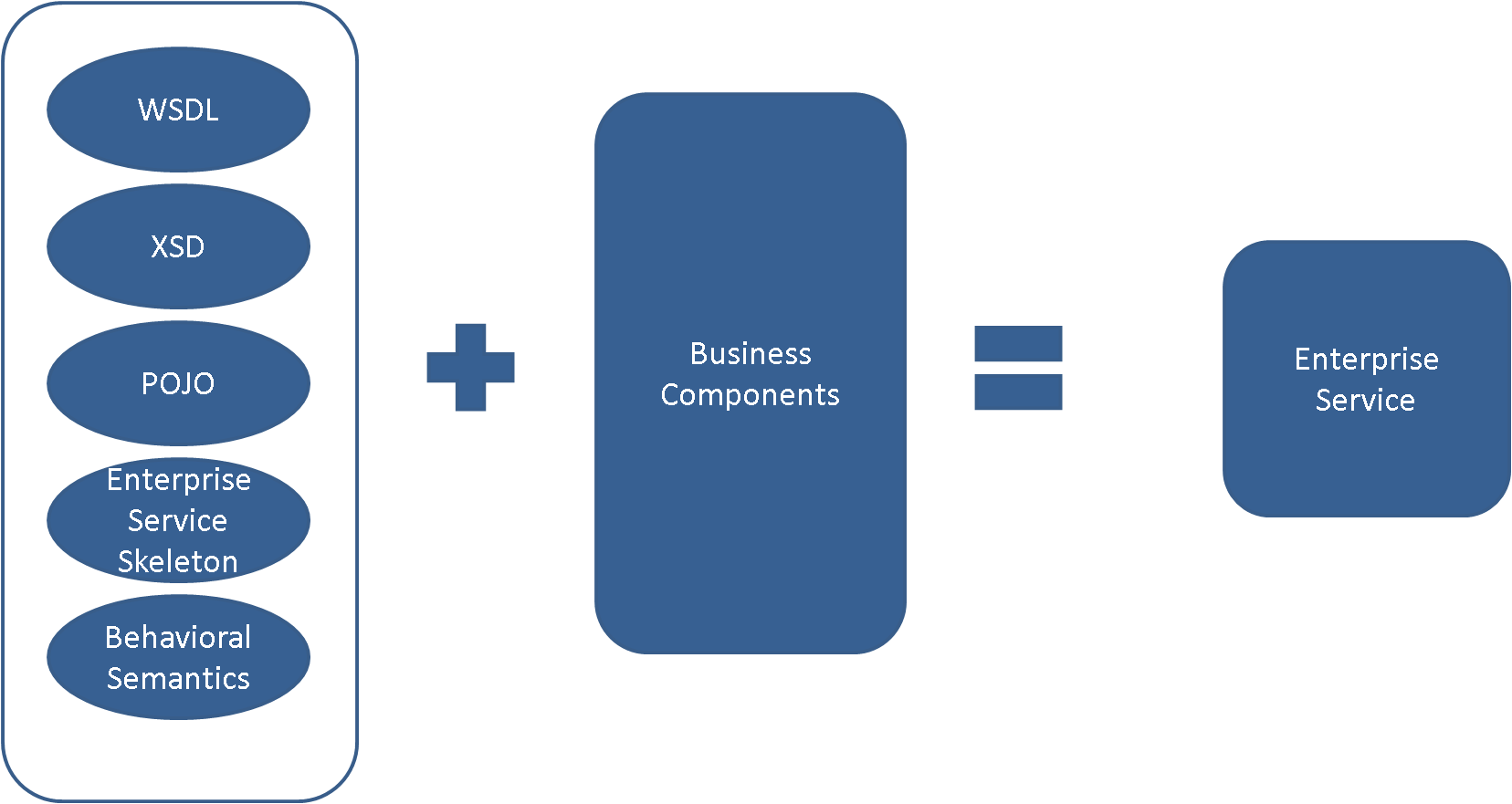


Figure : Service Assembler

Service assembler is responsible to automatically plugin service implementation with business components serving the actual business functions either through configuration or UI.

## Service Synchronizer

During development, it is often possible for a developer to update automatically generated code manually for various reasons. This could generate a gap between the model that was used to generate the code and manual modifications. It is important to keep these parts in sync to make sure any further runs to generate the service artifacts would not overwrite manual changes. Service synchronizer is responsible to capture any changes in the service artifacts and carry those changes back to the model to synchronize it with the code. Following diagram shows high level flow of the process.

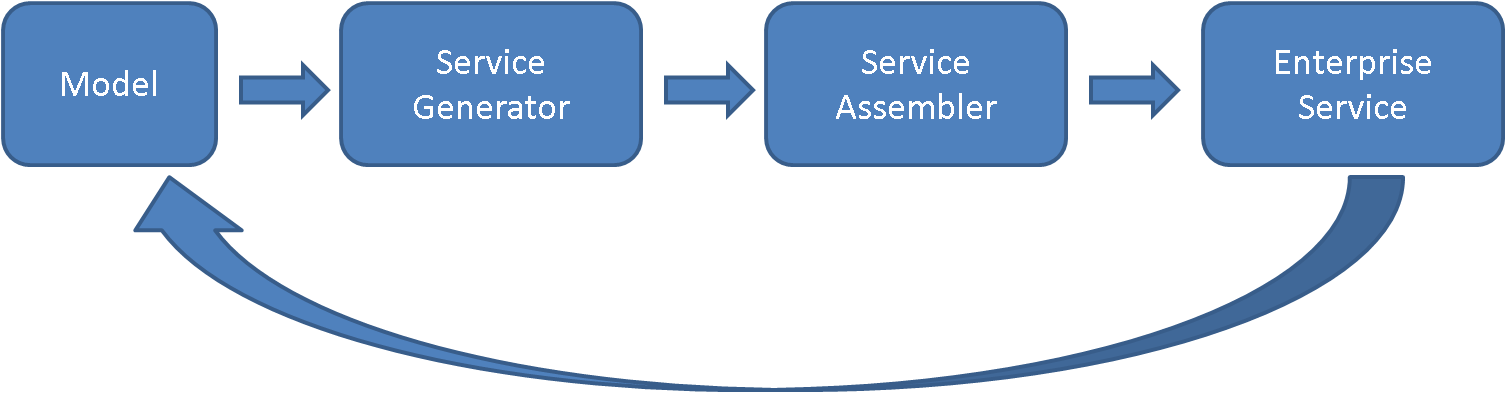


Figure : Service Synchronizer

As it is practical to make many types of changes to the generated artifacts, the initial approach of this design to restrict synchronization process to capture changes made to POJO operations. That is, if there are any new operations added, or changes to existing operations signature will be captured and synchronized back with the model. The synchronization process can be broken down into following components:

* Reader: This component is responsible to read generated artifacts and represent them in model format.
* Synchronizer: This component is responsible to compare original model with generated model from the updated artifacts. Synchronizer will present comparison results in a text format or on a UI with side-by-side comparison windows to move changes. Any confirmed changes will be written into the original model.

# Technical Design – Service Development Tooling

## Graphical User Interface

The SDK Eclipse IDE plug-in (referred to from here on as the plug-in) provides a development environment that fosters the generation of application artifacts from an Ecore model. The plug-in helps developers convert application models created in a variety of meta-model representations into an Ecore model. From this ecore model, pluggable artifact generators can produce artifacts. Of importance are the following components of the GUI:

* Model Converter Manager
* Model Explorer
* Generator Manager

The Model Converter Manager provides a graphical user interface (GUI) for the management and use of model converters. Model converters are utility components that can convert a meta-model of a specific type to an Ecore model, or an Ecore model to a meta-model of a specific type. Model converters are designed to be pluggable in nature, using the same mechanisms used to facilitate pluggable generators in the SDK Core Generator Manager.

The Model Explorer supports the viewing of the individual meta modelling facets of a model. Using the Model Explorer, users may view the meaning, persistence, presentation, validation, security, and object representation aspects of a meta model.

The Generator Manager is used to execute pluggable generators against the Ecore model. the actual functionality of this component is described in the SDK Core Generator Design document. In this document, treatment is limited to how this component interfaces with the plugin.

**Model Converter Manager Model Conversion Interface**

The Model Converter Manager (MCM) model conversion interface will allow users to identify the list of model converters available for the conversion of meta-models from one format to the other. Users may choose to try to convert a model by using the file dialog to locate the serialized model on the file system. After choosing to open the model, the available model converters will be queried by the MCM, and asked whether or not they can convert this meta-model. Only those converters that can convert the meta model that was opened by the user will be displayed on the interface screen. If no model converters are identified, the user will be told that no model converters for this conversion are available. The following mock gives a representation of how this screen will look to the user:

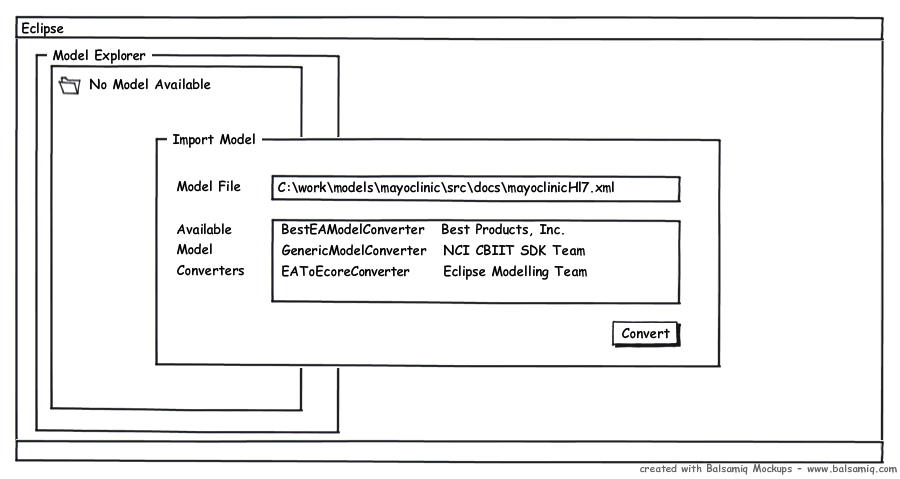


Figure : Model Conversion Interface

After the model converter is chosen, the user may press the convert button. Model conversion will then commence, with the model conversion progress reported to the Eclipse console. If the model conversion fails the user will be presented with the empty Model Explorer interface, and the Eclipse console will indicate the failure, along with the appropriate warning and error messages that can be used to diagnose the issues uncovered by the model converter.

If conversion succeeds, the user will be presented with the loaded Model Explorer interface. The user will be able to browse the converted data model and view the six different aspects of that model from different screens. In the next chapter, the parts of the Model Explorer will be explored.

**Model Explorer**

The Model Explorer allows the user to discover the aspects of the model that was converter. The following mockup describes how this interface will look:

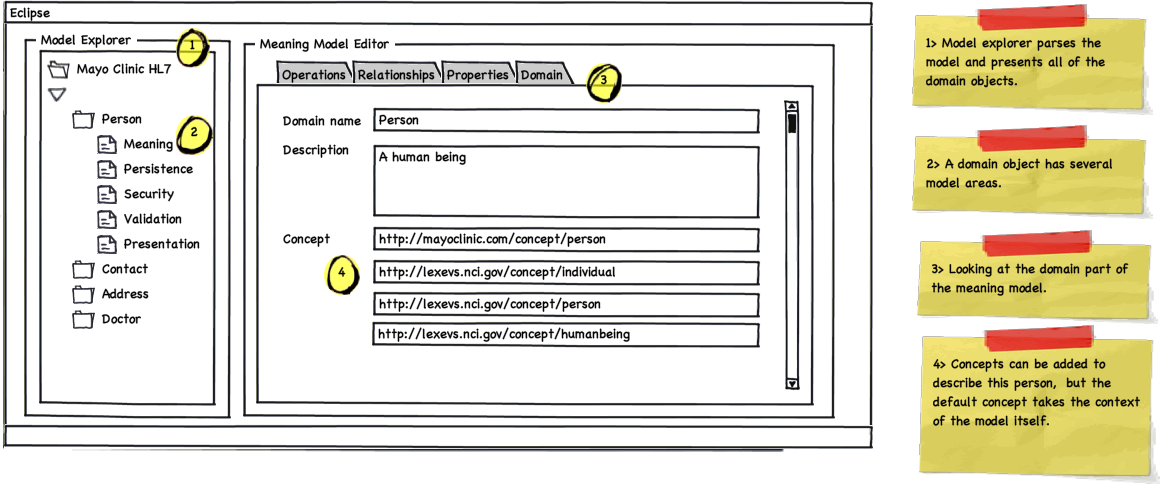


Figure : Model Explorer Interface

In this illustration, the meaning part of the meta model is displayed. This information on this screen is backed by the intermediate form of choice. On the left is the model explorer panel. This scrollable tree based view displays the models classes as domains. Clicking on a domain exposes the six different facets of models underneath it. By default the meaning screen is presented to the user first. The panel on the left is tabbed to represent four of the five components of the meaning facet of a model. The user may choose to look at the domain, property, operation, or relationship facets. The concept facet is not represented as a tab. Instead, concepts can be applied to either of these facets on the particular facet screen. In the above mockup, pointer 4 highlights how concept URIs are associated with a the model element Person domain. The following mockup show how this idea is extended to the property aspect of the meaning part of the model:

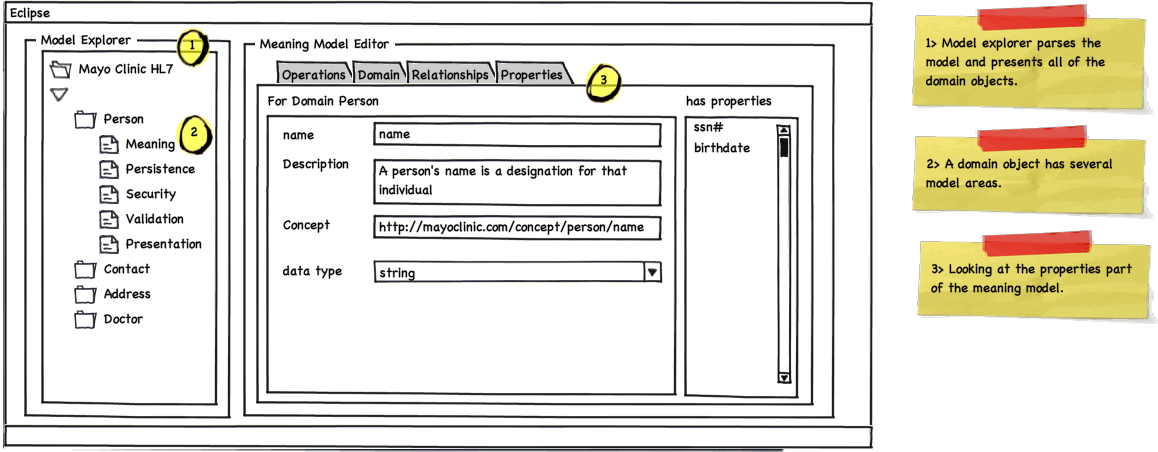


Figure : Model Property Browser Interface

This tab screen looks similar to the domain model tab, except as there are multiple properties per domain model, the properties list panel is included on the right side of the screen. A user may select a particular property for viewing in the center panel. Property aspects peculiar to the meaning part of the model are then displayed for this property. Of note is that a property may have concepts associated with it as well as datatype, name, and other information.

The relationship and operations tabs follow the same pattern. The following diagrams provide a picture of these aspects of the meaning model. Note the similarity in the user experience for these two tabs. Here is the relationships tab:

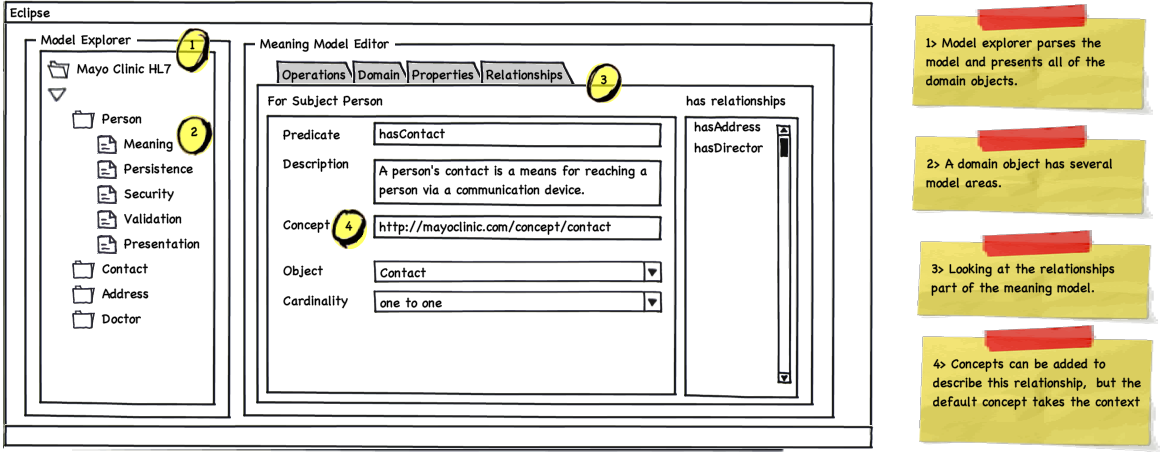


Figure : Model Relationship Browser Interface

and the operations tab:

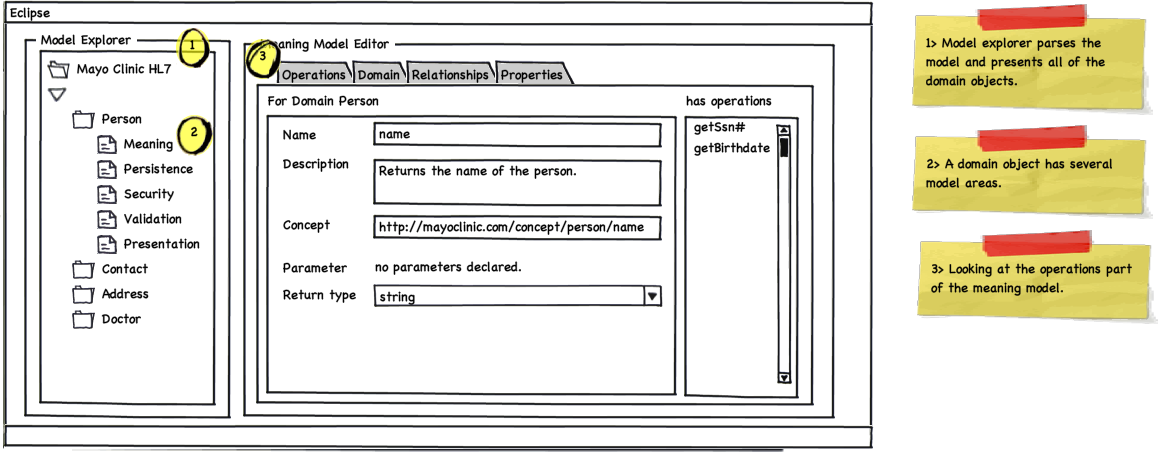


Figure : Model Operations Browser Interface

The other portions of the model are displayed in similar ways. For modelling persistence the following screens show examples of how this will work. Again an effort is made to keep the user interface look and feel consistent across the six aspects of the model. First a look at the persistence model for domains screen. This screen is shown by default when the “Persistence” folder is opened.

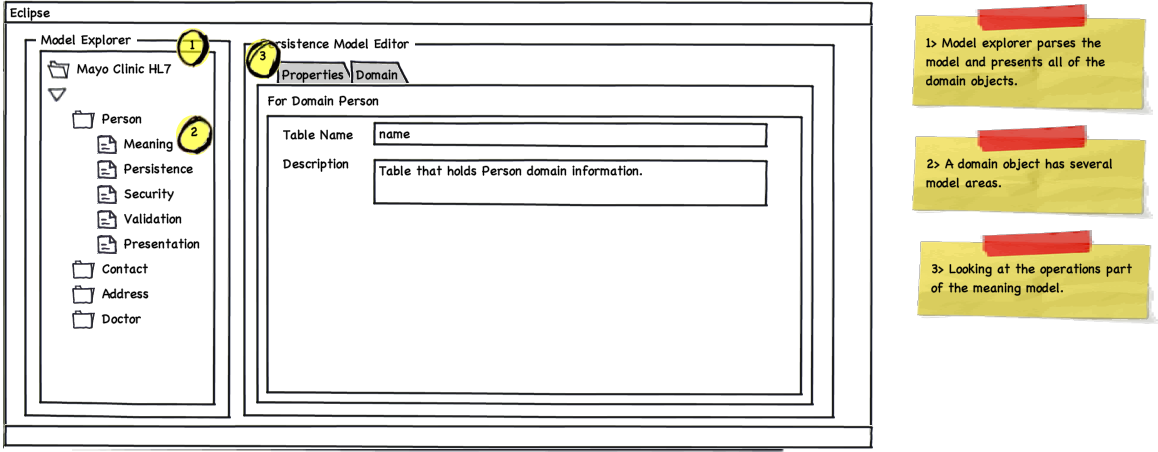


Figure : Data Model Browser Interface

Choosing the Properties tab on this screen will render a view similar to the following:

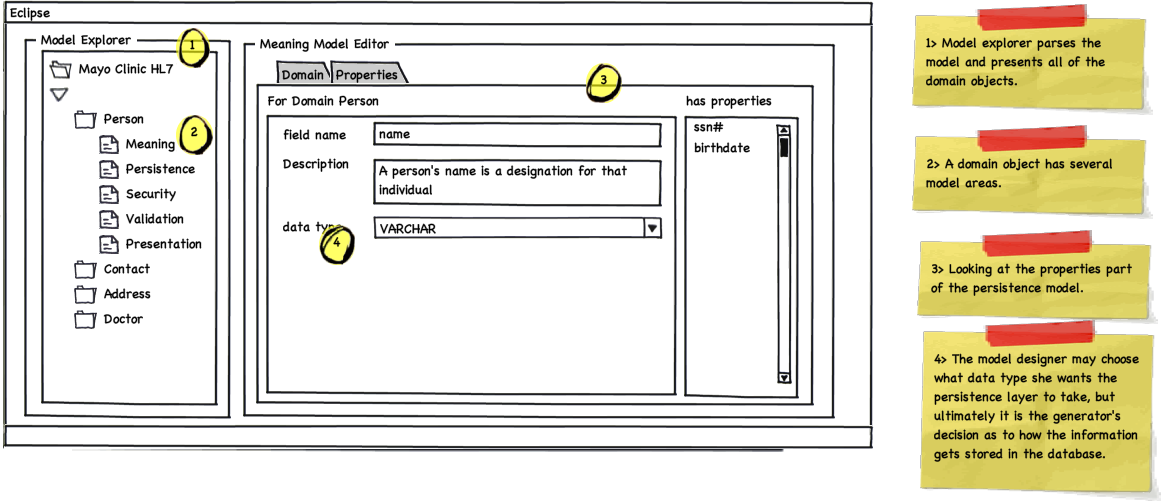


Figure : Model Properties Browser Interface

**Artifact Generation**

Once satisfied with the model, users of the plugin can choose a particular generator to generate the appropriate artifacts from this model. The plugin sports a generator choosing screen to facilitate this. The user may select from a collection of installed generators, and also select the which domains from the model should be used to create generated artifacts. The following illustrates the user interface for artifact generation. To get this screen, users are expected to right mouse click the model explorer to get to the “Generate Artifacts” floating menu item.

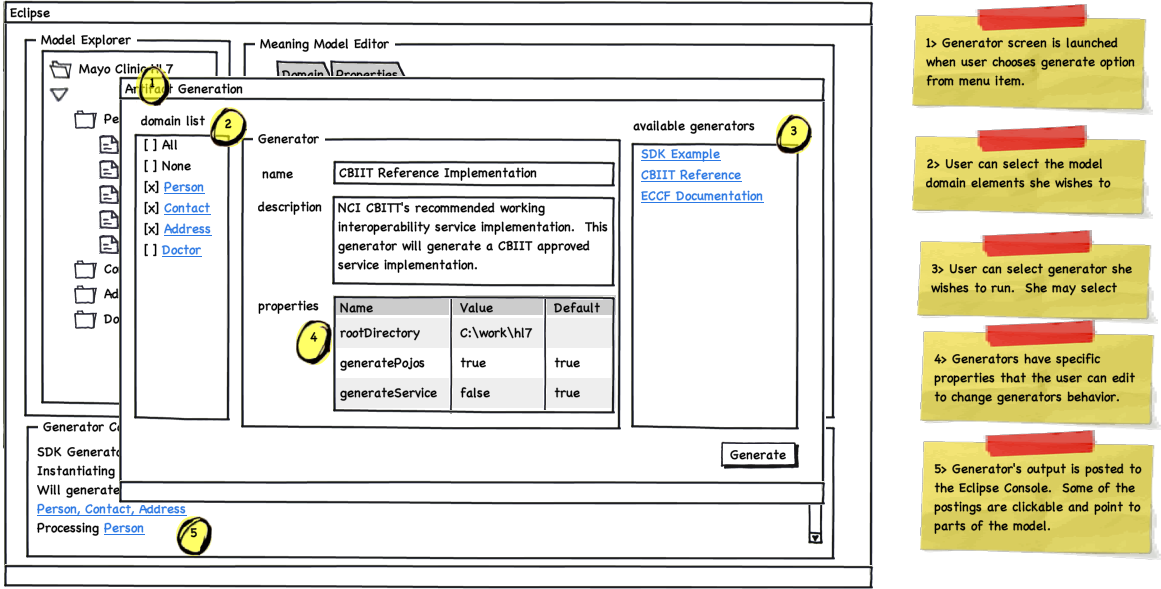


Figure : Generator Interface

As the Artifact Generation screen is presented to float over the Model Explorer screen. This was considered to be ideal as generation is a relatively short lived activity in which the user chooses the relevant domain elements to be used for generation, as well as the generator she wants to use. A user may choose to use only one generator at a time, but may decide to choose any number from zero to all of the domains from the model to be fed to the generator.

On the center panel, the name of the generator and the text describing the generator are displayed at the top. Also the user sees the properties the generator is expected to made available for its proper execution. The user may choose to edit these properties, or leave them as they are. Once the user has selected the domains to be processed, and the generator to process them, and is satisfied with the generator properties displayed, she may press the “Generate” button to begin generation.

At this point the Artifact Generation screen will disappear, and the Eclipse console will be displayed on the bottom portion of the screen. The domain by domain progress made by the plugin generator will be displayed here. Clickable status will scroll past as the generator makes progress. Once the generation progress is completed, the console with indicate a successful run, or display the list of errors that caused the run to be unsuccessful. The user may click on these errors to be taken to the part of the model that caused the issue. If the issue is addressable, then the new model can be used to feed the next generation sequence.

## Code Generator

### Model Representation

A model that represents other models is called a meta-model.  If it is at all possible to have a single meta modeling framework that can successfully represent all of the types of models we visited in the previous chapter, that would be beneficial.  The following is a treatment on the types of meta modeling techniques or frameworks that are available:

* 1. **Language specific** - a language specific representation refers to the programming language mechanisms used to represent the modeling of meaning constructs.  These support modeling of data types, constraints, properties, property containers (object classes), concepts, and relationships. Some meta modeling frameworks are better suited than others in modeling these constructs. These language based constructs are, of course, language dependent, and as such are best suited to be deployed within their own development environment.Java's annotation constructs are a good example of this.
  2. **ISO 11179 v3** - ISO 11179 v3is a meta modeling standard supporting the modeling of data types, constraints, properties, property containers (or object classes), first class concepts, and first class relationships.
  3. **OWL 2.0. -** OWL supports modeling of data types, object types, constraints, properties, property containers (classes), and first class relationships.  Concept support could be easily added and made a first class citizen for a given ontology.
  4. **Eclipse Modeling Framework (EMF) Ecore** - Ecore supports the modeling of properties, data types, property constraints, property containers (classes), and first class relationships.  Concepts are not directly supported, but they could also be added via Ecore annotation support.  Ecore enjoys support from the Eclipse community as well as IBM.
  5. **XSD, XMI, and UML 1-2**- These support the modeling of data types, constraints, properties, property containers (classes), and weaker forms of relationships like associations or aggregations.  Relationships are not first class citizens but they can be supported if a definition of a relationship is supplied.  Concepts are not supported, but they could also be added.  In the case of UML this could be accomplished via stereotypes.  With regards to XSD, additional complex types of type “Concept” could be introduced and used to model concepts as a second class entity.
  6. **Data Definition Language (DDL) -** These support the modeling of data types, limited constraints, properties as fields, property containers as tables or views, and weaker forms of relationships as foreign keys.  Relationships are not first class citizens but they can be supported if a definition of a relationship is supplied.  This definition would likely be represented as a table.  Likewise, concept modeling is not supported, but that could be addressed in a similar fashion to modeling relationships.  Finally, DDL is specifically oriented towards representing persistence models.  Models such as presentation, representation, security, and others are not efficiently modeled using DDL.

Users of SDK will certainly use language based approaches for modeling. They will also use UML modeling techniques and model representations such as XMI.  Still others will use DDL as their sole model representation.  It is apparent that any software development framework that facilitates ease of translation from one representation format to another will attain high usability ratings.  In particular, if that framework can also address the possibilities of lost information during model conversion amongst unequally powered meta models, this would be a positive development.

For instance, given the different power of ISO 11179 v3 versus DDL, it is inevitable that some attempts to meta model using DDL constructs that are stored and modeled in ISO 11179 v3 will fail.  DDL cannot natively support the notion of a “concept” for instance.  It would be ideal to have a single representation that can be gracefully transformed back in forth amongst these representations, dealing with information loss issues along the way.  Such a model would have to be able to sufficiently support the highest and most complicated demands of meta modeling.  The diagram below illustrates this idea of a universal intermediate form below.

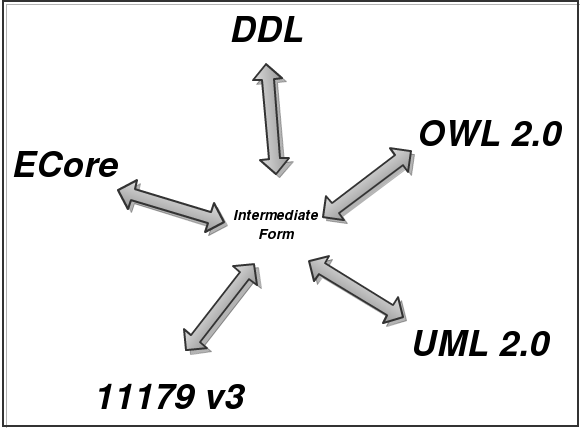


Figure : An intermediate form capable of representing the fundamental parts of modeling can serve as a universal translator for other meta-models.

Of these modeling frameworks, in terms of capability considerations, ISO 11179 v3 and OWL 2.0 come closest to fitting the bill, however, for reasons of pragmatism we would not recommend using these meta models as the central modeling framework at this time.  Later in this document the SDK team will introduce a more practical approach.

For practical purposes, the SDK team will chose EMF Ecore as the candidate for the new SDK artifact generator’s intermediate form.  As mentioned in an earlier treatment, Ecore’s model satisfies all the requirements for a meta model.  It can represent classes, properties, operations, and bi-directional relationships as first class model constructs.  Ecore’s lack of first class support for concepts is a small issue that could be addressed by the use of annotations.  In Ecore, annotations can be applied to classes, relationships, properties, and operations.   In addition, the other benefits of using Ecore cannot be overlooked, in that Ecore provides a code generation facility that can be used as a template for the creation of the SDK generator.  As mentioned before Ecore has both Java community and industry backing.  The Ecore model can be represented in other meta modeling formats such as OWL.  This would allow us to use this model outside of the Java world should there be a need to do so.

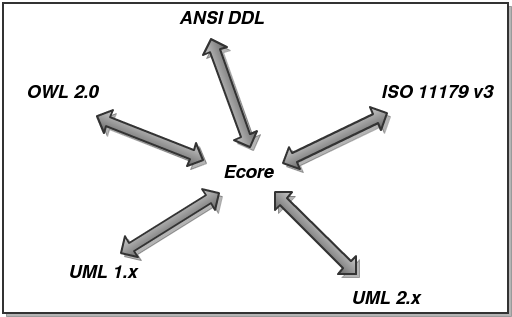


Figure : Ecore as a universal intermediate form for meta model translation.

### Challenges

Although Ecore is a full fledge code generation utility, attaining reuse by extending it can prove to be a challenge.  Outside of the usual upgrade path considerations, the Ecore generator has several limitations that make it a non-ideal candidate for total reuse for the new SDK platform.  The issues are as follows:

* Ecore generates only Java related artifacts.  It does not readily allow for the generation of artifacts in other languages.  This does not allow Ecore to accommodate all developers, since generation of multiple language artifacts is not supported.  An instance where this would be an issue would be in the generation of JavaScript for browser based form validation.
* Ecore uses the JET template system to generate code.  JET is not a pure presentation template system[[1]](#footnote-2), and is very Java centric in nature.  Moreover, there are more suitable open source templating systems available that are also multi platorm in nature.  One such example is StringTemplate[[2]](#footnote-3), which runs on both Java and Python.
* Ecore’s default Java generation artifacts are not designed to facilitate use outside of the Ecore environment.  SDK will create artifacts that can be used outside of the code generation and model frameworks used to create the code.  This source level dependency that cannot be broken without refactoring Ecore itself.  Although we will be using SDK to enhance semantic interoperability, we do not wish the SDK itself to inadvertently become a hamper to the sharing of its generated language specific constructs.

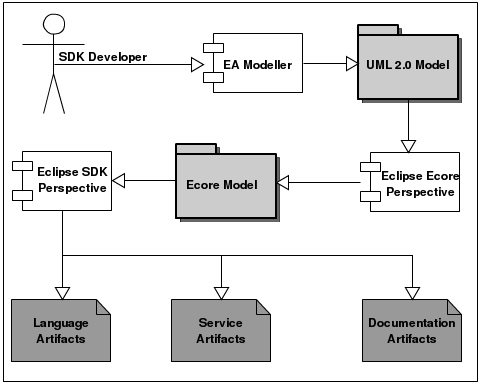


Figure : Sample development workflow involving model translation and artifact generation.

Instead of reusing Ecore as it is, SDK should use Ecore as a template for designing a new Eclipse based code generation utility that completely meets the goals of the SDK developer community.  The only direct dependency would be supporting the Ecore model representation.  This will allow SDK to take advantage of the model to model translation facilities present in Ecore, while still improving on the existing code generation abilities of Ecore itself.

**Model Tagged Values**

Outside of the meaning and persistence of a model, many meta-models do not provide the ability to model presentation, validation, representation, and the security aspects of the model. Indeed, the Ecore representation the SDK team will be using as an intermediate form does not provide this capability as a first class modeling feature. Consequently, we are relying on model annotations to express these other aspects. These annotations can be represented in other meta modeling languages as well. For instance, these annotations could be represented as object and data properties belonging to a class in OWL. In the Java language, the annotation construct provides a direct comparison to the EAnnotation in Ecore. In UML these annotations could be approximated using the tagged value construct.

The SDK team has come up with a string based representation format for model annotations. These tag values are documented in a separate document.

### Code Generator

In order to efficiently generate all of these different types of targets it is important that SDK be designed in a modular fashion.  SDK generators should be built in such a manner as to facilitate the easy inclusion of new generators to suit the purpose at hand.  This should be configurable.  
  
Illustrated below is an example of one such high level modular code generation architecture:

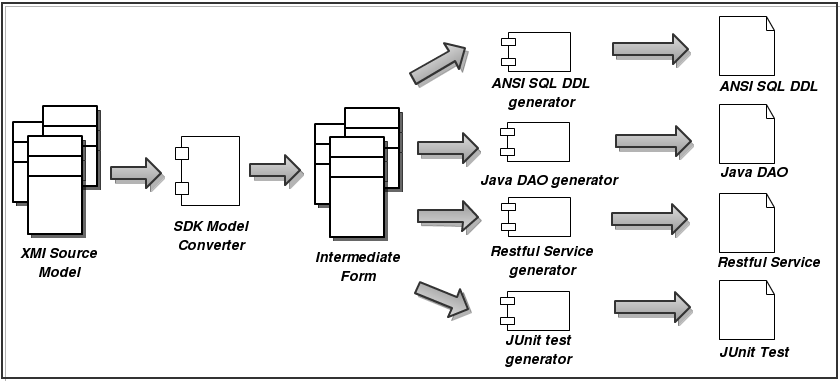


Figure : High level candidate architecture for the new SDK.

In this architecture, a domain model represented in XMI is translated by an XMI specific SDK model converter into an intermediate form.  All generators are familiar with this intermediate form construct and can create their own specific target artifacts independently of the other generators.  An added benefit of this approach is that generators can use other generators to generate fragments of code, which can then be combined to create a composite generated code fragment.  For instance, a DAO code generator may rely on an ANSI SQL code generator to generate the ANSI SQL that it will use in its DAO methods. Of course this reuse need not be limited to two generators. It would be completely acceptable for whole chains of generators to use each other in this fashion.

Another advantage of using this pluggable generator approach is that new platforms can be supported without changing support for existing platforms. As an example, the same intermediate form that was used to generate a web based application could also be used to generate an application for Google’s Android mobile phone platform. Furthermore, that Android application generator could be created without any support from the SDK development team. Along the same lines subsequent versions of the generated Andriod platform could be produced by generators that are Android platform version aware. This represents a significant step forward over the existing SDK code generating framework.

#### Code Generator Design

Generating source code can be powerful, but the program that writes the code can quickly become very complex and hard to understand. One way to reduce complexity and increase readability is to use templates. The Eclipse Modeling Framework (EMF) project contains two very powerful tools for generating source code: JET (Java Emitter Templates) and JMerge (Java Merge). With JET, a JSP-like syntax (actually a subset of the JSP syntax) makes it easy to write templates expressed the code in required format to generate. JET is a generic template engine that can be used to generate SQL, XML, Java source code and other output from templates. It is located in the org.eclipse.emf.codegen plug-in as part of the EMF runtime download.

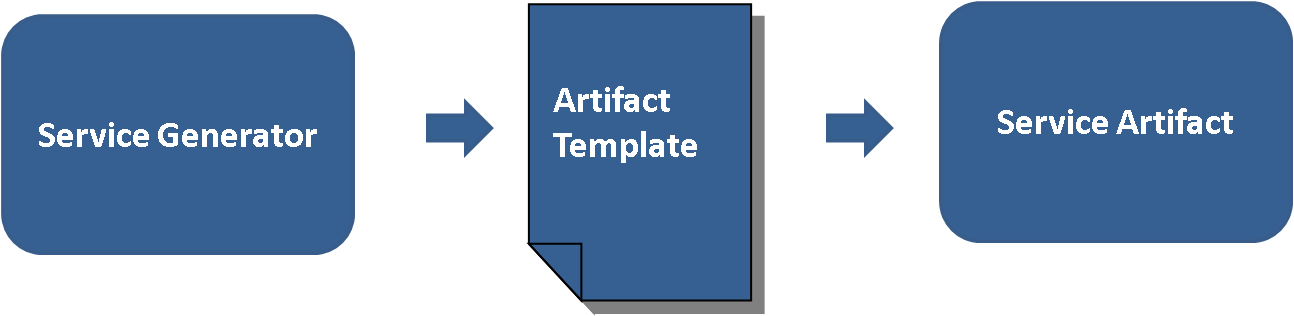


Figure : Template approach

Following is an example of JET file contents used to generate XML source code.

|  |
| --- |
| <%@ jet package="hello" imports="java.util.\*" class="XMLDemoTemplate" %>  <% List elementList = (List) argument; %>  <?xml version="1.0" encoding="UTF-8"?>  <demo>  <% for (Iterator i = elementList.iterator(); i.hasNext(); ) { %>  <element><%=i.next().toString()%></element>  <% } %>  </demo> |

The code below shows how to invoke the template instance.

|  |
| --- |
| List data = new ArrayList();  data.add("first");  data.add("second");  data.add("third");    XMLDemoTemplate generateXml = new XMLDemoTemplate();  String result = generateXml.generate(data);  System.out.println(result); |

This prints the following XML result to the console:

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <demo>  <element>first</element>  <element>second</element>  <element>third</element>  </demo> |

#### Validation

As shown and described above, model is the source of service generation. Service generator will need to validate the model before it starts generating different artifacts.

It should validate model format, model mappings and tag values. As shown in the diagram below, model validator is a separate component by itself that can be utilized by any component wish to validate a model. Model validator will support configurable validation rules through a standard XML format. This would promote loose coupling and reusability between these components. By running model validation, model validator will generate a human readable formatted output in XML and communicate back it to the caller.

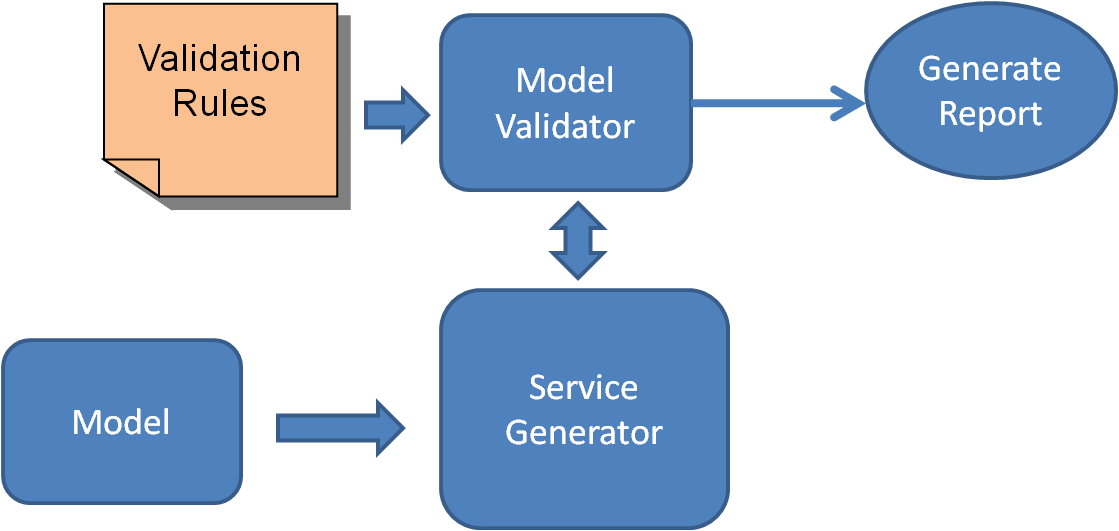


Figure : Model Validation Approach

#### Code Generation Framework

There are several components that comprise the SDK generator. They can be divided into 3 parts. The first is the intermediate form reader which contains all the components for reading an intermediate form from a file system, and then creating an in memory representation. The second is the generator conductor package. In this part the orchestration of the generators along with the management of the generator output is handled. Finally, there is the pluggable generator section, which is composed of a set of JSR-223 script based artifact generators and intermediate form validators. It is in this section that the creation of individual artifacts is manifested.

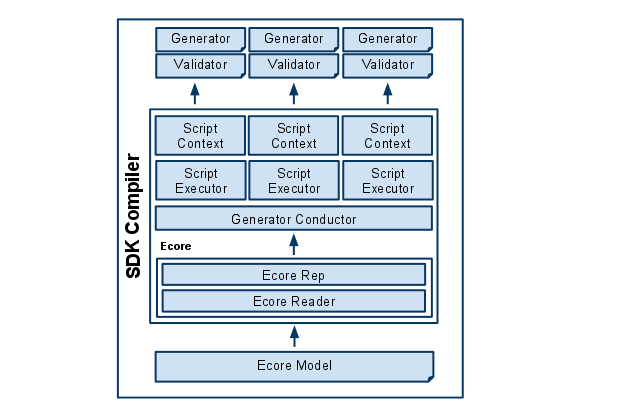


Figure : Code Generator Framework Architecture

This figure illustrates the design components that make up the new SDK generator.

* **Ecore Model -** XMI file saved by the Eclipse EMF Ecore IDE. This file represents information in the domain model including but not necessarily limited to, annotations, classes, properties, operations, and datatypes.
* **Ecore Reader -** The Ecore reader is part of the Eclipse EMF Ecore framework. It has the ability to read an Ecore XMI file and transform it into an in memory Ecore representation.
* **Ecore Representation -** The Ecore representation is a collection of Java objects that are members of the Eclipse Ecore package. These objects hold the meta model information that was contained in an Ecore XMI document. The Ecore in memory representation can be accessed during the Java runtime.
* **Generator Conductor -** This is the object responsible for level 0 validation of the Ecore model, as well as orchestrating the execution of a package of generator scripts. Level 0 validations are platform independent model quality checks that can be applied by the generator conductor as opposed to the pluggable generator. The generator conductor also makes sure that every generator sees every domain in the domain model that was selected for generation exactly once. It instantiates a Script Executor and a Script Context for each script in the generator package. The Script Executor is instantiated again and again for each execution, but the Script Context, once created, remains available to its assigned generator for the duration of the generation execution.
* **Script Executor -** This object is responsible for the executing the generator script in its target script language. Script Executors pass the generation execution objects like the Script Context, the error manager, a FileUtility for artifact file management, and logging support to the generator scripts. They also provide the script with the domain representation and the object that references the domain object for which artifacts should be generated as well as a user specified generator specific property map.
* **Error Manager -** This object provides the executing generator scripts a facility for reporting errors to the Generator Conductor.
* **Script Context -** The object serves as a memory location for each generator script as well as a global memory store for the entire pluggable generator. The Script Context holds a map object that the generator scripts can use to store its own artifact generation information. It also contains a reference to the package level map. This map can be used by all generator scripts in the generator package to pass information amongst all the generator scripts in the package.
* **Validator -** Validator is a script that will be executed once and once only for a given script generator. Validators are used only to make sure that the Ecore representation passed to the generators are valid and can be processed without error. Validators can be written in any language that is supported by JSR-223.
* **Generator Scripts -** Generator Scripts are scripts that are responsible for generating artifacts from models. Generator Scripts are expected to receive a domain model representation and a domain object. They also receive a reference to the root output directory to be used to write the generated artifact. The Generator Conductor shall determine the set of generator scripts by looking into the “generator” directory. In the generator directory, generators will be contained in their own directory structure. The name of the immediate sub directory under the generator directory shall be taken as the name of the generator package. All generator scripts in the generator package shall be executed together as one generator application.

The following diagram describes how these classes are associated with each other and gives more details surrounding their individual responsibilities and constituent parts.

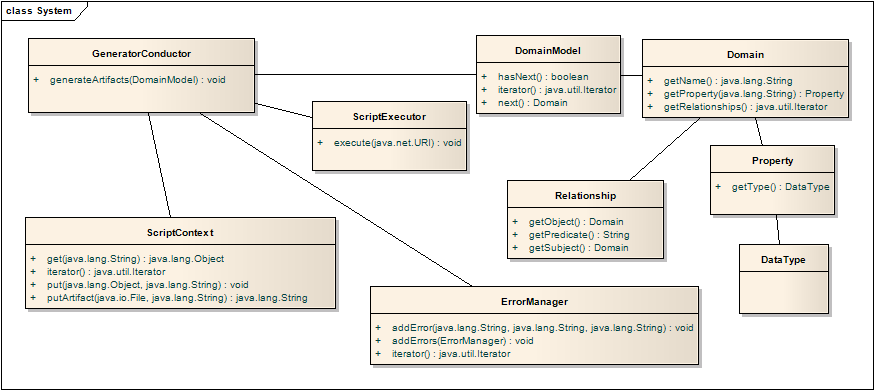


Figure : Code Generator Component Class Diagram

**Pluggable Generator Layout**

**<generator name>**

-> artifacts.txt (list of the artifacts this generator will generate)

-> description.txt (description of generator)

-> version.txt (generator version information)

-> multiple scripts of the form “<script name>.<language extension>”

-> multiple script libraries of the form “<script name>.<language extension>Lib”

-> single script of the form “validate.<language extension>” (For level 1 validation)

-> jar (directory that stores supporting jar files)

-> generator.properties (Java style property file for defining generator properties)

-> template (directory that stores StringTemplate template files)

**Generation Workflow**

The SDK generator will follow a single workflow that will not change irregardless of what is being generated. In essence, the SDK generator is responsible for exposing the intermediate form file to the JSR-223 script generators. Barring errors in consuming the intermediate form, generators are guaranteed to be executed once for every domain object identified in the model. Generators are then able to create artifacts relevant to the domain objects they are exposed to.

The diagram below illustrates the entire work flow for the SDK generator:

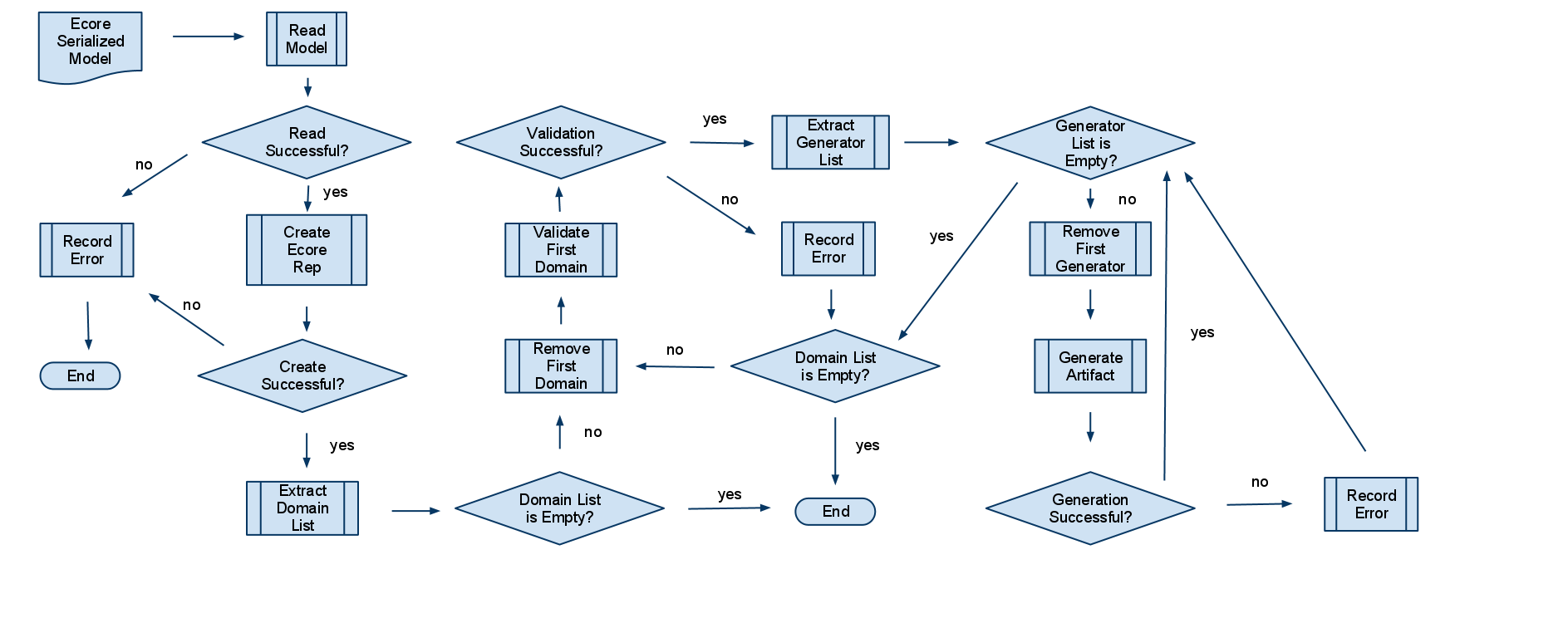


Figure : Code Generator Workflow

This figure illustrates the work flow process for generating a set of artifacts for a given domain model.

Here are the success path steps that lead to the first artifact being generated by a single generator script for the first domain object in the model.

1. The intermediate form (Ecore Serialized Mode) is read into memory.
2. Then the intermediate form in memory representation (Ecore representation) is created.
3. The intermediate form is subjected to platform independent model validations.
4. The intermediate form is subjected to generator specific model validations.
5. The list of domain objects is extracted from the domain model in the in memory intermediate form representation.
6. The first domain object is removed.
7. Load list of generator scripts.
8. The first generator script is removed.
9. The first generator script is executed on the first domain model and the in memory intermediate form.
10. The first generator writes the generated artifact to the root output directory.
11. The generator then removes the next domain object from the domain list and repeats the previous steps from 6 - 10 until there are no more domain objects to process.

For each of these steps, an error can occur. As long as the intermediate form is successfully read into memory, the generator conductor will continually execute generator scripts for each and every domain in the domain model until every generator script has seen every domain. Error collected by the error manager will be logged to the file system and reported to the screen as compiler errors.

**Logging**

The SDK Compiler will rely on the java.util.logging package. No other logging facility will be supported as this is sufficient and will not be effected by third party upgrade cycles. Logging messages deemed as severe will automatically be reported to the error manager.

**Error Reporting**

SDK generator will report errors experience by generator scripts at the end of the generation execution. An error manager will be made available to all generator scripts. This error manager will expose and interface that will allow generator scripts to report an error as a category, name, and message. This will allow the reader of generated messages to interpret where the error was discovered. The SDK generator will add additional information to this error when it is reported. The format for a generated compiler message is described below:

<Date> <Time> <Package Name> <Script Name> <Category> <Name> <Message>

If an exception was thrown the stack trace will be appended on the end of this message. Any Throwable exception thrown by generator scripts will also be caught by the Generator Conductor and added to the error manager for report as a compiler error. The stack trace will be appended at the end of the compiler message after the <Message> section.

**StringTemplate templating library**

The StringTemplate templating library has been chosen to implement templating requirements for the pluggable generators. Although generators are not forced to use StringTemplate, it is the official templating solution for the SDK, and utility tooling will be provided to support this library. Standardization on a templating library maximizes the ability for developers to reuse templates across generators. Commonly generated constructs such as SQL, DDL, and HTML are candidates for the creation of reusable templates.

StringTemplate brings an added benefit of encouraging a clear separation of model and controller code. This decoupling of models and controllers helps foster generator code reuse.

**Java Specification Request (JSR) - 223**

JSR-223 is is a specification that describes how to integrate scripting languages into the Java platform. This enables developers of multiple scripting languages the ability to write code in a supported scripting language, and execute that code on the Java platform. JSR-224 scripting language support is quite impressive, with scripting engines available for Java, JavaScript, Python, Ruby, Perl, and others[[3]](#footnote-4). It has been supported since JDK 1.5.

Using the JSR-223 scripting support in the JVM opens up the SDK generator abilities to a wider audience. In particular, with the presence of Rhino, a JSR-223 compliant JavaScript interpreter from Mozilla, a wide net of developers will be able to create their own generators.

Scripting developers using the SDK generator will have access to the critical Java objects that contain all the information that would be needed to interpreter a domain model as well as produce a generated artifact. The following diagram illustrates the high level architecture of the scripting environment developers will be using to access the intermediate form domain model object and other supporting generator infrastructure objects.

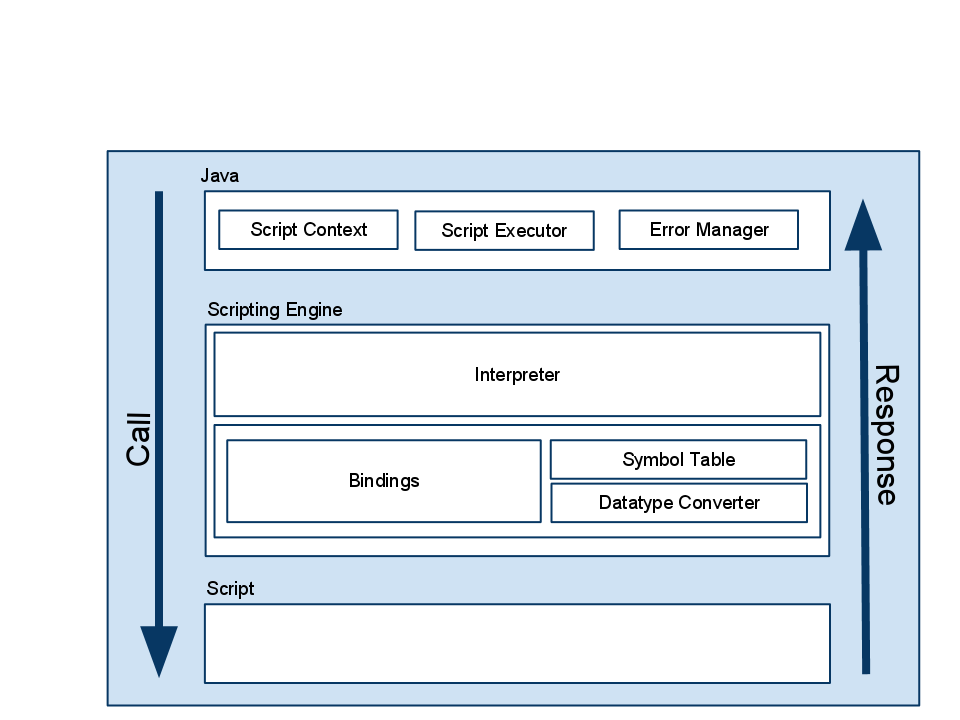


Figure : JSR 223 Based Scripting Architecture

As the diagram shows, the Scripting Engine consists of an Interpreter, and map of bindings, a data type converter, and a symbol table. Relevant to the SDK design is the presence of a bindings map. The SDK generator will bind Java objects (map values) to names (map keys). These names (map keys) will appear to the scripting developer as globally accessible variables in their program. From the inception, at the very least, the following objects will be made available via this mechanism.

* ScriptContext as variable SCRIPT\_CONTEXT- This object stores the memory slots to be used by the developer to store information that must persist beyond the execution of a single generator script.
* Log Utility as variable name LOG - java.util.logging.Logger based object to be used to log generator debugging messages.
* Intermediate form domain model as variable MODEL - the domain model in memory.
* Intermediate form domain of focus as variable DOMAIN\_FOCUS - the domain artifact to be generated.
* ErrorManager as variable ERROR\_MANAGER - this object shall be used to report generator errors. These are the errors that will be reported at the end of execution as official compiler errors.
* FileUtility as variable FILE\_UTIL - This utility is used to be the file handle to the directory that will be containing the generated artifacts.
* StringTemplateGroup as variable TEMPLATE\_GROUP - This templating object will store the group of templates located in the templating directory. Generator developers may access their templates via this object.
* Properties object as variable PROPERTY - This object contains the property names and values for the pluggable generator.

Providing these objects to the generator scripts will ensure uniform behavior for all scripts executing in the SDK generator environment.

## Mapping and Transformation

At this point, we have not designed the model mapping and transformation components. We anticipate these components to plug into 1) the GUI and 2) model representation.

## Semantic Metadata Management

At this point, we have not designed the model mapping and transformation components. We anticipate these components to plug into 1) the GUI and 2) model representation.

## Deployment

At this point, we have not designed the model mapping and transformation components. We anticipate these components to plug into 1) the GUI and 2) model representation.

1. The JET template system supports the use of programming logic in templates. This breaks the model view controller separation paradigm and as such could encourage the creation of coupled template presentation and creation logic. [↑](#footnote-ref-2)
2. Terrence Parr is the creator of StringTemplate. He is a avid compiler theorist and implementer,

   and StringTemplate benefits much from his research. More about StringTemplate can be found at

   www.stringtemplate.org. [↑](#footnote-ref-3)
3. Support for Microsoft Visual Basic was started by Sun in 2006 as a project called Semplice. Although demostrated at a conference, the source code for this engine was never released. In the future a Visual Basic or .Net scripting support may be created by a third party, or may eventually be released by Oracle. [↑](#footnote-ref-4)