

**Service Development Tooling**

**Requirements and Design Document**

Version: 1.0

Last Modified: 9/30/2010

Team : caCORE SDK Team

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** |
| --- | --- | --- | --- |
| 1.0 | 9/30/2010 | 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** |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

Product Team

|  |  |  |
| --- | --- | --- |
| **Management** | **Role** | **Comment** |
| Sichen Liu | Product Manager |  |
| Sreenath Nampally | COTR |  |
| Greg Raley | Technical Project Manager |  |
|  |  |  |
|  |  |  |

|  |  |  |
| --- | --- | --- |
| **Development** | **Role** | **Comment** |
| Satish Patel | Technical Architect |  |
| Bediako George | Developer |  |
| John Chen | Developer |  |
| Vijay Parmar | Developer |  |
| Dan Dumitru | Developer |  |

|  |  |  |
| --- | --- | --- |
| **Testing** | **Role** | **Comment** |
| Ye Wu | QA Manager |  |
| Xiaoling Chen | QA Engineer |  |
| Aynur Abdurazik | QA Engineer |  |
|  |  |  |
|  |  |  |

|  |  |  |
| --- | --- | --- |
| **Documentation** | **Role** | **Comment** |
| Jill Hadfield | Documentation Manager |  |
| Ann Wiley | Wiki Documentation |  |
| Bronwyn Gagne | Documentation |  |
| Carolyn Klinger | Documentation |  |
|  |  |  |

|  |  |  |
| --- | --- | --- |
| **Other** | **Role** | **Comment** |
| Kelley Hunsecker | Download Center Updates |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

Product Information

|  |  |  |
| --- | --- | --- |
| **Resource** | **URL** | **Comment** |
| Product URL |  |  |
| SCM URL | <http://gforge.nci.nih.gov/svnroot/cacoresdk/trunk/hydra/> |  |
| Defect Tracking |  |  |
| Feature Request |  |  |
| Listserv |  |  |
| Support |  |  |
| Documentation | <http://gforge.nci.nih.gov/svnroot/cacoresdk/trunk/hydra/src/doc/> |  |

Cross-Product Dependencies

|  |  |  |
| --- | --- | --- |
| **Resource** | **Contact** | **Comment** |
| Semantic Infrastructure v2.0 | Dave Hau (Product Manager) |  |
| caGrid (PST) v2.0 | Avinash Shanbhag (Product Manager) |  |
| caAdapter v5.0 | Sichen Liu (Product Manager) |  |
| CSM | Sichen Liu (Product Manager) |  |
| CLM | Sichen Liu (Product Manager) |  |
| ISO 21090 | Sichen Liu (Product Manager) |  |
|  |  |  |

External Software Dependencies

|  |  |  |
| --- | --- | --- |
| **Resource** | **URL** | **Comment** |
| Java |  |  |
| Enterprise Architect |  |  |
| ArgoUML |  |  |
| JBoss |  |  |
| Tomcat |  |  |
| Oracle |  |  |
| Postgres |  |  |
| MySQL |  |  |
| Ant |  |  |
| Eclipse |  |  |
| Windows |  |  |
| Linux |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

External Library Dependencies

|  |  |  |
| --- | --- | --- |
| **Resource** | **URL** | **Comment** |
| Eclipse |  |  |
| OSGI |  |  |
| UML |  |  |
| Spring |  | Only used for the SDK example. |
| Hibernate |  | Not used by current prototype. |
| JAXB |  | Only used for the SDK example. |
| JDOM |  | Not used by current prototype. |
| JSR 286 |  | Not used by current prototype. |
| Acegi |  | Not used by current prototype. |
| Apache Commons |  |  |
| CXF |  |  |
| StringTemplate |  |  |
|  |  |  |

NCI CBIIT Specific Information

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Resource** | **App Server** | **DB** | **CI Server** | **Comment** |
| DEV Tier |  |  |  |  |
| QA Tier |  |  |  |  |
| STAGE Tier |  |  |  |  |
| PROD Tier |  |  |  |  |
| TRAIN Tier |  |  |  |  |
| DEV CI Server |  |  |  |  |
| AntHillPro CI Server |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Table of Contents

1. Introduction 11

1.1 Document background and intent 11

1.2 Intended Audience 11

1.3 Purpose of the Document 11

1.4 Organization of the Document 11

2. Stakeholders 12

2.1 Implementation Governance 13

3. Goal and Objectives 14

4. Background 15

4.1 Current Solution/Current State 15

4.1.1 Service Modes 16

4.2 Problem statement 17

5. Product Requirements 18

6. Business Use Cases 20

6.1 Use Case – Manage Model 20

6.1.1 Use Case – Create/Update Model 20

6.1.2 Use Case – Import Existing Models/Components from Semantic Infrastructure 20

6.1.3 Use Case – Export Model from UML Modeling Tool 20

6.1.4 Use Case – Import Model in UML Modeling Tool 20

6.2 Use Case – Develop Pluggable Generator 20

6.2.1 Use Case – Develop Generator Template 20

6.2.2 Use Case – Integrated Template in the Generator Framework 20

6.2.3 Use Case – Build the Generator Application 20

6.2.4 Use Case – Deploy the Generator Application 20

6.3 Use Case – Manage Service Development Kit 20

6.3.1 Use Case – Install Service Development Kit 20

6.3.2 Use Case – Update Service Development Kit 20

6.3.3 Use Case – Remove Service Development Kit 20

6.3.4 Use Case – Install plug-in into Service Development Kit 20

6.3.5 Use Case – Update plug-in into Service Development Kit 20

6.3.6 Use Case – Remove plug-in from Service Development Kit 20

6.4 Use Case – Develop Service 20

6.4.1 Use Case – Import/Load UML Model 21

6.4.2 Use Case – Update Model Components/Properties 21

6.4.3 Use Case – Generate Code 21

6.4.4 Use Case – Synchronize Code 21

6.5 Use Case – Develop Service Client 21

6.5.1 Use Case – Import/Load WSDL Document 21

6.5.2 Use Case – Generate Client 21

6.6 Use Case – Perform Model Mapping 21

6.6.1 Use Case – Import/Load UML Model 21

6.6.2 Use Case – Update Model Components/Properties 21

6.6.3 Use Case – Map Model Components (Primitive Object to Database) 21

6.6.4 Use Case – Map Model Components (Primitive Object to ISO 21090 Object) 21

6.6.5 Use Case – Map Model Components (ISO 21090 Object to Database) 21

6.6.6 Use Case – Map Model Components (Primitive Objects to Primitive XSD) 21

6.6.7 Use Case – Map Model Components (Primitive Object to ISO 21090 XSD) 21

6.7 Use Case – Integration with Semantic Infrastructure 21

6.7.1 Use Case – Locate Semantic Infrastructure Repository 21

6.7.2 Use Case – Import Existing Models/Components from Semantic Infrastructure 21

6.7.3 Use Case – Perform Semantic Annotation with Existing Metadata 21

6.7.4 Use Case – Create New Semantic Metadata in Local Repository 21

6.7.5 Use Case – Publish Semantic Metadata to Semantic Infrastructure 21

6.7.6 Use Case – Pull Semantic Metadata updates from Semantic Infrastructure 21

6.8 Use Case – Enterprise Service 21

6.8.1 Use Case – Query Service 22

6.8.2 Use Case – Serialize/De-Serialize Results 22

6.8.3 Use Case – Login into the Service 22

6.8.4 Use Case – Query Secured Operations 22

6.9 Use Case – Deployment 22

6.9.1 Use Case – Deploy Service 22

6.9.2 Use Case – Install Client 22

7. System Use Cases 22

7.1 Use Case – Model Management 22

7.1.1 Use Case – Model Conversion 22

7.1.2 Use Case – Model Load 22

7.1.3 Use Case – Model Update 22

7.1.4 Use Case – Model Search 22

7.2 Use Case – Develop Pluggable Generator 22

7.3 Use Case – Manage Service Development Kit 22

7.4 Use Case – Develop Service 22

7.5 Use Case – Develop Service Client 22

7.6 Use Case – Perform Model Mapping 22

7.6.1 Use Case – Data Transformation 23

7.7 Use Case – Integration with Semantic Infrastructure 23

7.7.1 Use Case – Discover Semantic Infrastructure Repositories 23

7.7.2 Use Case – Create/Search/Update/Delete Metadata in Local Repository 23

7.8 Use Case – Enterprise Service 23

7.8.1 Use Case – Report Service Identity and Metadata to Service Registry 23

7.8.2 Use Case – Query Translation 23

7.8.3 Use Case – Query Execution 23

7.8.4 Use Case – Serialization/De-Serialization 23

7.8.5 Use Case – Authentication 23

7.8.6 Use Case – Authorization 23

7.9 Use Case – Deployment 23

7.9.1 Use Case – Setup Environment 23

7.9.2 Use Case – Verify Environment 23

7.9.3 Use Case – Deploy System 23

8. Requirements Traceability Matrix (RTM) 24

9. Software Overview 25

9.1 Enterprise Service 25

9.2 Service Development Tool 26

9.2.1 Graphical User Interface (GUI) 26

9.2.2 Code Generator 27

9.2.3 Mapping and Transformation 27

9.2.4 Semantic Metadata Management 28

9.2.5 Deployment 28

10. Architecture 29

10.1 Technical Architecture 29

10.1.1 Technical Architecture – Enterprise Service 29

10.1.2 Technical Architecture – Service Development Tooling 30

10.1.3 Technical Architecture – Model Representation 30

10.1.4 Technical Architecture - Code Generator 31

10.1.1 Technical Architecture – Service Assembler 32

10.1.2 Technical Architecture – Service Synchronizer 32

10.1.3 Technical Architecture – Semantic Metadata Management 32

10.1.4 Technical Architecture – Model Mapping 32

10.1.5 Technical Architecture – Deployment 32

10.2 Deployment Architecture 32

10.2.1 Deployment Architecture – Enterprise Service 33

10.2.2 Deployment Architecture - Service Development Tool 33

11. Information Models 34

11.1 Logical Model 34

11.2 Data Model 34

12. Technical Design 35

12.1 Graphical User Interface 35

12.1.1 Overview and design 35

12.1.2 High-level class diagrams 41

12.2 Code Generator 45

12.2.1 Challenges 45

12.2.2 Code Generator Design 47

13. Security Design 57

14. Testing Considerations 58

15. Deployment Considerations 59

Table of Figures

[Figure 1: Governance Structure 13](#_Toc274217863)

[Figure 2: Data Service Architecture 16](#_Toc274217864)

[Figure 3: Analytical Service Architecture 16](#_Toc274217865)

[Figure 4: Service Development Tool Architecture 26](#_Toc274217866)

[Figure 5: ISO 21090 Enabled Service Components 27](#_Toc274217867)

[Figure 6: Enterprise Service Architecture 29](file:///C:\main\sdk\Hydra%20-%20Design.docx#_Toc274217868)

[Figure 7: Model Representation 30](#_Toc274217869)

[Figure 8: Code Generator Architecture 31](#_Toc274217870)

[Figure 9: Service Assembler Architecture 32](#_Toc274217871)

[Figure 10: Model Conversion Interface 36](#_Toc274217872)

[Figure 11: Model Explorer Interface 37](#_Toc274217873)

[Figure 12: Model Property Browser Interface 38](#_Toc274217874)

[Figure 13: Model Relationship Browser Interface 38](#_Toc274217875)

[Figure 14: Model Operations Browser Interface 39](#_Toc274217876)

[Figure 15: Data Model Browser Interface 39](#_Toc274217877)

[Figure 16: Model Properties Browser Interface 40](#_Toc274217878)

[Figure 17: Generator Interface 40](#_Toc274217879)

[Figure 18: Logical Model of IDE 42](#_Toc274217880)

[Figure 19: IDE - Converter 43](#_Toc274217881)

[Figure 20: IDE - Model Explorer 44](#_Toc274217882)

[Figure 21: IDE - Generator 45](#_Toc274217883)

[Figure 22: Sample development workflow involving model translation and artifact generation. 46](#_Toc274217884)

[Figure 23: Template approach 47](#_Toc274217885)

[Figure 24: Model Validation Approach 49](#_Toc274217886)

[Figure 25: Code Generator Framework Architecture 50](#_Toc274217887)

[Figure 26: Code Generator Component Class Diagram 52](#_Toc274217888)

[Figure 27: Code Generator Workflow 53](#_Toc274217889)

[Figure 28: JSR 223 Based Scripting Architecture 55](#_Toc274217890)

# Introduction

## Document background and intent

This document is an accumulation of documentation and information created while researching and prototyping usability enhancements to the caCORE Software Development Kit. It is intended as a live document that will be updated and expanded as work progresses and includes stubs for areas that still need to be considered and addressed (or were considered but not in depth).

Areas of coverage range from background information on the current solution all the way through technical design of the current implementation. As the document grows, it may need to be reworked into several documents for ease of use. Possible documents that could be derived include:

Overview: sections 3 – goal and objectives, 4 – background, 9 – software overview

Requirements and use cases: sections 3 – goal and objectives, 4 – background, 5 – product requirements, 6 – use cases, 7 – system use cases, 8 – requirements traceability matrix

Architecture and design: 9 – software overview, 10 – architecture, 11 – information models, 12 – technical design, 13 – security design

Test: 14 – test considerations

Deployment: 15 – deployment consideration

## Intended Audience

This document is intended for Product Managers, Software Architects and Software Developers who intend to learn the internals of the software. This document contains high level information that may be of interest to the end users of the product but the document itself is not intended to cater them.

## Purpose of the Document

The purpose of this document is to provide high level as well as detailed view of the Hydra software. The document discusses major components of the system and their needs as well as roles, the product requirements and their implementation details and usage instructions.

## Organization of the Document

* **Section**  – provides information on the stakeholders and how the software implementation is governed
* **Section**  – provides brief description on the goals and objectives of the project
* **Section**  – provides background and history information on the project
* **Section**  – provides list of the major technical requirements that are considered for the software implementation
* **Section –** provides use cases from the user’s perspective
* **Section –** provides use cases from the system’s perspective
* **Section**   **–** provides cross-reference information between the requirements and the respective use cases
* **Section –** provides details on the business functionality and the features of the software that the end users are interested in
* **Section**  – provides technical description on the major building blocks of the system and how these building blocks are used to meet the objectives
* **Section**  - provides details on the information model used in the system and their relevance to the different parts of the application
* **Section**  – provides detailed technical approach implemented in the system
* **Section**  – provides information on security implementation and security policy configuration
* **Section**  – provides high level information on how the product should be tested. Detail testing procedures are listed in the QA test plan
* **Section**  – provides details about the deployment requirements for the software and installation steps. Detailed user oriented installation steps are listed in the installation and usage guide.

# Stakeholders

|  |  |  |
| --- | --- | --- |
| **Name** | **Role** | **Comment** |
| NCI CBIIT Management | Provide guidance and vision of the product. Direct the development team to develop the product functionalities. |  |
| SAIC-F | Provide guidance and vision of the product. Direct the development team to develop the product functionalities. |  |
| caCORE SDK Users | Use the software product, provide feedback, request features |  |
| caGrid SDK Users | Use the software product, provide feedback, request features |  |
| Other caBIG Users | Use the software product, provide feedback, request features |  |
| Non caBIG Users | Use the software product, provide feedback, request features |  |
|  |  |  |
| Development Team | Develop software under direction from NCI CBIIT management and SAIC-F |  |
| QA Team | Verify software implementation against the requirements |  |
| Documentation Team | Assist in developing the product documentation |  |
| Training Team | Assist in developing the product training material |  |
| Systems Team | Provide support in deploying the product on NCICB Tiers. Provide access to required software hosting and software development environment |  |
| BDA Team | Assist development team in automating software builds and deploying the software on the NCICB tiers |  |

## Implementation Governance



Figure : Governance Structure

The software development is done by the development team based on the input received from the user community. The user community provides feedback to the Governance team and the Product management team (which includes SAIC-F COTR and product manager). The Governance team works with the users and recommends change in the product policy to the product management team. The product management team drives the implementation based on the policy and the features input received. This user driven tool development process enhances the usability of the product and yield in greater adoption with better user experience.

# Goal and Objectives

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

**Integrated Service Development Tooling:** A service development tool is needed that is capable of supporting the service creation, service deployment, and metadata management capabilities. 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 technologies. We need to provide a clear migration path to bring these users to the new infrastructure tools.

# Background

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.

In the caBIG eco-system there are many systems that works together to provide the core capabilities. The primary systems that make up the eco-system are

1. Semantic Infrastructure
2. caGrid Platform
3. Security Infrastructure
4. Enterprise Services

To support the creation of the enterprise services based on the major systems listed above, we need to develop sophisticated tooling.

## Current Solution/Current State

The NCI CBIIT and caBIG provides several tools which are used in specific combination to develop the required service capabilities. The last release of the software bundle 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

Under the current solution, the user has to use the above mentioned tools in a specific combination in order to accomplish the task of developing and deploying semantically aware services.

### 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.

## Problem statement

The implementation of the current generation of the service development tool provides an integrated solution to create the analytical and data service. The previous generation infrastructure tools had several limitations and issues; primary issues being the following:

1. Difficulty of understand and use the tools and technology
2. Limited functionality available to the users
3. Does not provide iterative development support

The infrastructure development team is taking the lessons learned from the past generation and combining it with the new requirements to define the roadmap of the next generation of infrastructure. As part of the roadmap development, the roadmap team is identifying the capabilities which need to be converted in a concrete toolset. Following capabilities can be grouped into a unified tool which can help user in the service development life cycle.

* **Capability 1:** Model Driven Development
* **Capability 2:** Service Creation
* **Capability 3:** Data Transformation
* **Capability 4:** Semantic Annotations
* **Capability 5:** ISO 21090 Datatype Support
* **Capability 6:** Enterprise Service
* **Capability 7:** Service Deployment

# Product Requirements

* **Requirement 1:** 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.
* **Requirement 2:** 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.
* **Requirement 3:** 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.
* **Requirement 4:** 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.
* **Requirement 5:** 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
* **Requirement 6:** 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.
* **Requirement 7:** 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.
* **Requirement 8:** 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.
* **Requirement 9:** 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.
* **Requirement 10:** 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.
* **Requirement 11:** 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.
* **Requirement 12:** 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.

# Business Use Cases

This section lists use cases that were considered during development. Since the development was not strictly use-case driven and the work, while re-usable, was a proof of concept, the use cases were not fleshed out in detail. The inclusion of a use case means it was considered but doesn’t necessarily mean it has been implemented to date.

## Use Case – Manage Model

### Use Case – Create/Update Model

### Use Case – Import Existing Models/Components from Semantic Infrastructure

### Use Case – Export Model from UML Modeling Tool

### Use Case – Import Model in UML Modeling Tool

## Use Case – Develop Pluggable Generator

### Use Case – Develop Generator Template

### Use Case – Integrated Template in the Generator Framework

### Use Case – Build the Generator Application

### Use Case – Deploy the Generator Application

## Use Case – Manage Service Development Kit

### Use Case – Install Service Development Kit

### Use Case – Update Service Development Kit

### Use Case – Remove Service Development Kit

### Use Case – Install plug-in into Service Development Kit

### Use Case – Update plug-in into Service Development Kit

### Use Case – Remove plug-in from Service Development Kit

## Use Case – Develop Service

### Use Case – Import/Load UML Model

### Use Case – Update Model Components/Properties

### Use Case – Generate Code

### Use Case – Synchronize Code

## Use Case – Develop Service Client

### Use Case – Import/Load WSDL Document

### Use Case – Generate Client

## Use Case – Perform Model Mapping

### Use Case – Import/Load UML Model

### Use Case – Update Model Components/Properties

### Use Case – Map Model Components (Primitive Object to Database)

### Use Case – Map Model Components (Primitive Object to ISO 21090 Object)

### Use Case – Map Model Components (ISO 21090 Object to Database)

### Use Case – Map Model Components (Primitive Objects to Primitive XSD)

### Use Case – Map Model Components (Primitive Object to ISO 21090 XSD)

## Use Case – Integration with Semantic Infrastructure

### Use Case – Locate Semantic Infrastructure Repository

### Use Case – Import Existing Models/Components from Semantic Infrastructure

### Use Case – Perform Semantic Annotation with Existing Metadata

### Use Case – Create New Semantic Metadata in Local Repository

### Use Case – Publish Semantic Metadata to Semantic Infrastructure

### Use Case – Pull Semantic Metadata updates from Semantic Infrastructure

## Use Case – Enterprise Service

### Use Case – Query Service

### Use Case – Serialize/De-Serialize Results

### Use Case – Login into the Service

### Use Case – Query Secured Operations

## Use Case – Deployment

### Use Case – Deploy Service

### Use Case – Install Client

# System Use Cases

This section lists use cases that were considered during development. Since the development was not strictly use-case driven and the work, while re-usable, was a proof of concept, the use cases were not fleshed out in detail. The inclusion of a use case means it was considered but doesn’t necessarily mean it has been implemented to date.

## Use Case – Model Management

### Use Case – Model Conversion

### Use Case – Model Load

### Use Case – Model Update

### Use Case – Model Search

## Use Case – Develop Pluggable Generator

## Use Case – Manage Service Development Kit

## Use Case – Develop Service

## Use Case – Develop Service Client

## Use Case – Perform Model Mapping

### Use Case – Data Transformation

## Use Case – Integration with Semantic Infrastructure

### Use Case – Discover Semantic Infrastructure Repositories

### Use Case – Create/Search/Update/Delete Metadata in Local Repository

## Use Case – Enterprise Service

### Use Case – Report Service Identity and Metadata to Service Registry

### Use Case – Query Translation

### Use Case – Query Execution

### Use Case – Serialization/De-Serialization

### Use Case – Authentication

### Use Case – Authorization

## Use Case – Deployment

### Use Case – Setup Environment

### Use Case – Verify Environment

### Use Case – Deploy System

# Requirements Traceability Matrix (RTM)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Requirement** | **Capability** | **Business Use Case** | **System Use Case** | **Comment** |
| Requirement 1 | Capability 1 | 6.1, 6.4, 6.6, 6.7 | 7.1, 7.4, 7.6, 7.7 |  |
| Requirement 2 | Capability 2 | 6.4, 6.6, 6.7, 6.9 | 7.4, 7.6, 7.7, 7.9 |  |
| Requirement 3 | Capability 2 | 6.2, 6.5 | 7.2, 7.5 |  |
| Requirement 4 | Capability 2 | 6.5 | 7.5 |  |
| Requirement 5 | Capability 3 | 6.6 | 7.6 |  |
| Requirement 6 | Capability 3 | 6.6 | 7.6 |  |
| Requirement 7 | Capability 4 | 6.7 | 7.7 |  |
| Requirement 8 | Capability 5 | 6.8 | 7.8 |  |
| Requirement 9 | Capability 5 | 6.8 | 7.8 |  |
| Requirement 10 | Capability 6 | 6.8 | 7.8 |  |
| Requirement 11 | Capability 6 | 6.8 | 7.8 |  |
| Requirement 12 | Capability 7 | 6.9 | 7.9 |  |

# Software Overview

Based on the identified requirements and use cases, we believe that the service development tooling will be developed in following major areas: 1) Enterprise Service, and 2) Service development tool. The details of these areas are provided below.

## Enterprise 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. There are two types of services that the infrastructure tool will support: 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. The common components of the service recognize and support the ISO 21090 datatypes. For example, the CQL as a query language can specify the query based on the domain objects using the primitive data types as well as domain objects based on the ISO 21090 datatypes. Similarly, the fine grained authorization implementation recognizes attributes of the ISO 21090 datatypes and allows security policy configuration against that.

The enterprise service contains many small modules including

1. Service skeleton
2. Security (authentication and authorization touch points)
3. Audit integration points
4. Code generated from the model
5. Data transformation functions
6. Querying function
7. Message/Data Serialization/De-Serialization functions

With exception of the artifacts generated from the model, all modules are pre-developed. The service development tool generates the code from the model and combines other modules to produce a complete working system. The custom code provided by the end user is integrated with the finished service as well.

**Note:** 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.

## Service Development Tool



Figure : Service Development Tool Architecture

Service development tool is the infrastructure component which allows the users to build the SAIF/ECCF compliant enterprise services. It provides all the functionalities necessary to meet needs of a wide array of users. The core areas of the service development tool components are:

1. Graphical User Interface (GUI)
2. Code Generator
3. Mapping and Transformation
4. Semantic Metadata Management
5. Deployment

These core functions of the service development tool provide ability to meet all the needed capabilities.

### 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

### Code Generator

The core of the service development tool is a Model Driven Architecture (MDA) oriented code generation 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 allows the users to generate various types of artifacts. The service development tool provides many different types of generators out of the box including Java beans, ECCF documents, XSDs, Model Reports etc. The user can develop and plug-in additional generators to produce the artifacts that are not supported out of the box. The artifacts produced by the core engine are used to build and support the enterprise service.

### 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 possible combinations in which the information can be converted into ISO 21090 datatype representation format. The data 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.

The mapping of the models is supported by very sophisticated GUI. Using a robust interface, the user can load the model and map the model components with another model. The user chooses 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. 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. With the service development tool, the developers can annotate the model and the source code with the semantic metadata. We provide following functionalities in the service development area for this purpose:

* Locate the semantic metadata repositories
* Retrieve models and sections of the models from the semantic metadata repositories
* Retrieve semantic metadata for the new model elements from the public repository
* Create new metadata whenever the semantic metadata do not exist in the public repository and store this in the “local-metadata” repository
* Push the newly created metadata from local repository to the public repository
* Get updates on the already used semantic metadata

Using these features, the developers can 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 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 using the cloud based deployment capabilities. The service development tool produces the service, which can be deployed with its deployment capabilities out of the box. The BDA scripts are utilized to manage the deployment functions. An additional GUI on top of the BDA scripts is provided so that the users of the developed service can easily deploy these services in their environment using few simple clicks.

# Architecture

## Technical Architecture

### Technical Architecture – Enterprise Service

Figure : Enterprise Service Architecture

The enterprise service is a Apache CXF based web service. Our service differs from the simple web service due to its inherit nature to support semantics over the grid environment. The service registers itself in the service registry and keeps reporting its status to the registry on periodic basis. The users of the grid discover information about the service by querying the service registry. The domain elements of the service are semantically described using the Enterprise Vocabulary from EVS. This semantically annotated service’s metadata are registered in the Semantic Metadata Registry. The users can query the semantic metadata registry at the runtime and obtain the static as well as behavioral semantics information about the service. The Service Client and the Service itself communicates information using XML based messages which is validated against the XSD registered in the semantic metadata repository.

The service itself relies on the user provided implementation to complete the service operation. The service API can perform application level authorization against its preferred security database and execute the necessary operations when allowed by its security policy.

### Technical Architecture – Service Development Tooling

The service development tooling is responsible for creating the service described in the earlier sub-section. The tool comprise of various components with capabilities including 1) code generator 2) model mapping 3) semantic metadata management 4) service assembler and 5) graphical user interface. The tool itself is based on the OSGI styled plug-in architecture of Eclipse IDE. The graphical user interface is used to control the functionality provided by individual plug-ins. Additional functionality is added in this Eclipse IDE using separate plug-in followed by GUI enhancements to facilitate the newly added functionality.

### Technical Architecture – 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:

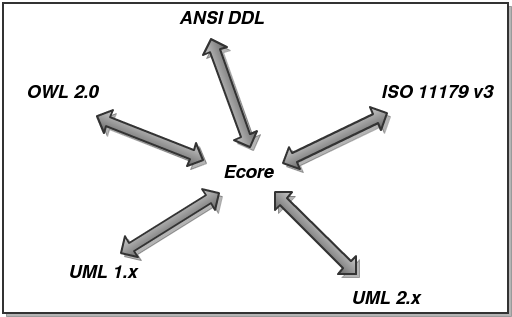


Figure : Model Representation

The EMF Ecore model format is used as a candidate for the new artifact generator’s intermediate form.   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 code generator.  The Ecore model can be represented in other meta modeling formats such as OWL.  This allow us to use this model outside of the Java world should there be a need to do so.

### Technical Architecture - Code Generator

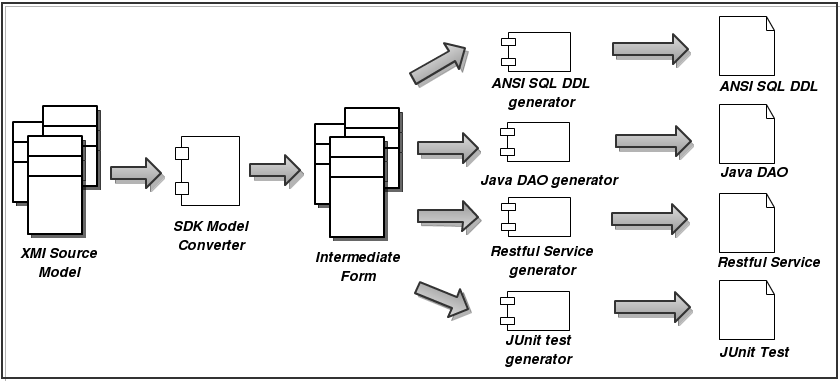


Figure : Code Generator Architecture

In this architecture, a domain model represented in XMI is translated by an XMI specific 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 our development team. Along the same lines subsequent versions of the generated Android 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.

### Technical Architecture – Service Assembler



Figure : Service Assembler Architecture

The service assembler is responsible for assembling various components together to create a functional service. In the service the functional components includes (but not limited to) service skeleton, WSDL file, XSDs, POJO beans, Configuration files and business components. Some of these components will be generated from the model while the others will be developed by the users and provided to the assembler. For most cases the service assembler is as simple as simple build script which puts various files together in an archive format (e.g. war file) as understood by the application server.

### Technical Architecture – Service Synchronizer

***This section is to be considered in a future release.***

### Technical Architecture – Semantic Metadata Management

***This section is to be considered in a future release.***

### Technical Architecture – Model Mapping

***This section is to be considered in a future release.***

### Technical Architecture – Deployment

***This section is to be considered in a future release.***

## Deployment Architecture

***This section is to be considered in a future release.***

### Deployment Architecture – Enterprise Service

***This section is to be considered in a future release.***

### Deployment Architecture - Service Development Tool

***This section is to be considered in a future release.***

# Information Models

***This section is to be considered in a future release.***

## Logical Model

***This section is to be considered in a future release.***

## Data Model

***This section is to be considered in a future release.***

# Technical Design

## Graphical User Interface

### Overview and design

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:

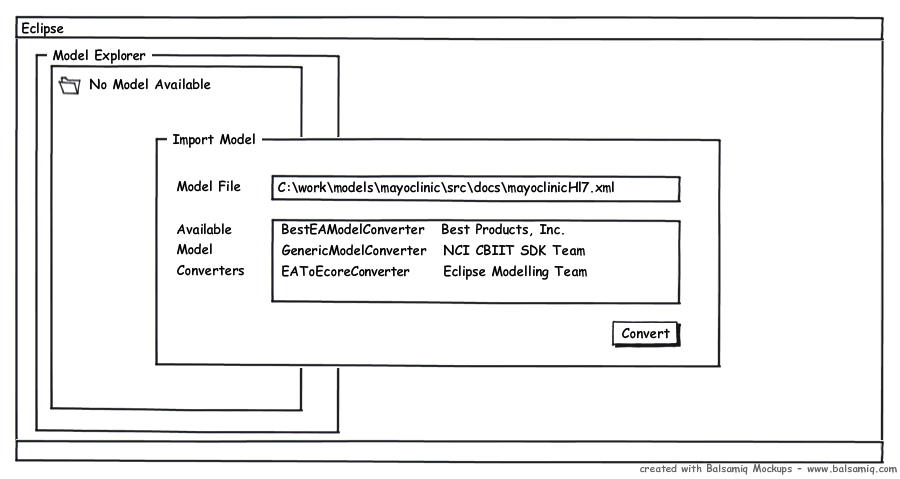


Figure : Model Conversion Interface

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:

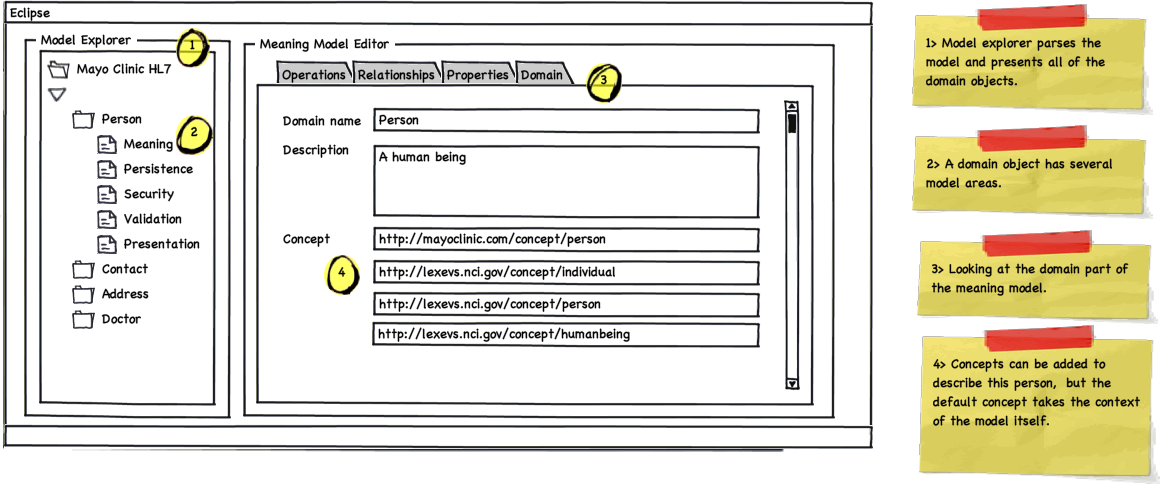


Figure : Model Explorer Interface

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:

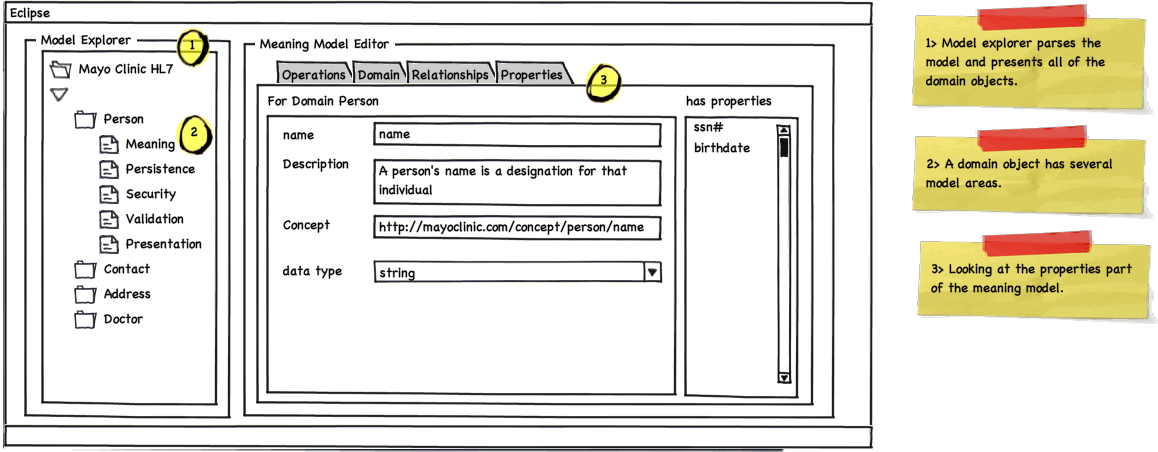


Figure : Model Property Browser Interface

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:

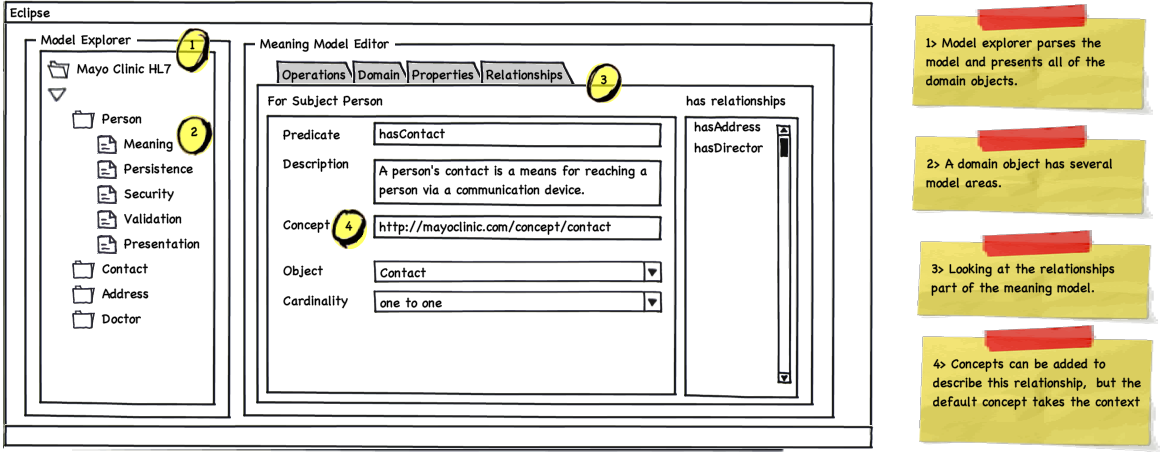


Figure : Model Relationship Browser Interface

and the operations tab:

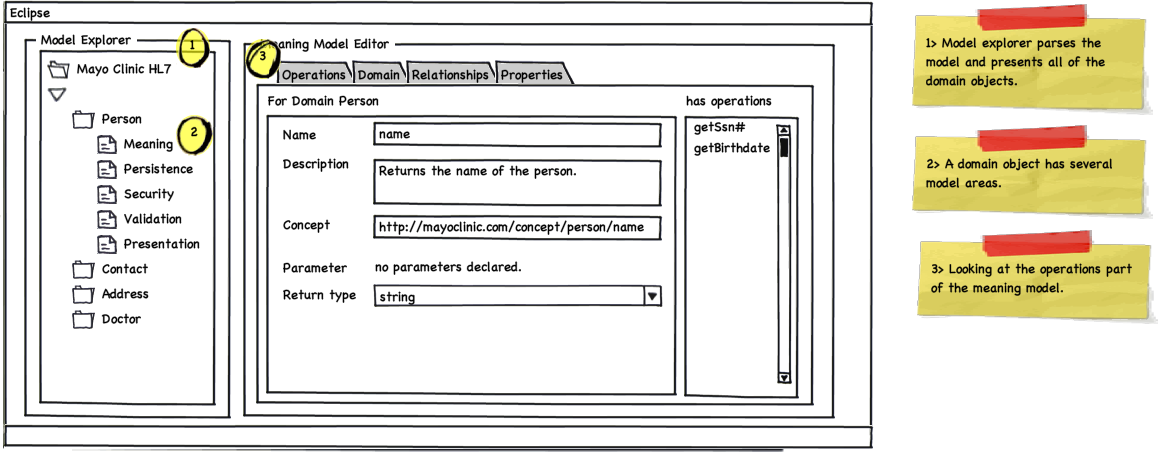


Figure : Model Operations Browser Interface

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.

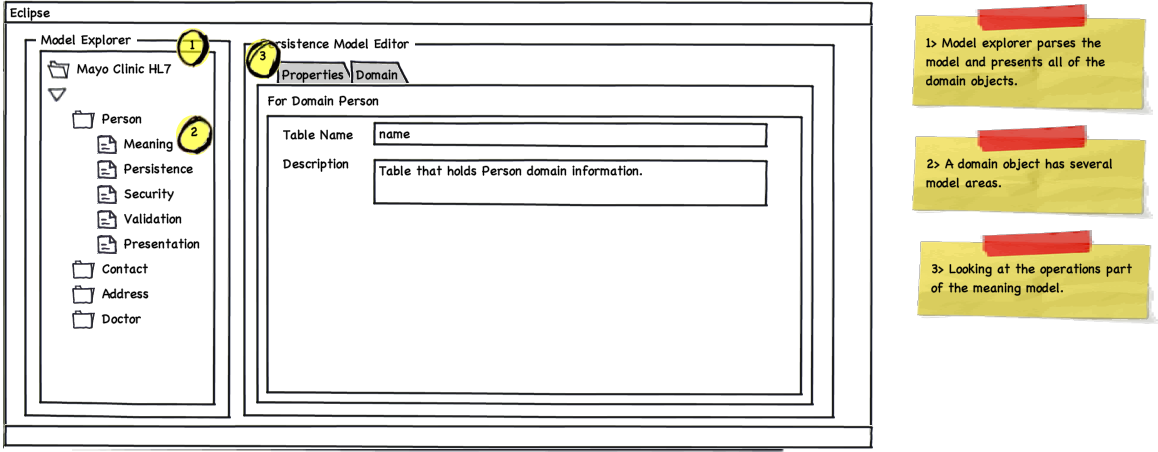


Figure : Data Model Browser Interface

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

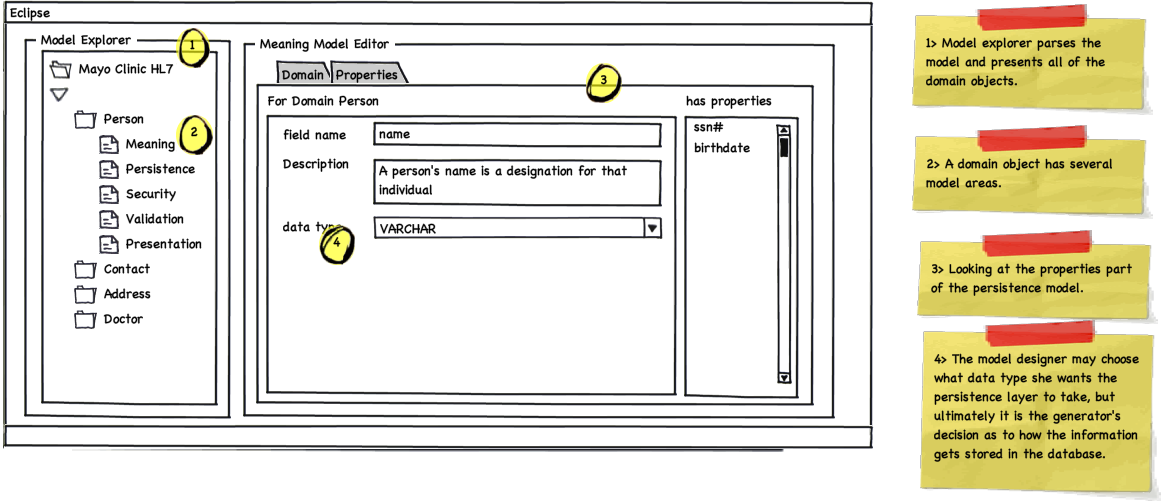


Figure : Model Properties Browser Interface

**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.

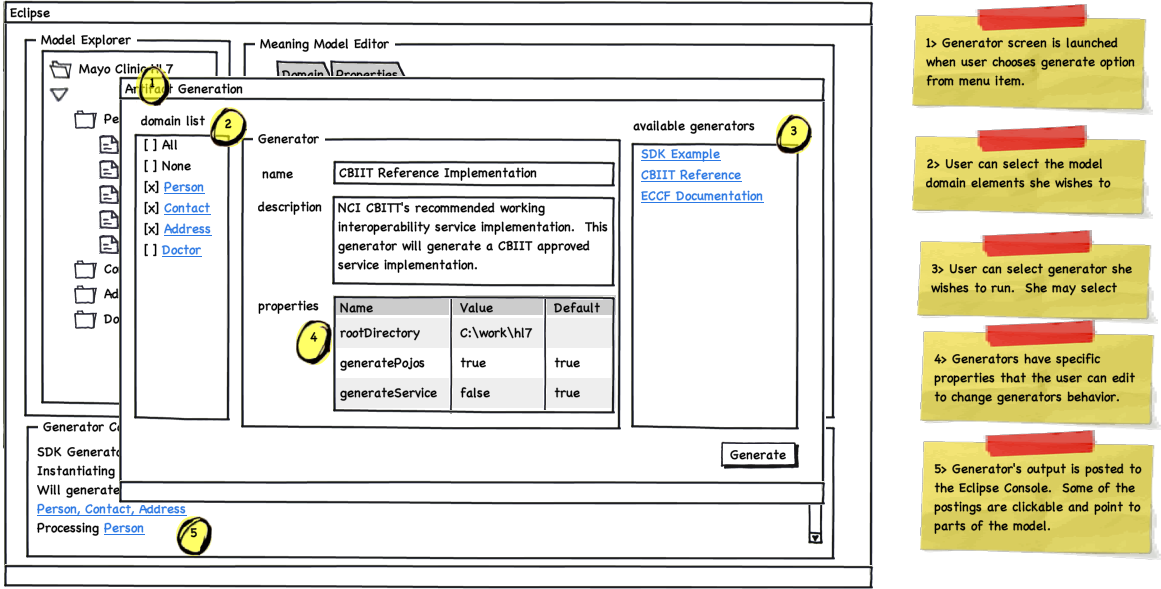


Figure : Generator Interface

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.

### High-level class diagrams

These class diagrams provide a high-level view of the design of key sections. Not all classes are included.



Figure : Logical Model of IDE



Figure : IDE - Converter



Figure : IDE - Model Explorer



Figure : IDE - Generator

## Code Generator

### 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-1), 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-2), 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.

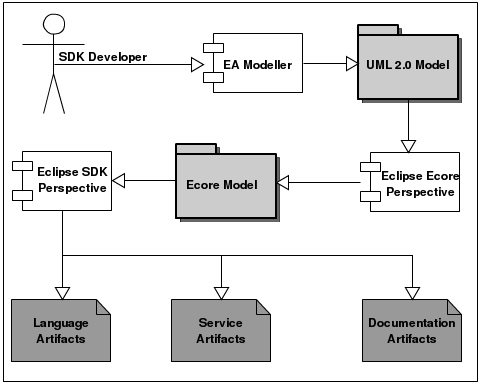


Figure : Sample development workflow involving model translation and artifact generation.

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 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. Even though the the Eclipse Modeling Framework (EMF) project uses JET (Java Emitter Templates) and JMerge (Java Merge) to generate code, the Hydra team decided on using StringTemplate. StringTemplate boasts a powerful templating syntax which enforces strict separation of model, view and controller. As such, templates created with StringTemplate are easily reused. StringTemplate can be used to generate SQL, XML, or Java source code, as well as other output types like documentation, deployement scripts, or property files. It is located in the stringtemplate.org as a part of the ANTLR compiler suite.

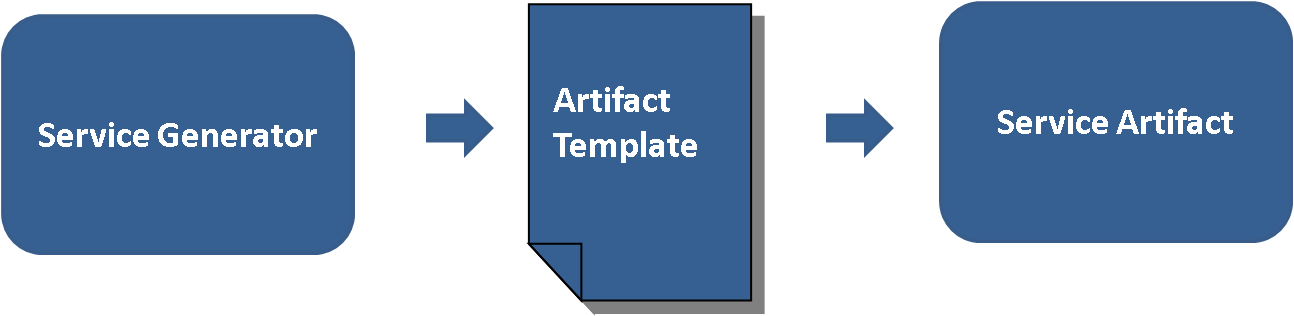


Figure : Template approach

Following is an example of a StringTemplate file contents used to generate a Java pojo for use in Apache CXF.

|  |
| --- |
| package $packageName$;  import java.io.Serializable;  $importSt;separator=";"$  public class $className$ implements Serializable{  /\*\*  \* An attribute to allow serialization of the domain objects  \*/  private static final long serialVersionUID = 1234567890L;    $pojoAttribute; separator="\n"$  $pojoOperation; separator="\n"$  } |

The code below shows how to invoke the template instance.

|  |
| --- |
| StringTemplateGroup stringTemplateGroup = new StringTemplateGroup(“hydra”);  stringTemplate = stringTemplateGroup.getInstanceOf(“pojo”);  stringTemplate.setAttribute(“className”, “Person”);  stringTemplate.setAttribute(“pojoAttribute”, “public String name;”);  stringTemplate.setAttribute(“pojoOperation”, public String getName() { return name; } |

This prints the following Java result to the console:

|  |
| --- |
| public class Person implements Serializable{  /\*\*  \* An attribute to allow serialization of the domain objects  \*/  private static final long serialVersionUID = 1234567890L;    public String name;  public String getName() { return name; }  } |

A more in depth treatment of stringtemplate can be found at <http://www.antlr.org/wiki/display/ST/Five+minute+Introduction>

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.

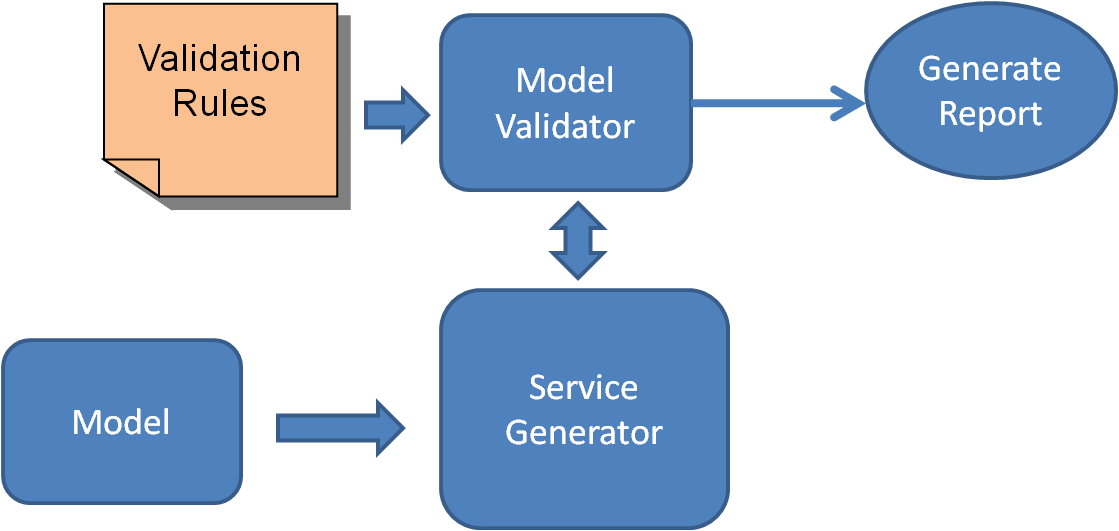


Figure : Model Validation Approach

#### 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.

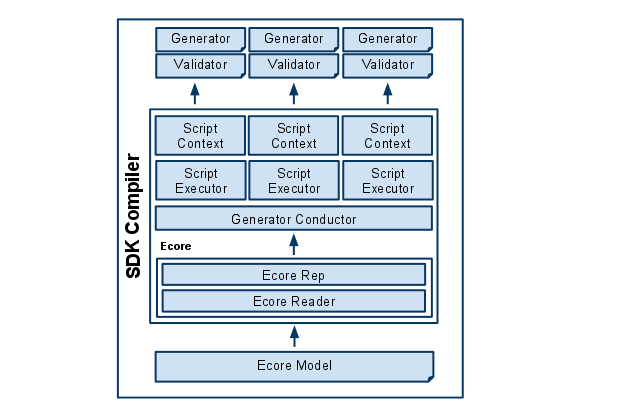


Figure : Code Generator Framework Architecture

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.

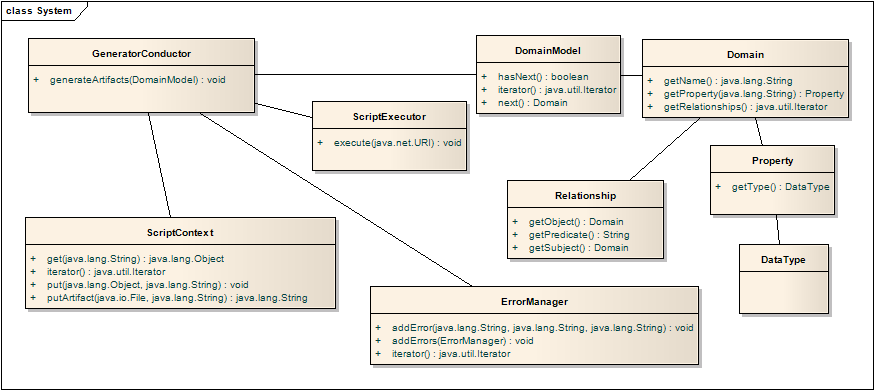


Figure : Code Generator Component Class Diagram

**Pluggable Generator Layout**

**<generator name>**

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

-> info.properties (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:

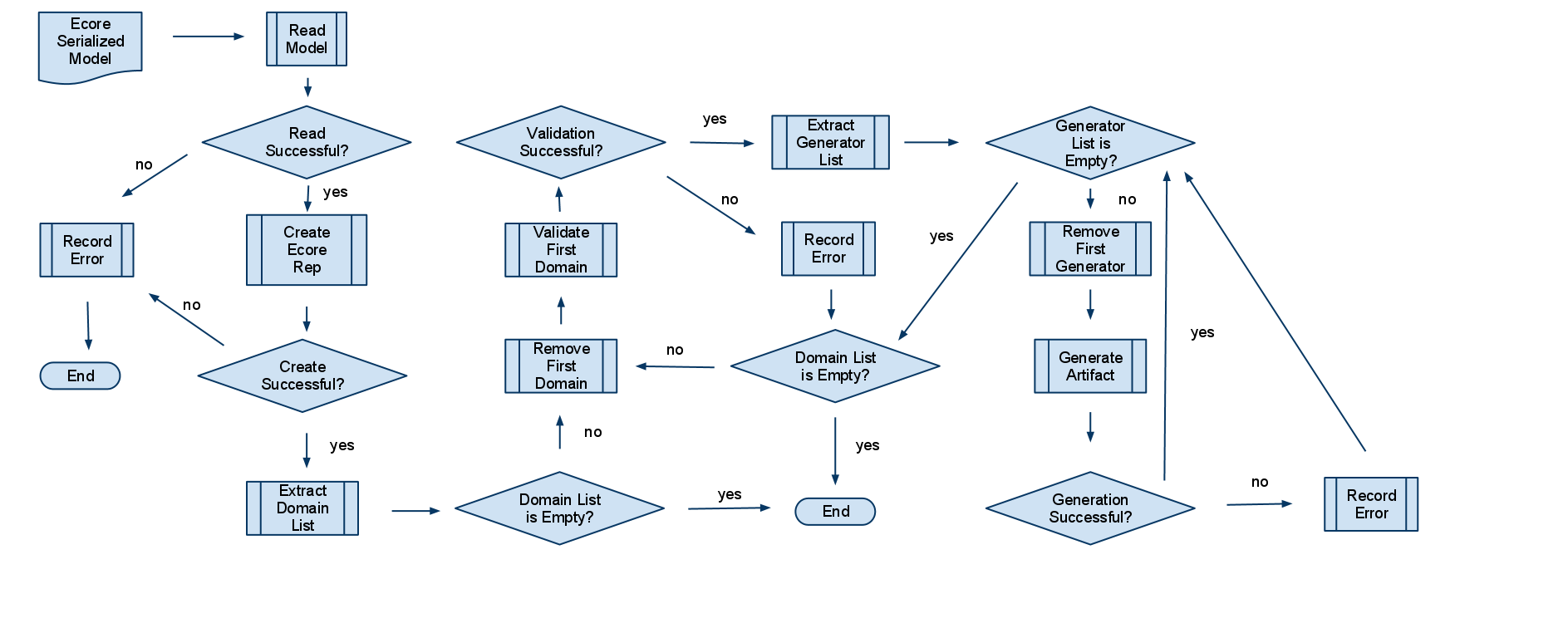


Figure : Code Generator Workflow

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 affected 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. By decoupling models and controllers, generator code reuse is encouraged.

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

JSR-223 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-223 scripting language support is quite impressive, with scripting engines available for Java, JavaScript, Python, Ruby, Perl, and others[[3]](#footnote-3). 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.

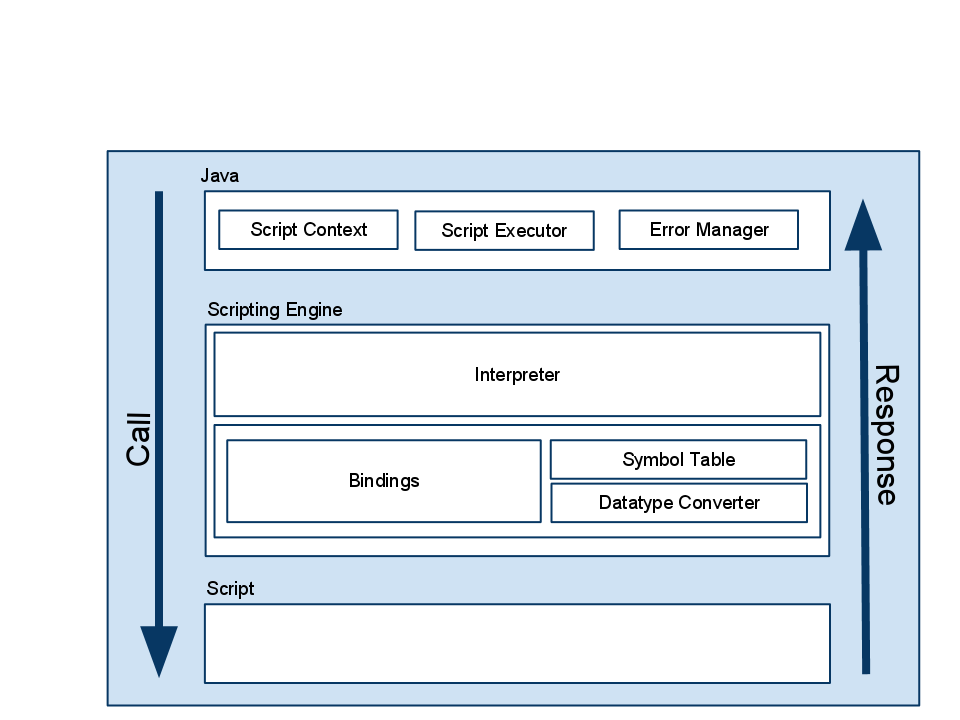


Figure : JSR 223 Based Scripting Architecture

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.

# Security Design

# Testing Considerations

***This section is to be considered in a future release.***

# Deployment Considerations

***This section is to be considered in a future release.***

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-1)
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-2)
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-3)