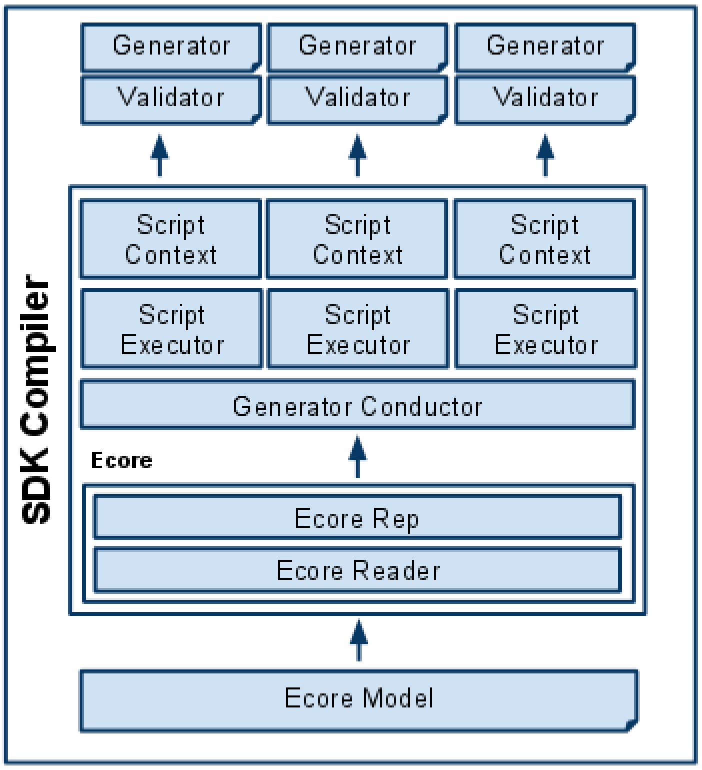
Introduction

This document provides an architectural assessment of the SDK generator. Readers of this document will gain an understanding of the components of the SDK generator and their interaction through the generation process. Technology considerations are limited to the Java SDK platforms 1.5 and up, as they are supporters of the critical Java scripting support capabilities that the SDK generator will depend on. Support for the JDK 1.4 environment is not required.

Principal Design Components

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.



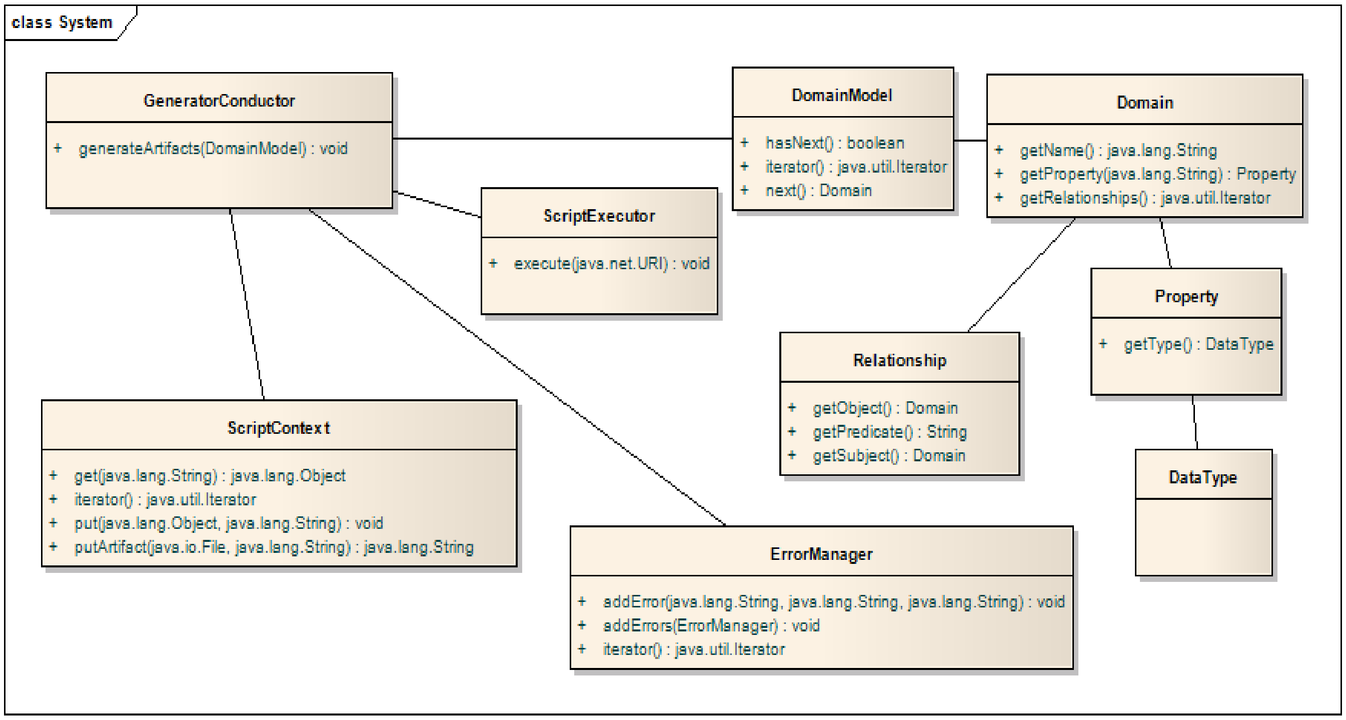
This figure illustrates the design components that make up the new SDK generator.

- **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.


class System
GeneratorConductor
+ generateArtifacts(DomainModel) : void
DomainModel
+ hasNext() : boolean
+ iterator() : java.util.Iterator
+ next() : Domain
Domain
+ getName() : java.lang.String
+ getProperty(java.lang.String) : Property
+ getRelationships() : java.util.Iterator
ScriptExecutor
+ execute(java.net.URI) : void
Property
+ getType() : DataType
Relationship
+ getObject() : Domain
+ getPredicate() : String
+ getSubject() : Domain
ScriptContext
+ get(java.lang.String) : java.lang.Object
+ iterator() : java.util.Iterator
+ put(java.lang.Object, java.lang.String) : void
+ putArtifact(java.io.File, java.lang.String) : java.lang.String
DataType
ErrorManager
+ addError(java.lang.String, java.lang.String, java.lang.String) : void
+ addErrors(ErrorManager) : void
+ iterator() : java.util.Iterator

Pluggable Generator Layout

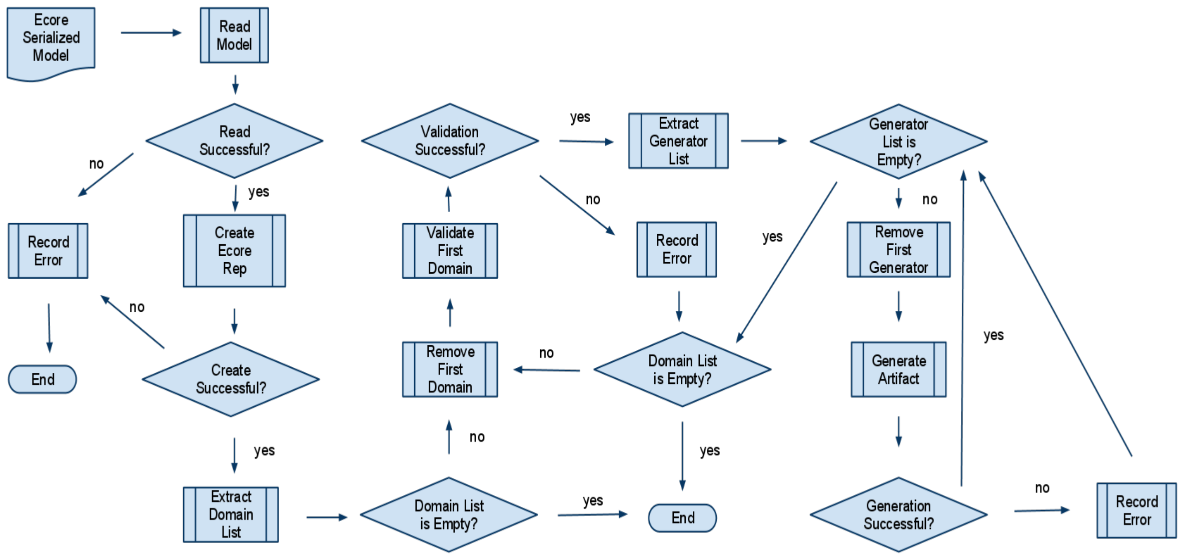
<generator name>

-> 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:



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.

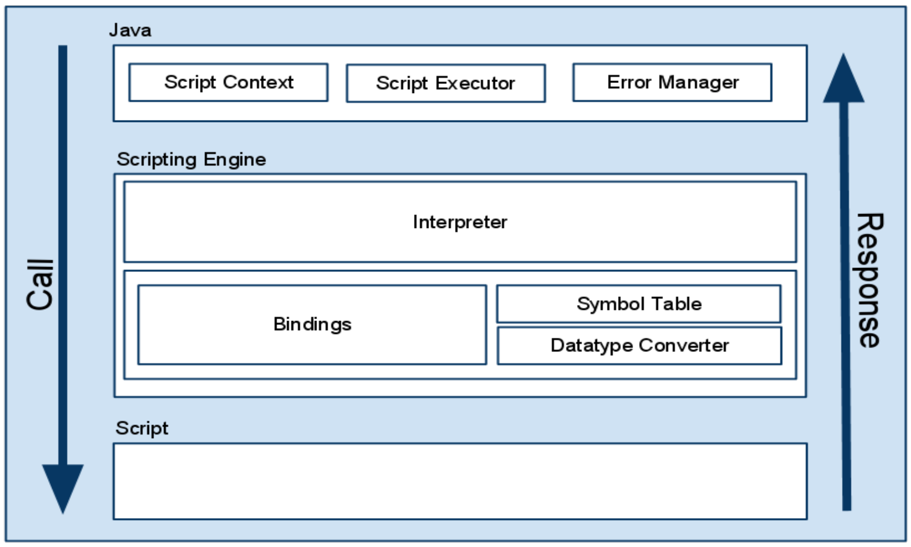
StringTemplate brings an added benefit of encouraging a clear separation of model and controller code. This decoupling of models and controllers helps foster generator code reuse.

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[1]. 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.



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

[1]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.

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.

More information about JSR-223 can be found at https://scripting.dev.java.net/.