ORACLE
INSURANCE

Oracle® Insurance Policy Administration

Activity Processing

Version 9.2.0.0.0

Part Number: E16287_01
March 2010

Table of Contents

OVERVIEW

Activity processing is a core piece of Oracle Insurance Policy Administration (OIPA) and Oracle Insurance New Business Underwriting (OINBU) systems. Every administrative event that occurs in an insurance policy, plan, client or company is described as an activity in the system. An activity is an event that may or may not change relative data depending on a business's requirements.

An activity is an instance of a transaction. A transaction can be considered as synonymous to a class and an activity is synonymous to the object that is an instance of that class. Transactions are XML rules that are configurable according to business requirements. They are configured using the Oracle Insurance Rules Palette. Transactions are configured to define input variables, processing logic and lists of changes it makes to data such as policy information. Transactions can be defined in the system at the policy, plan, client and company level depending on the type of data it needs to execute. For example, typical OIPA transactions at the policy level are premium, billing and anniversary processing.

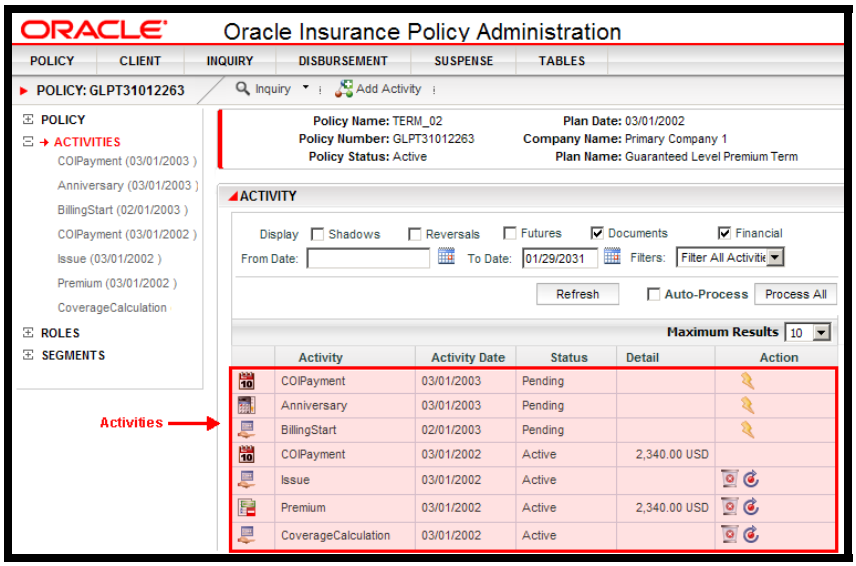

Figure 1 – Front-end OIPA User Interface Listing Activities that Occurred at the Policy Level

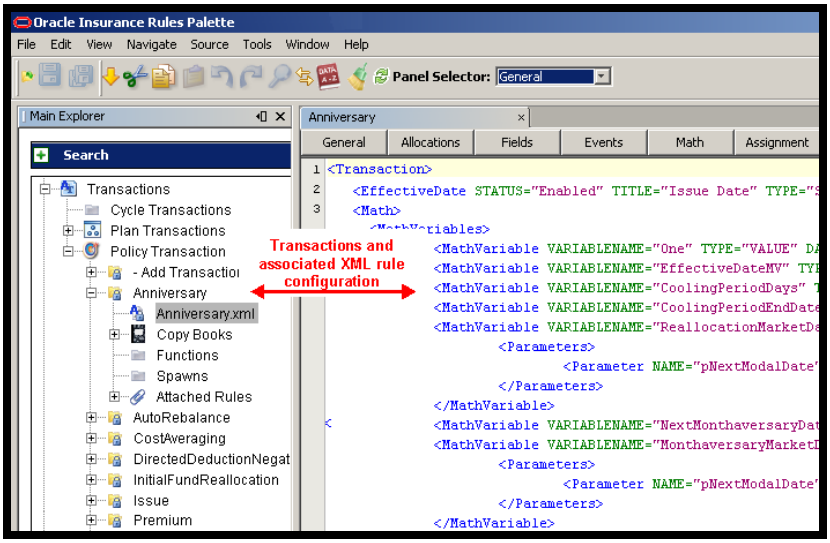

Figure 2 – Rules Palette Interface for Configuration of Transactions that Become Activities

SHARED RULES ENGINE

The shared rules engine is the component that performs activity processing for the OIPA and OINBU applications. In the system an activity is processed to manage an insurance event that has occurred. The shared rules engine loads the transaction that the activity is an instance of, processes the data according to the rules and math specified in the transaction, and sends the results back to the calling module. The database stores both the configured transaction rules and the actual insurance data.

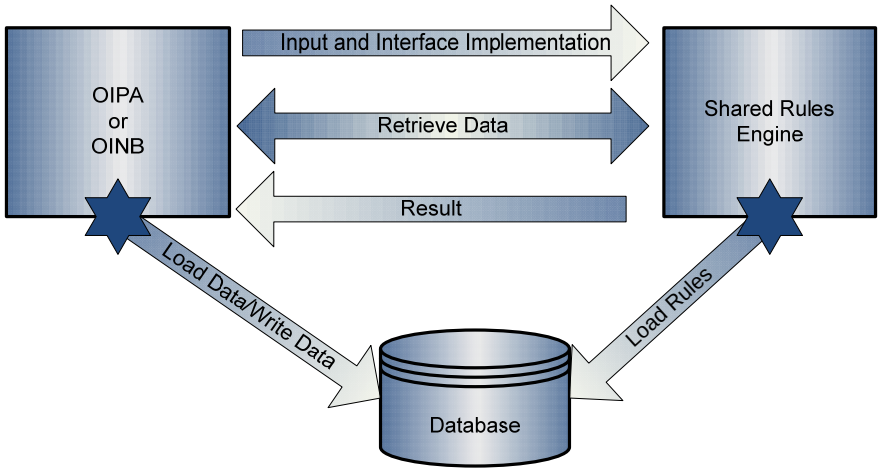

Figure 3 - High Level Interaction Diagram

The above diagram shows a high level interaction between the calling applications and shared rules engine. OIPA calls the shared rules engine with input data and interfaces implemented for callback to process an activity. The shared rules engine then loads the transaction and any associated rules from the database to process the data. Using the callback interface if any additional data or rules are needed they are retrieved from the OIPA system. The engine then packages the result and returns it to OIPA. OIPA iterates the results for changes and writes the data to the database.

There are six components of the shared rules engine that come together to process an activity. They are as follows:

1. Processor
2. Generators
3. Math
4. Application Process Execution. This is part of the calling application but the shared rules engine calls into the application process execution during activity processing.
5. Extensibility
6. Profiling

NOTE: Currently all applications such as OIPA or OINBU and the shared rules engine remain in the same process space. They are not designed and will not function if deployed across process space.

INTERFACE BETWEEN THE SHARED RULES ENGINE AND OIPA

This section details the input/output and interfaces for communicating with the shared rules engine. Currently the shared rules engine and OIPA are not clearly separated with interfaces. OIPA does call select classes from the shared rules engine directly and the shared rules engine libraries are required for OIPA to be compiled. The goal of a clear separation has been identified.

Currently the ActivityProcessorBll.process class from the shared rules engine is called to process an activity. The class inputs three parameters and returns one result dcl.

Input Parameters

1. ***VariableHashMap*** - This is a HashMap of key value pair. The key is a string and the value is an instance of VariableDcl. All activity, policy, plan, transaction and withholding data are flattened into a key value pair for lookup.

   Example - All rules inside transaction math and rule processing look up using the keys shown below for the corresponding values.

   - **Activity:FieldsXX** would be used for activity properties.
   - **Policy:FieldXX** would be used for policy properties.

2. ***IApplicationCallback -*** This is an interface that has getter for other interfaces. All of the interfaces are lists of callback methods that solves a purpose. The callback interfaces are as follows:

   - **DataRetriever** – When the shared rules engine needs to execute a SQL statement or retrieve data based on name queries related to application activity processing, it uses DataRetriever to retrieve the data. The data is returned as IDataDcl that contains the row and column details from the result set. Even though the shared rules engine has a direct connection to the database for loading rules for processing, it reaches out to the calling application for application specific data. This method fits into a goal of separating the database to contain application data and rules information separately.
   - **RateRetriever** – This interface is used to retrieve rates depending on the rate description and the criteria for the rates. The calling application, such as OIPA, can store the rates for insurance in any manner and implement this interface for processing needs specified in the rules.
   - **IActivityTaskExecutor** – This interface is used to process other activities as part of this parent activity. This is used especially when running backdated activities. To process backdated activities,aAll active activities that appear in the activity timeline after the backdated activity must be undone and then the backdated activity must be processed. This interface is used to run other activities and commit them as part of the outer processing activity.
   - **IPolicyValuationBll** – This interface is used to value a policy and return the cash value. It also is used to locate details about the funds and their cash value, as well as deposits and removal history.
   - **ICurrencyBll** – This interface is used to load currency and round currency information.

3. ***ActivityProcessType –*** An enumerated type (enum) that specifies if the activity is complete. It does not need to check for undo activities, redo activities or a quote that does not save data to the database.

Output Result

ActivityProcessResultDcl – A complex Dcl that contains the inputs passed, math calculation variables, errors if applicable, a list of updates, inserts and deletes to the data as part of the rule processing. This data is then iterated to be updated to the database.

UNDERSTANDING ACTIVITIES

The processing of an activity is controlled by various attributes associated with the activity. Activities are instances of XML transaction rules being applied on data at a specified level in the application. The AsActivity table stores records for activity processing that house applicable business event data. The AsTransaction table stores the XML transaction logic that processes activity data. To understand the basics of activities there are three areas of main focus, which are the transaction type, the activity type and the activity status. Each of these areas of focus is tracked using type codes. Type codes may be found in the AsCodes table or via Admin Explorer>Codes in the Rules Palette. The type codes used in activity processing are used by the system and should not be changed.

NOTE: You may locate more documentation on this topic on Oracle's Technology Network at http://download.oracle.com/docs/cd/E14444_01/oipa_v8_1_activity_processing.pdf. The documentation at this location may provide your project team with the information needed to audit activity processing. Version 8 and Version 9 activity processing remain the same for auditing purposes.

TRANSACTION TYPES

The transaction type code that is stored in AsTransaction in the XML transaction rule associated with the activity plays an important role in activity processing. The transaction type code specifies what type of data or at what level the activity will process. This then drives which processing, such as math or valuation, should be executed by the system.

OIPA Transaction Types

Transaction Type	Description
Policy Financial	This transaction executes at the policy level and may or may not have financial impact.
Policy Document	This transaction executes at the policy level and generates documents or reports.
Plan Financial	This transaction executes at the plan level and may or may not have financial impact.
Client Financial	This transaction executes at the client level and may or may not have financial impact.
Policy Financial Non Reversible	This transaction executes at the policy level and cannot be reversed via the user interface.

ACTIVITY TYPES

Each activity record has an activity type code that is stored in the AsActivity table. The type code definition can be located in the AsCode table. These types should not be confused with the transaction type code or the status of an activity, but instead, used in conjunction with them to understand how an activity was generated and what status the activity is currently in. Activities can be generated by an end user or the system may automatically generate activities because a dependent activity's data was changed.

OIPA Activity Types

Activity Type	Type Code	Description
Natural	01	Activity entered manually by a user. Activity that was spawned for the first time from a natural activity. A spawned activity even though system generated can be considered a natural activity because the user manually processed the activity that spawned it. Activity created by a web service.
Reversal	02	Reversal activity that was created by an end user either manually deleting or recycling an activity. Spawned activity that was reversed because the originating activity was manually reversed.
Undo	03	Activity that is created to reverse an active activity that is created by the system as part of running another reversal or as part of processing a pre-dated activity. This behaves exactly as the Reversal but just differentiates itself as system generated.
Redo	04	System generated activity that was automatically created due to the generation of an Undo activity.

ACTIVITY STATUS

The activity statuses are fundamental for activity processing and historical recording. They indicate at the activity level the status of that activity record. In comparison, the activity types section records the type of activity that was processed. The activity status, with the date stamp in current and history records, identifies the significant point of process and provides internal control for activities.

Activity Statuses

Activity Status Type	Type Code	Description
Pending	02	The activity is not yet processed. Pending data requires action before applying to current processing and math calculation. All required data must be entered and the lighting bolt selected to change the status from pending to active.
Active	01	Indicates the activity is active. Refers to current data that has completed activity processing and math calculation. This includes processing, any changes to table and inserting XML to write to the table. No more processing can be done on this activity.
Pending Ready	09	An attempt was made to run the activity but was unsuccessful.
NUV Pending	13	The activity is active but it does not have NUVs for some or all of the funds associated with activity processing. This will process later when NUVs become available. This status does not invoke undo/ redo processing for future active activities.
Gain Loss Pending	14	This activity is active but gain loss calculation is pending and is not complete. This will be processed later when NUVs are available. This status does not invoke undo/ redo processing for future active activities.
Shadow	12	This activity is effectively deleted from the system from an end user perspective as it is a result of an activity being reversed. It is available in the database and system for auditing purposes.
Pending Shadow	34	An activity whose data was entered, but never processed and then deleted.

Activity Processing Flow

The activity processing flowchart walks you through system steps.

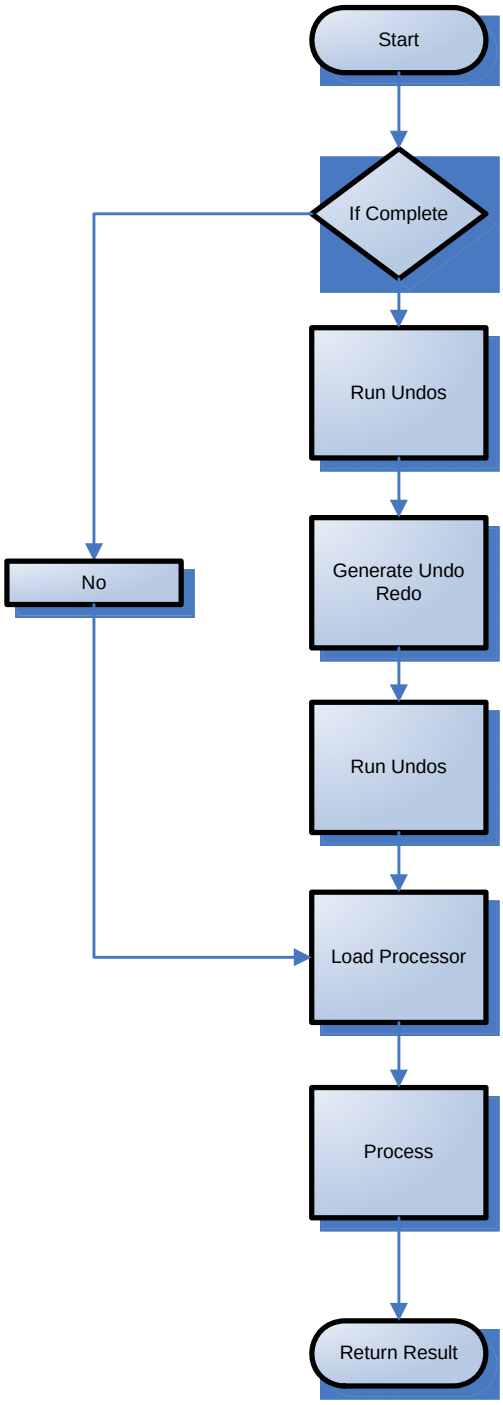

Figure 4 – Activity Processing Flow

Start – The shared rules engine receives a request from OIPA. The processing proceeds only if the activity is not active.

If Complete – This is the third parameter ***ActivityProcessType*** that is sent by OIPA. It has three values: COMPLETE, SKIP_UNDOREDO_GENERATION or QUOTE. Strip down processing is done for options other than COMPLETE. QUOTE is for quoting an activity and SKIP_UNDOREDO_GENERATION is called in undo processing and for a specific instance called during cycle processing after processing one activity in a policy. COMPLETE is the default option.

Run Undos – This step looks for all pending undo activites that need to be run with an effective date after the current activity effective date and executes them. This logic calls back into OIPA and it calls the shared rules engine in recursion to execute the undo activity. If there are no activities in future relative to the current then this step is skipped.

Generate Undo/Redo – This step looks for all activities that are active with an effective date after the current activity effective date. It creates an Undo/Redo for those activities.

Run Undos – If in the previous steps there are any activities generated then this step runs the undos of the activities generated.

Load Processor – It loads the corresponding processor depending on the activity type code, activity status code and transaction type code.

Process - Call the processor process method. This is explained in detail in the next section of this document.

SUBCOMPONENTS OF THE SHARED RULES ENGINE

Depending on the activity type that is sent for processing, an appropriate processor is initiated that handles the processing steps. The different processing types and diagram are as follows:

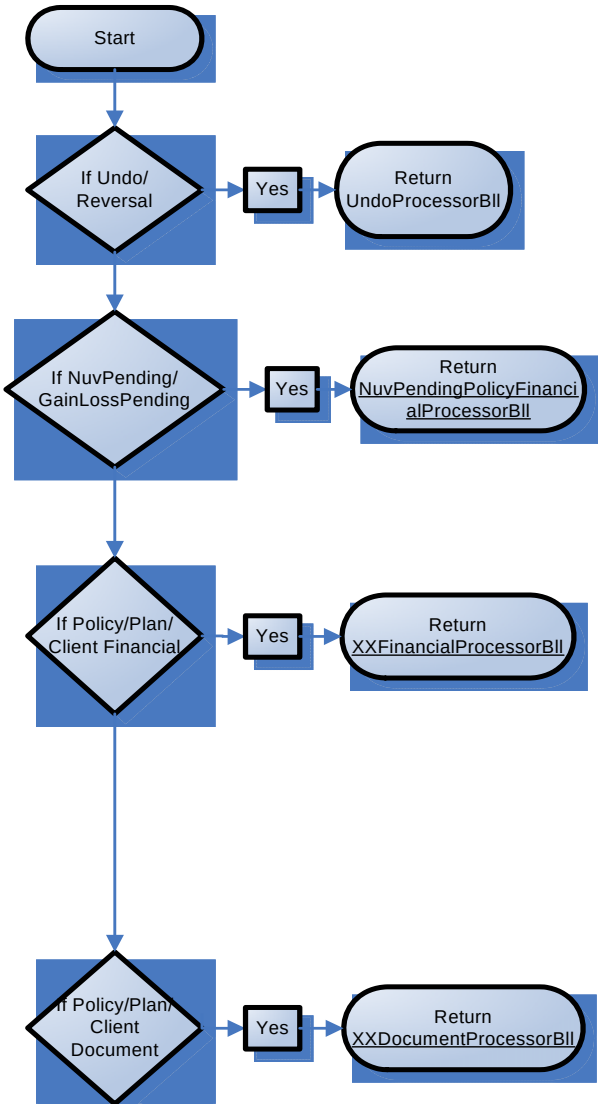

Figure 5 – Processing Types

Undo/Reversal Activity – Activity is already processed and it needs to be undone. Handled by UndoProcessorBll.java

Nuv Pending/Gain Loss Pending Activity – Activity is processed and is in active status, but some NUV's are missing or Gain Loss calculation is missing due to missing data. Handled by NuvPendingPolicyFinancialProcessorBll.java

Policy Level Activity – Activity at the policy holder that impacts the policy alone. Handled by PolicyFinancialProcessorBll.java

Client Level Activity – Activity at the client level that impacts client data and might impact all policies the client holds. Handled By ClientFinancialProcessorBll.java

Plan Level Activity – Activity at the plan level that aggregates all policies in the plan like reports or other changes to the plan. Handled by PlanFinancialProcessorBll.java

Document Generation – Activity that generates just reports are handled by DocumentProcessorBll classes. Separate classes exist for Policy Level Documents(PolicyDocumentProcessorBll) , Plan level documents (PlanDocumentProcessorBll) and Client level Documents (ClientDocumentProcessorBll).

LIST OF PROCESSES AS PART OF ACTIVITY PROCESSING

Depending on the processors, different sections of activity processing are executed. The processes are as follows:

- **doPreliminary** – Loads NUVs for funds and prepares the activity for processing.
- **Suspense Processing** – Processes suspense for funds received.
- **doValuation** – Values the policy of all funds and calculates the cash value and other variables. This is called only when the transaction calls for the valuation in its rules. The shared rules engine calls OIPA to do the valuation using the interface. The calculated values are later used in other sections of activity processing.
- **doMath** – Calculates the math section of rules.
- **doBusinessLogic** – Runs the application process execution associated with the activity.
- **doAssignment** – Runs assignment processing.
- **doDisbursement** – Runs disbursement processing.
- **doAccounting** – Runs accounting for bookkeeping purpose.
- **doSpawn** – Runs spawn logic to spawn new activities based on the transaction specified rules.

There are various sub processes that run during activity processing. Processors like Undo and NuvPending run few of these and also run other processes like loading the changes that happened during activity processing and reversing those changes.

GENERATORS

Generators are classes that produce other classes for execution. All XML rules are configurable with expressions and conditions and generators are used to execute these rules. Generators are responsible for loading the rules, performing error checking, translating the rules and creating java source code at run time for the rules, and compiling them into classes using janino. Then they create an instance of the run time generated class and return them to the caller for execution.

Generators have the following functions:

- Load the rules.
- Parse the rules and check for errors. Report Errors if needed.
- Translate the rules to java code and compile the class.
- Cache the translation for next time lookup.

Generators run in the modes described below. The mode can be set in the application property file, such as PAS.properties. Information regarding the PAS.properties file can be located on Oracle's Technology Network at http://download.oracle.com/docs/cd/E16287_01/oipa_v92000_pas_properties.pdf.

The PAS.properties file section where you set the application mode:

#--
 # application mode (DEVELOPMENT or PRODUCTION)
 # Development mode is where configuration changes are allowed.
 # Production mode is where configuration change is a new release and JVM is restarted when they are changed.
#--

application.mode= DEVELOPMENT

In DEVELOPMENT mode the system allows rules to be changed in the database during application runtime. This mode should be used during active development. Generators load the rules every time, generate a hash key and cache the generated classes associated with the hash key. If the rules are changed using Oracle Insurance Rules Palette, then the hash key generated will be different, which will force the generator to translate and compile again. If the rules are not changed then it reaches out to its cache and returns the cached instance.

application.mode= PRODUCTION

In PRODUCTION mode the system does not allow the changing of rules. If rules are modified, it requires all the JVM's to be stopped and restarted so that caches are cleared. In production mode, rules are cached and so are the translated classes. Hence no check is made to ensure changes.

Generators support debugging mode and non debugging mode. The Rules Palette can debug into transactions and do a line by line execution of the math section using a web service. In order to debug via the Rules Palette, the application should be started in debugging mode. Debugging mode adds a lot of extra information to enable remote debugging and hence generators create extra lines of code.

The PAS.properties file section for settings debugging:

#--
 # This property allows remote level debugging or not. Yes or No.
#--

debug.remoteDebugging=No

If set to No, the application will not support remote debugging at runtime. If set to Yes, then remote debugging is supported.

To support developer debugging of activity processing, Generators can save the generated classes to a local file system if configured in the property file. If debug.IdentiyTranslator is set to Yes, in the java files generated, at the end of each file it will add a comment identifying the translator class and the line number that generated that line of code. This is extremely useful in debugging the generated source code and changing it for future needs.

There are different types of generators for different purposes. They are described in the next section of this document.

The PAS.properties file section for settings for debugging properties:

#--
 # Directory to save generated source code.
 # This property will be used to debug issues with sre processing.
 # Generated source code while processing will be saved in the
 # directory specified. Only to be used in Non Production environment.
 # debug.identifyTranslator will write comments for every line identifying
 # the translator(line number) that generated that part of code.
#--
debug.SaveGeneratedClass=Yes
debug.identifyTranslator=Yes
debug.SaveGeneratedClassDirectory=c:\\temp

PASTRANSACTIONGENERATOR

PasTransactionGenerator is a Generator specific to OIPA transactions. It understands the rules of the OIPA transaction and generates the classes suited to its needs for processing. All generated classes by this Generator extend from PasTransactionBll, which implements the basics of OIPA activity processing.

Logic Flow of the Transaction Generator

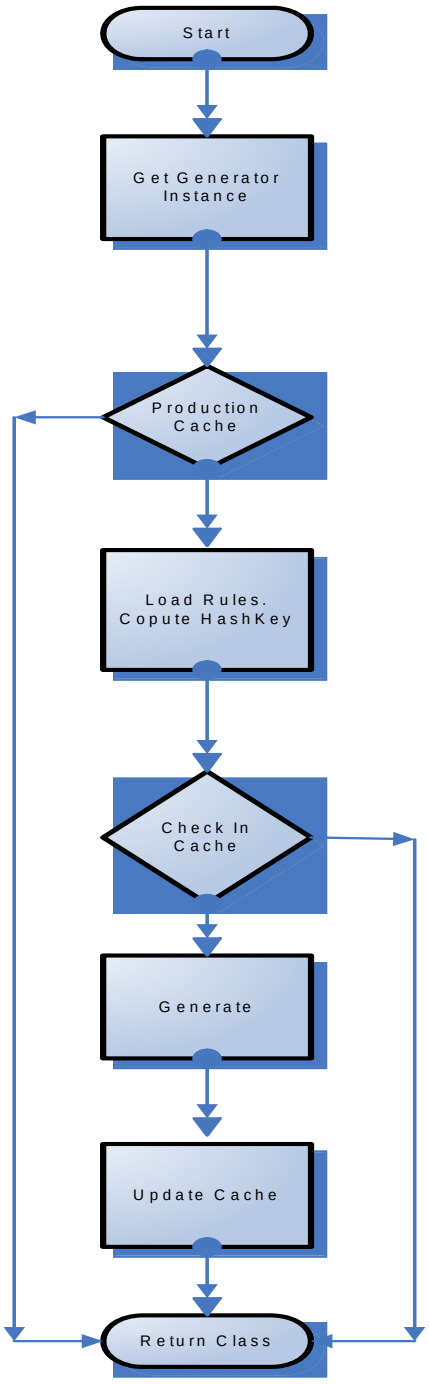

Figure 6 – Flow of Transaction Generator

Start – Shared rules engine calls the static method in the Generator for activity processing. *PasTransactionGenerator. getTransactionBllForProcessing*

Get Generator Instance - Creates an instance of a generator class per transactionguid. There is only one instance of the Generator per transaction, but many instances of the Generator for different transactions. This prevents multiple threads calling to process activities of same transaction type and simultaneously translating the same rules. Only one thread translates the rules for a transaction and other threads, it there are any, wait for the first thread to complete. It then uses the class for processing.

Production Cache – Generator then looks at the cache to see if a class exists for this transaction GUID. In Development mode, the cache will not contain the key. In Production mode, if the transaction is already translated, it will pick it up and return.

Load Rules, Compute HashKey – If false in the above decision, the Generator loads the rules from the database. It computes the unique hash key for the rules XML.

Check Cache – It then checks in the cache to see if it has a class file for the hash key generated. In Development mode, if the transaction is updated then the hash key will be different and it will force the Generator to translate again.

Generate – It will parse the rules, translate the rules using translators and then compile the generated java classes. It also saves to the file system if specified in the property file.

Update Cache – It updates the cache depending on development or production mode for future use. Future calls with the same transaction GUID and same hash key are not translated.

Return Class – Returns the instance of the generated class to the caller.

SegmentCalculatorGenerator

SegmentCalculatorGenerator is used to create classes at runtime for segment calculation based on the rules. This creates classes specific to the OIPA system. SegmentCalculator follows the same algorithm as the PasTransactionGenerator except that it has only one instance of the generator class for all segment rules where PasTransactionGenerator has one instance per transaction GUID.

Hence at any given point of time in a JVM, only one segment calculation can be translated.

NOTE: If there are multiple request to retrieve a segment calculator class, one gets through and others block until the class is returned from the cache or translated and compiled. The code should change similar to TransactionGenerator. Currently there are no issues identified because segment calculation is much less compared to activity processing.

MathEngineFactory

This is the stand alone math Generator. This class is not named like other classes, which end in the word Generator. It is good to note this, so as not to cause confusion regarding it being a Generator. The MathEngineFactory loads the rules with only math sections and creates a class that executes the math rules and returns the results. This is an independent math generator that is used by valuation, exposed computation and any module that has math sections that need to execute.

FunctionDefinationGenerator

FunctionDefinitionGenerator is used to create function code. The generated function rule classes are embedded within the transaction, segment or math classes. Generated function classes do not exist as a separate entity but are available as inner classes. These Generators are not thread synchronized because currently they are called from one of the above generators and they are throttled above.

ScreenEventGenerator

There are three types of ScreenEventGenerators: OnLoadGenerator, OnSubmitGenerator and OnChangeGenerators. These are used to process the rules at three different events of the application. They are not related to activity processing but part of the shared rules engine as they involve processing math calculation.

MATH

Math module is a sub component of the shared rules engine that is responsible for executing any math sections in the rules. In a rule all tags between the `<MathVariables>` element are handled by this sub component.

Conceptual Math Functionality

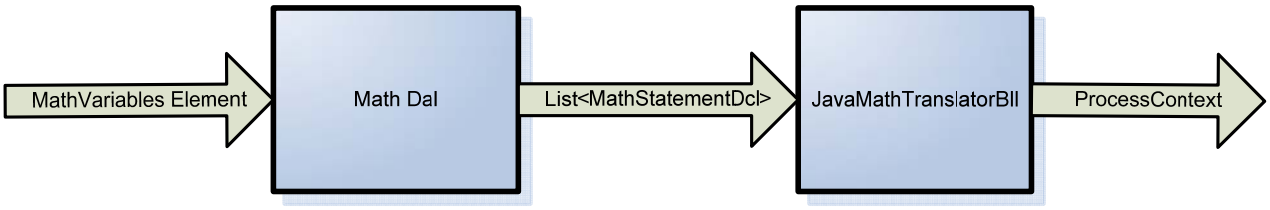

Figure 7 – Conceptual Math Functionality

1. Generator that generates java source for <MathVariables> section calls the MathDal with the location to MathVariables Element in the rules xml file.
2. The MathDal classes parses the element and its sub elements and creates a list `<MathStatementDcl>` and returns it as an output. MathStatementDcl's represents the entire tree hierarchy of the math section with loops and MathIF's.
3. The above List `<MathStatementDcl>` is sent to JavaMathTranslatorBll for translation. Each math statement dcl is translated and the corresponding java code for that statement is set in the MathStatementDcl itself.
4. JavaMathTranslatorBll returns an instance of ProcessContext that contain lots of Information. It contains
5. List of MathVariables declared.
6. List of functions called.
7. Other structures for dependency and debugging purposes.
8. Generators get the ProcessContext and generates the final class with variable declaration, statements and function calls for compilation and execution.

TRANSLATORS

Translator classes are the most important piece of the Math sub component. Translators are responsible for translating every MathStatementDcl to its corresponding java source code. Translators perform the error checking and also code generation for the single XML line.

Each <MathVariable> type that is defined by the TYPE attribute has one or more translators associated with it depending on the operations allowed on the math type and its complexity. MathVariableType.java, an enum, defines the list of all MathVariable TYPE and the corresponding translator classes. JavaMathTranslatorBll iterates through the MathStatementDcl and invokes the corresponding translator with the MathStatementDcl to perform the translation.

NOTE: Please see the XML Configuration Guide on Oracle's Technology Network for more details regarding XML schemas and definitions used by various OIPA rules.