exigen®

OpenL Tablets Developer Guide

OpenL Tablets 5.13

OpenL Tablets BRMS

Document number: TP_OpenL_Dev_2.4_LSh
Revised: 09-22-2014

Table of Contents

Preface

This preface is an introduction to the *OpenL Tablets Developer Guide*.

The following topics are included in this preface:

Audience

This guide is mainly intended for developers who create applications employing the table based decision making mechanisms offered by OpenL Tablets technology. However, business analysts and other users can also benefit from this guide by learning the basic OpenL Tablets concepts described herein.

Basic knowledge of Java, Ant, and Excel® is required to use this guide effectively.

Related Information

The following table lists sources of information related to contents of this guide:

Related information	
Title	**Description**
OpenL Tablets WebStudio User Guide	Document describing OpenL Tablets WebStudio, a web application for managing OpenL Tablets projects through web browser.
http://openl-tablets.sourceforge.net/	OpenL Tablets open source project website.

Typographic Conventions

The following styles and conventions are used in this guide:

Typographic styles and conventions	
Convention	**Description**
Bold	• Represents user interface items such as check boxes, command buttons, dialog boxes, drop-down list values, field names, menu commands, menus, option buttons, perspectives, tabs, tooltip labels, tree elements, views, and windows. • Represents keys, such as **F9** or **CTRL+A.** • Represents a term the first time it is defined.
`Courier`	Represents file and directory names, code, system messages, and command-line commands.
`Courier Bold`	Represents emphasized text in code.

Typographic styles and conventions	
Convention	**Description**
Select **File > Save As**	Represents a command to perform, such as opening the **File** menu and selecting **Save As.**
Italic	• Represents any information to be entered in a field. • Represents documentation titles.
< >	Represents placeholder values to be substituted with user specific values.
Hyperlink	Represents a hyperlink. Clicking a hyperlink displays the information topic or external source.

Chapter 1: Introducing OpenL Tablets

This section introduces OpenL Tablets and describes its main concepts.

The following topics are included in this section:

What Is OpenL Tablets?

OpenL Tablets is a business rules management system and business rules engine based on tables presented in Excel documents. Using unique concepts, OpenL Tablets facilitates treating business documents containing business logic specifications as executable source code. Since the format of tables used by OpenL Tablets is familiar to business users, OpenL Tablets bridges a gap between business users and developers, thus reducing costly enterprise software development errors and dramatically shortening the software development cycle.

In a very simplified overview, OpenL Tablets can be considered as a table processor that extracts tables from Excel documents and makes them accessible from application.

The major advantages of using OpenL Tablets are as follows:

1. OpenL Tablets removes the gap between software implementation and business documents, rules, and policies.
2. Business rules become transparent to developers.
3. For example, decision tables are transformed into Java methods or directly into web service methods. The transformation is performed automatically.
4. OpenL Tablets verifies syntax and type errors in all project document data, providing convenient and detailed error reporting. OpenL Tablets is able to directly point to a problem in an Excel document.
5. OpenL Tablets provides calculation explanation capabilities, enabling expansion of any calculation result by pointing to source arguments in the original documents.
6. OpenL Tablets enables users to create and maintain tests to insure reliable work of all rules.
7. OpenL Tablets provides cross-indexing and search capabilities within all project documents.
8. OpenL Tablets provides full rules lifecycle support through its business rules management applications

OpenL Tablets supports the `.xls` and `.xlsx` file formats.

Basic Concepts

This section describes the following main OpenL Tablets concepts:

Rules

In OpenL Tablets, a **rule** is a logical statement consisting of conditions and actions. If a rule is called and all its conditions are true then the corresponding actions are executed. Basically, a rule is an IF-THEN statement. The following is an example of a rule expressed in human language:

If a service request costs less than 1,000 dollars and takes less than 8 hours to execute then the service request must be approved automatically.

Instead of executing actions, rules can also return data values to the calling program.

Tables

Basic information OpenL Tablets deals with, such as rules and data, is presented in tables. Different types of tables serve different purposes. For detailed information on table types, see OpenL Tablets Reference Guide, the *Table Types* section.

Projects

An **OpenL Tablets project** is a container of all resources required for processing rule related information. Usually, a project contains Excel files, Java code. For detailed information on projects, see OpenL Tablets Reference Guide, chapter *Working with Projects*.

There can be situations where OpenL Tablets projects are used in the development environment but not in production, depending on the technical aspects of a solution.

Wrapper

A **wrapper** is a Java object that exposes rule tables via Java methods, data tables as Java objects and allows developers to access table information from code. Wrappers are essential for solutions where compiled OpenL Tablets project code is embedded in solution applications. If tables are accessed through web services, client applications are not aware of wrappers but they are still used on the server.

For more information about wrappers, see section Using OpenL Tablets rules from Java Code.

Execution mode for OpenL project

Execution mode for OpenL project is a light weight compilation mode that enables only evaluating of rules; but editing, tracing and search are not available. Since the Engine will not load test tables and keep debug information in memory in this mode, memory consumption is up to 5 times less than for debug mode.

By default the execution mode (`exectionMode=true`) is used in OpenL Tablets Web Services.

The debug mode (`exectionMode=false`) is used by default in WebStudio.

Flag indicating required mode is introduced in Runtime API and in wrappers.

To compile OpenL project in execution mode:

In case you use OpenL high level API (instantiation strategies) you can define an execution mode in constructor of the particular instantiation strategy
In case you use low level API (Engine factories) you can set execution mode flag using `setExecutionMode(boolean)` method

System Overview

The following diagram shows how OpenL Tablets is used by different types of users:

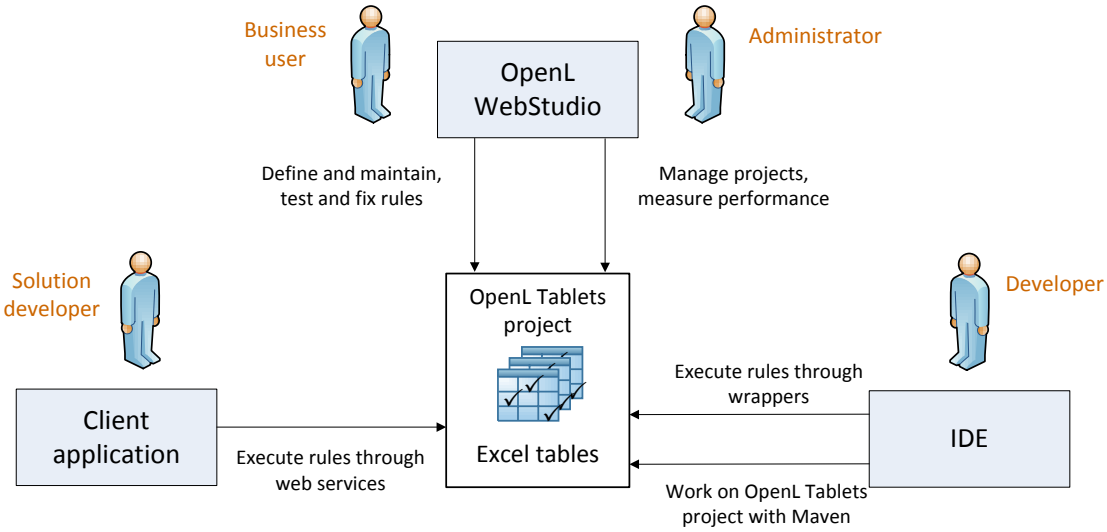

Figure 1: OpenL Tablets overview

The following is a typical lifecycle of an OpenL Tablets project:

1. A business analyst creates a new OpenL Tablets project in WebStudio. Optionally, development team may provide the analyst with a project in case of complex configuration.

   The business analyst also creates correctly structured tables in Excel files based on requirements and includes them in the project. Typically, this task is performed through Excel or OpenL Tablets WebStudio in a web browser.
2. The business analyst performs unit and integration tests by creating test tables and performance tests on rules through OpenL Tablets WebStudio. As a result fully working rules are created and ready to be used.
3. A developer adds configuration to the project according to application needs. As alternative approach he can create a new OpenL Tablets project in his IDE via OpenL Maven Archetype and adjust it to use business user input.

4. A developer employs business rules directly through the OpenL Tablets engine or remotely through web services.
5. Whenever required, the business user updates or adds new rules to project tables. OpenL Tablets business rules management applications (OpenL Tablets WebStudio, Rules Repository, Rule Service) can be setuped to provide self-service environment for business user changes.

Quick Start with OpenL Tablets

OpenL Tablets provide a few ways to create a new project. We recommend to use Simple Project Maven Archetype approach for creating a new project first time or create it via WebStudio. You can find all approaches to create a project with detailed descriptions in How to Start With OpenL Rules Project section.

After a project is created, you can use a zip or excel file for importing the project to OpenL Tablets WebStudio. You can find detailed information about importing existed project into WebStudio in *OpenL Tablets WebStudio User Guide*.

OpenL Tablets WebStudio provides convenient UI to work with rules. However, you can avoid its usage and work with rules from your IDE only using OpenL Tablets Maven Plugin. The plugin provides compilation and testing of rules and wrapper generation support.

Also, OpenL Tablets has OpenL Tablet Demo Package. You can download it in OpenL Tablets website. A demo is a zip file that contains a Tomcat with configured OpenL Tablets WebStudio and OpenL Tablets Web Services projects. You can use it for effectively start of using OpenL Tablets products.

Chapter 2: OpenL Tablets Rules Projects

This section describes about OpenL Tablets Rules projects: how to create and how to use them.

The following topics are included in this section:

OpenL Rules Project

OpenL Rules project is a project that contains Excel files with OpenL Tablets rules and may have a rules project descriptor. The rules project descriptor is a XML file that defines project configuration and allows to set project dependencies. Via dependency functionality OpenL Rules Project can easily use rules from other projects.

Rules Project Descriptor

A rules project descriptor is an XML file that contains information about the project and configuration details used by OpenL to load and compile the rules project. The predefined name is used for a rules project descriptor - *rules.xml*

Quick Overview

The snippet bellow is the example of rules project descriptor:

```
<project>
	<!-- Project name. -->
	<name>Project name</name>
	<!-- Optional. Comment string to project. -->
	<comment>comment</comment>

	<!-- OpenL project includes one or more rules modules.  -->
	<modules>

		<module>
			<name>MyModule1</name>
			<type>API</type>

<!--
				Rules document which is usually excel file in the project.
			-->
			<rules-root path="MyModule1.xls"/>
```

```
		</module>

		<module>
			<name>MyModule2</name>
			<type>API</type>

<!--
				Rules document which is usually excel file in the project.
			-->
			<rules-root path="MyModule2.xls"/>
			<method-filter>
				<includes>
					<value> * </value>
				</includes>
			</method-filter>
		</module>
	</modules>

<dependencies>
		<dependency>
			<name>projectName</name>
			<autoIncluded>false</autoIncluded>
		</dependency>
	</dependencies>
	<properties-file-name-pattern>{lob}</properties-file-name-pattern>
	<properties-file-name-
processor>default.DefaultPropertiesFileNameProcessor</properties-file-name-processor>
	<!-- Project's classpath (list of all source dependencies). -->
	<classpath>
		<entry path="path1"/>
		<entry path="path2"/>
	</classpath>

</project>
```

Descriptor Elements

Descriptor file contains several sections that describe projects configuration.

Project configurations

Project section		
Tag	**Required**	**Description**
name	yes	Project name. String value which defines user-friendly project name.
comment	no	Comment for project.
dependency	no	Define dependencies to projects.
modules	yes	Defines modules of project. Project can have one or more modules.
classpath	no	Project relative classpath.
properties-file-name-pattern	no	A file name pattern that will be used by file name processor. File name processor adds extracted module properties from a module file name.
properties-file-name-processor	no	A custom implementation of org.openl.rules.project.PropertiesFileNameProcessor that is used instead of default implementation.

Module configurations

Module section		
Tag	**Required**	**Description**
name	yes/no	Module name. String value which defines user-friendly module name. *Notice*: Used by WebStudio application as module display name. Not required for modules defined via wildcard.
type	yes	Module instantiation type. Possible values (case-insensitive): **dynamic**, **api, static** (deprecated). Defines the way of OpenL project instantiation.
classname	yes/no	Used together with *type*. Defines name of rules interface. Not required for *api* type.
method-filer	no	A filter that defines what tables should be used for interface generation. Java regular expression can be used to define a filter for multiple methods.
rules-root	yes/no	Used together with *type*. Defines path to the main file of rules module. Ant pattern can be used to define multiple modules via wildcard. See Ant patterns for more information.

Dependency configurations

Dependency section		
Tag	**Required**	**Description**
name	yes	A dependency project name.
autoIncluded	yes	If true, all modules from dependency project will be used in this project modules. If false, modules from dependency project can be used in this project as dependencies and each module will define its own list of used dependency modules.

Classpath configurations

Classpath section		
Tag	**Required**	**Description**
entry	no	Defines path for the classpath entry (classes or jar file).

Project Resolving

There is a *RulesProjectResolver* that resolves all OpenL projects inside the workspace. The Resolver just lists all folders in workspace and tries to detect OpenL project by some predefined strategy. The most easy way to init *RulesProjectResolver* is to use the static method *loadProjectResolverFromClassPath()* that will use **project-resolver-beans.xml** from classpath (this is usual Spring beans config that defines all resolving strategies and their order).

Make sure that resolving strategies are in the right order: some projects may be matched by several resolving strategies. By default, resolving strategies are in the following order:

Project Descriptor resolving strategy

This resolving strategy is the strictest resolving strategy. This strategy is based on the descriptor file (see above).

Excel file resolving strategy

This is the resolving strategy for the simplest OpenL project which contains only Excel files in root folder without wrappers and descriptor. Each Excel file represents a module.

How to start with OpenL Rules Project

At first, you need to create OpenL Rules project. It can be done in following ways: using maven archetype, using OpenL Tablets WebStudio and manually.

Create project using Maven archetype

OpenL Tablets provides archetype which can be used to create simple OpenL Rules project.

Execute in command line the following command:

```
mvn archetype:generate
```

Maven runs archetype console wizard. Select *openl-simple-project-archetype* menu item. Follow with maven creation wizard.

After all steps are finished you should get a new maven based project on file system. It's an OpenL Rules project which has one module with simple rules in it.

Execute in command line the following command from the root of the project folder to compile the project:

```
mvn install
```

After this command, you'll find in the target folder:

1. zip file (with "-deployable" suffix) for importing a project to WebStudio. You can find more information in [*OpenL Tablets WebStudio User Guide*].
2. zip file (with "-runnable" suffix) that can be executed after unpacking. It demonstrates how OpenL rules can be invoked from java code.
3. jar file that contains only compiled java classes. This jar can be putted in classpath of your project and used as depended library.

Create project in OpenL Tablets WebStudio

OpenL Tablets WebStudio allows users to create new rule projects in the Repository in one of the following ways:

- Create a rule project from template

- Create a rule project from Excel file(s)
- Create a rule project from zip archive
- Import a rule project from workspace

The following diagram explains how projects are stored in WebStudio and then deployed and used by WebServices:

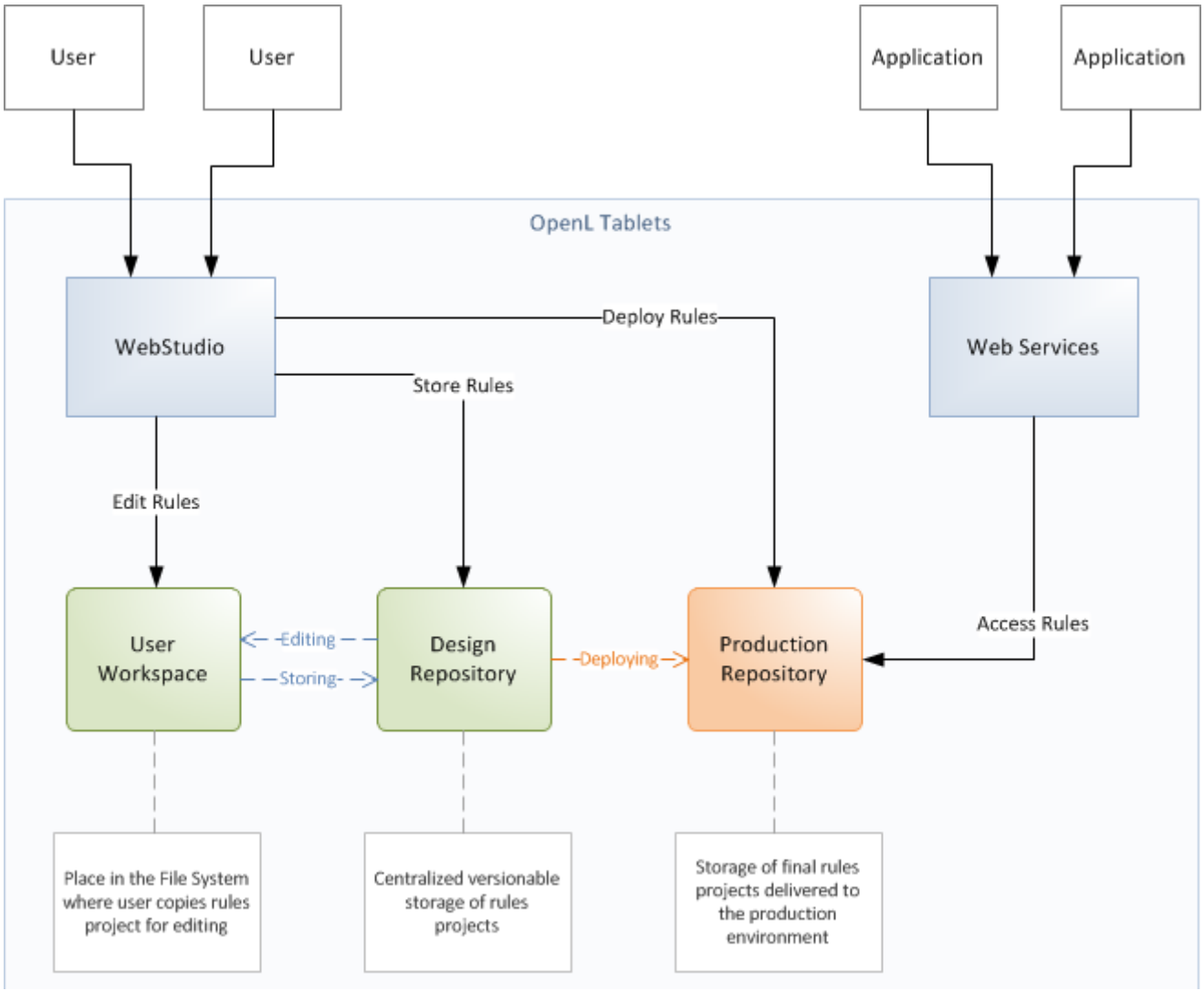

Figure 2: OpenL Tablets WebStudio and Web Services Integration

When a user starts editing a project, it is extracted from Design Repository and placed on the File System (User Workspace). The project becomes locked in Design Repository for editing by other users. After editing is finished, the user saves the project. It means that an updated version of the project is saved to Design Repository and available for editing by other users again.

Web Services use separate repository instance (Production Repository in the picture above). WebStudio can be configured to deploy complete and tested rules projects to that repository.

See [*OpenL Tablets WebStudio User Guide*] for more information.

Create project manually

OpenL doesn't oblige user to use predefined ways of project creation and provides way to use your own project structure. You can use OpenL Project Resolving mechanism as a base for your project structure definition. Depends on resolving strategy you need create more or less files and folders but several project's elements must be defined. See Project's elements section of the current document for more details.

Edit rules

After you created a project you need to create business rules. You can do it using OpenL Tablets WebStudio application or manually using MS Excel. If you use the simple rules project there are several simple predefined rules you have as an example.

Using OpenL Tablets rules from Java Code

Access to rules and data in Excel tables is realized through OpenL Tablets API. OpenL Tablets provides wrapper to developers to facilitate easier usage.

This section illustrates the creation of a wrapper for a **Simple** project in IDE. There is only one rule **hello1** in the **Simple** project by default.

Rules void hello1(int hour)			
Rule	**C1**	**C2**	**A1**
	min <= hour	hour <= max	System.out.println(greeting + ", World!")
	int min	int max	String greeting
Rule	**From**	**To**	**Greeting**
R10	0	11	Good Morning
R20	12	17	Good Afternoon
R30	18	21	Good Evening
R40	22	23	Good Night

Figure 3: The hello1 rule table

Proceed as follows:

1. In the project `src` folder, create an interface as follows:

```
public interface Simple {
    void hello1(int i);
}
```

2. Create a wrapper object as follows:

```
import static java.lang.System.out;
import org.openl.rules.runtime.RuleEngineFactory;

public class Example {
```

```
        public static void main(String[] args) {
                //define the interface
                RuleEngineFactory<Simple > rulesFactory =
                        new RuleEngineFactory<Simple>(" TemplateRules.xls",
                                                        Simple.class);

                Simple rules = rulesFactory.newInstance();
                rules.hello1(12);

        }
    }
```

When the class is run, it executes and displays **Good Afternoon, World!**

Interface can be generated by OpenL Tablets in runtime, if developer doesn't define it when initializing rule engine factory. In this case rules can be executed via reflection.

The following example illustrates using a wrapper with generated interface in runtime:

```
public static void callRulesWithGeneratedInterface(){
        // Creates new instance of OpenL Rules Factory
        RuleEngineFactory<?> rulesFactory =
ew RuleEngineFactory<Object>("TemplateRules.xls");
                                //Creates new instance of dynamic Java Wrapper for our lesson
        Object rules = rulesFactory.newInstance();

        //Get current hour
        Calendar calendar = Calendar.getInstance();
        int hour = calendar.get(Calendar.HOUR_OF_DAY);

        Class<?> clazz = rulesFactory.getInterfaceClass();

        try{
                Method method = clazz.getMethod("hello1", int.class);
                out.println("* Executing OpenL rules...\n");
                method.invoke(rules, hour);
        }catch(NoSuchMethodException e){
        }catch (InvocationTargetException e) {
        }catch (IllegalAccessException e) {
        }
}
```

Using OpenL Tablets Rules with the Runtime Context

This section describes the use of the runtime context for dispatching versioned rules by dimension properties values.

For example, consider two rules overloaded by dimension properties. Both rules have the same name.

The first rule, covering an Auto line of business, is following (pay attention to line with lob property):

Rules void hello1(int hour)			
properties	createdOn	4/7/10	
	createdBy	LOCAL	
	lob	Auto	
Rule	C1	C2	A1
	min <= hour	hour <= max	System.out.println(greeting + ", World!")
	int min	int max	String greeting
Rule	From	To	Greeting
R10	0	11	Good Morning
R20	12	17	Good Afternoon
R30	18	21	Good Evening
R40	22	23	Good Night

Figure 4: The auto rule

The second rule, covering a Home line of business, is following:

Rules void hello1(int hour)			
properties	modifyOn	4/7/10	
	modifiedBy	LOCAL	
	lob	Home	
Rule	C1	C2	A1
	min <= hour	hour <= max	System.out.println(greeting + ",Guys!")
	int min	int max	String greeting
Rule	From	To	Greeting
R10	0	11	It is Mornig
R20	12	17	It is Afternoon
R30	18	21	It is Evening
R40	22	23	It is Night

Figure 5: The homerule

A wrapper enables the user to define which of these rules must be executed.

```
// Getting runtime environment which contains context
IRuntimeEnv env = ((IEngineWrapper) rules).getRuntimeEnv();

// Creating context
IRulesRuntimeContext context = new DefaultRulesRuntimeContext();
env.setContext(context);
// define context
context.setLob("Home”);
```

As a result, the code of the wrapper with the run-time context resembles the following:

```
import static java.lang.System.out;

import org.openl.rules.context.DefaultRulesRuntimeContext;
import org.openl.rules.context.IRulesRuntimeContext;
import org.openl.rules.runtime.RuleEngineFactory;
import org.openl.runtime.IEngineWrapper;
import org.openl.vm.IRuntimeEnv;
public class ExampleOfUsingRuntimeContext {

              public static void main(String[] args) {
                     //define the interface
```

```
                    RuleEngineFactory<simple> rulesFactory = new RuleEngineFactory<Simple>("
TemplateRules.xls", Simple.class);
                    simple rules = rulesFactory.newInstance();
                    // Getting runtime environment which contains context
                    IRuntimeEnv env = ((IEngineWrapper) rules).getRuntimeEnv();
                    // Creating context (most probably in future, the code will be different)
                    IRulesRuntimeContext context = RulesRuntimeContextFactory.
buildRulesRuntimeContext();                    env.setContext(context);
                    context.setLob("Home");
                    rules.hello1(12);

      }
}
```

Run this class. In the console, ensure that the rule with **lob = Home** was executed. With the input parameter **int = 12,** the **It is Afternoon, Guys** phrase is displayed.

Using OpenL Tablets projects from Java code

OpenL Tablets projects can be instantiated via `SimpleProjectEngineFactory`.This factory is designed to be created via `SimpleProjectEngineFactoryBuilder`. You should configure a builder. The main builder method is `setProject(String location)` You should specify project location folder via this method.

Following example instantiate OpenL Tablets project:

```
ProjectEngineFactory<Object> projectEngineFactory = new
SimpleProjectEngineFactory.SimpleProjectEngineFactoryBuilder<Object>().setProject(<project
location>) .build();
Object instance = projectEngineFactory.newInstance();
```

Above example instantiate OpenL Tablets project with generated in runtime interface. You can invoke a method from instantiated project via reflection mechanism. ProjectEngineFactory returns generated interface via `getInterfaceClass()` method;

If you want to use static interface you should specify it in `SimpleProjectEngineFactoryBuilder` Following example illustrate how to instantiate a project with static interface.

```
SimpleProjectEngineFactory<SayHello> simpleProjectEngineFactory = new
SimpleProjectEngineFactoryBuilder<SayHello>().setProject<project location>)
                .setInterfaceClass(SayHello.class)
                .build();
SayHello instance = simpleProjectEngineFactory.newInstance();
```

`SimpleProjectEngineFactoryBuilder` has additional methods to configure an engine factory. For example method `setWorkspace()` define a projects workspace for dependent projects resolving. You can change execution mode via `setExecutionMode()` method, by default runtime execution mode is enabled. If instance class should provide runtime context you should specify it via `setProvideRuntimeContext(true).`

OpenL Tablets WebStudio supports compilation of a module from project in single mode. You can compile a module from project in single module via `setModule(String moduleName)`. If this method was used with module name “single mode” compilation will be used for this module from project.

How to access to Test Table from Java Code

Test results can be accessed through the test table API. For example, the following code fragment executes all test runs in a test table called **insuranceTest** and displays the number of failed test runs:

```
RuleEngineFactory<?> rulesFactory = new RuleEngineFactory<?>("Tutorial_1.xls");
IOpenClass openClass = rulesFactory.getCompiledOpenClass();
IRuntimeEnv env = SimpleVMFactory.buildSimpleVM().getRuntimeEnv();
Object target = openClass.newInstance(env);
IOpenMethod method = openClass.getMatchingMethod("testMethodName", testMethodParams);
TestUnitsResults res = (TestUnitsResults) testMethod.invoke(engine, new Object[0], env);
```

Generating Java Classes from Datatype Tables

Some rules require complex data models as input parameters. Developers should generate classes for each datatype defined in Excel file for using them in static interface as method arguments. The static interface can be used in engine factory. For an example of how to create and use a wrapper, see Using OpenL Tablets rules from Java Code

Note: Datatype is an OpenL table of Datatype type that is created by a business user and defines custom data type. Using these data types inside the OpenL Tablets rules is recommended as the best practice. For detailed information about datatypes see [OpenL Tablets - Reference Guide], Datatype Table.

To generate a datatype classes, proceed as follows:

1. Use Maven
   a. Configure OpenL maven plugin as described in Configuring OpenL Maven Plugin
   b. Run maven script
2. Use Ant
   a. Configure the Ant task file as described in Configuring the Ant Task File
   b. Execute the Ant task file

Configuring OpenL Maven Plugin

To generate interface for rules and datatype classes defined in the MS Excel file, add following maven configuration to your pom.xml file:

```
<build>
  [...]
  <plugins>
      [...]
      <plugin>
          <groupId>org.openl.rules</groupId>
          <artifactId>openl-maven-plugin</artifactId>
          <version>${openl.rules.version}</version>
          <configuration>
              <generateInterfaces>
                  <generateInterface>
                      <srcFile>src/main/openl/rules/TemplateRules.xls</srcFile>
                      <targetClass>
                        org.company.gen.TemplateRulesInterface
                      </targetClass>
                  </generateInterface>
              </generateInterfaces>
          </configuration>
          <executions>
              <execution>
                  <goals>
                      <goal>generate</goal>
                  </goals>
              </execution>
          </executions>
      </plugin>
```

```
  </plugins>
  [...]
</build>
```

In this case classes and rules project descriptor (rules.xml) will be generated on each maven run on generate-sources phase.

Each `<generateInterface>` section has a number of parameters. The following table describes `<generateInterface>` section parameters.

Name	Type	Required	Description
srcFile	String	true	Reference to the Excel file for which an interface class must be generated.
targetClass	String	true	Full name of the interface class to be generated. OpenL Tablets WebStudio recognizes modules in projects by interface classes and uses their names in the user interface. If there are multiple wrappers with identical names, only one of them is recognized as a module in OpenL Tablets WebStudio.
displayName	String	false	End user oriented title of the file that appears in OpenL Tablets WebStudio. Default value is Excel file name without extension.
targetSrcDir	String	false	Folder where the generated interface class must be placed. For example: "src/main/java". Default value is: "${project.build.sourceDirectory}"
openlName	String	false	OpenL configuration to be used. For OpenL Tablets, the following value must always be used: org.openl.xls. Default value is: "org.openl.xls"
userHome	String	false	Location of user-defined resources relative to the current OpenL Tablets project. Default value is: "."
userClassPath	String	false	Reference to the folder with additional compiled classes that will be imported by the module when the interface is generated. Default value is: null.
ignoreTestMethods	boolean	false	If true - test methods will not be added to interface class. Used only in JavaInterfaceAntTask. Default value is: true.
generateUnitTests	boolean	false	Overwrites base generateUnitTests value
unitTestTemplatePath	String	false	Overwrites base unitTestTemplatePath value
overwriteUnitTests	boolean	false	Overwrites base overwriteUnitTests value

More configuration options you can find in [OpenL Tablets Maven Plugin Guide].

Configuring the Ant Task File

The following is an example of the build file:

```
<project name="GenJavaWrapper" default="generate" basedir="../">
      <taskdef name="openlgen" classname="org.openl.conf.ant.JavaWrapperAntTask"/>

<target name="generate">
```

```
        <echo message="Generating wrapper classes..."/>

        <openlgen openlName="org.openl.xls" userHome="."
                srcFile="rules/Rules.xls"
                targetClass="com.exigen.claims.RulesWrapper"
                displayName="Rule datatypes"
                targetSrcDir="gen"
        >
        </openlgen>

        <openlgen openlName="org.openl.xls" userHome="."
                srcFile="rules/Data.xls"
                targetClass=" com.exigen.claims.DataWrapper"
                displayName="Data datatypes"
                targetSrcDir="gen"
        >
        </openlgen>

</target>
</project>
```

When the file is executed, it automatically creates datatypes Java classes for specified Excel files. The Ant task file must be adjusted to match contents of the specific project.

For each Excel file, an individual `<openlgen>` section must be added between the `<target>` and `</target>` tags.

Each `<openlgen>` section has a number of parameters that must be adjusted. The following table describes `<openlgen>` section parameters:

Parameters in the <openlgen> section	
Parameter	**Description**
`openlName`	OpenL configuration to be used. For OpenL Tablets, the following value must always be used: `org.openl.xls`
`userHome`	Location of user defined resources relative to the current OpenL Tablets project.
`srcFile`	Reference to the Excel file for which a wrapper class must be generated.
`targetClass`	Full name of the wrapper class to be generated.
`displayName`	End user oriented title of the file that appears in OpenL Tablets WebStudio.
`targetSrcDir`	Folder where the generated wrapper class must be placed.

Data and data types handling in OpenL Tablets

Datatype lifecycle

1. Create a Datatype table in rules file.
1. At runtime for each datatype, java class is generated (see Byte code generation at runtime).
2. If the Generating Java Classes from Datatype Tables was used, the appropriate generated java classes should be included in classpath (see Java files generation)

Inheritance in Datatypes

There is a possibility to inherit one datatype from another in OpenL Tablets. New data type that inherits from another one will have access to all fields defined in the parent data type. If a child datatype contains fields that are defined in the parent datatype you will get warnings or errors (if the field is declared with different types in the child and the parent datatype).

Also constructor with all fields of the child datatype will contain all fields from the parent datatype and *toString*, *equals* and *hashCode* methods will use all fields form the parent datatype.

Byte code generation at runtime

At runtime, when OpenL engine instance is being built, for each datatype component java byte code is being generated in case there are no previously generated java files on classpath (see Java files generation) it represents simple java bean for this datatype. This byte code is being loaded to classloader so the object of type `Class<?>` can be accessed. Using this object through reflections new instances are being created and fields of datatypes are being initialized (see `DatatypeOpenClass` and `DatatypeOpenField` classes).

Important! If you have previously generated java class files for your datatypes on classpath, they will be used at runtime. And it doesn't matter that you have made any changes in Excel. To apply these changes, remove java files and generate java classes from Datatype tables.

Java files generation

As generation of datatypes is performed at runtime and developers can't access these classes in their code, Generating Java Classes from Datatype Tables mechanism is introduced which gives the possibility of generating java files and putting them on the file system. So users can use these data types in their code.

[OpenL internals]Access datatype at runtime and after building OpenL wrapper

After parsing, each data type is put to compilation context, so it will be accessible for rules during binding. Also all data types are placed to `IOpenClass` of whole module and will be accessible from `CompiledOpenClass#getTypes` when OpenL wrapper is generated.
Each `TableSyntaxNode` that is of the *xls.datatype* type contains an object of data type as its member.

Data Table

A **data table** contains relational data that can be referenced as follows:

from other tables within OpenL Tablets
from Java code through wrappers as Java arrays
through OpenL Tablets run-time API as a field of the `Rules` class instance

Data int numbers
this
Numbers
10
20
30
40
50

Figure 6: Simple data table

In the example above, information in the data table can be accessed from Java code as shown in the following code example:

```
int[] num = tableWrapper.getNumbers();

for (int i = 0; i < num.length; i++) {
       System.out.println(num[i]);
}
```

where `tableWrapper` is an instance of the wrapper class of the Excel file.

Datatype Person	
String	name
String	ssn

Data Person p1	
name	ssn
Name	**SSN**
Jonh	555-55-0001
Paul	555-55-0002
Peter	555-55-0003
Mary	555-55-0004

Figure 7: Datatype table and a corresponding data table

In Java code, the data table `p1` can be accessed as follows:

```
Person[] persArr = tableWrapper.getP1();

for (int i = 0; i < persArr.length; i++) {
       System.out.println(persArr[i].getName() + ' ' + persArr[i].getSsn());
}
```

where `tableWrapper` is an instance of the Excel file wrapper.

Customizing Table Properties

OpenL Tablets design allows customizing available Table properties. The Engine employs itself to provide support of properties customization. The `TablePropertiesDefinitions.xlsx` file contains all declaration required to handle and process Table properties.

Updating table properties requires recompiling the OpenL Tablets product. Contact your OpenL Tablets provider to retrieve the table properties file. When the changes are made, send the file back to the provider, and a new OpenL Tablets package will be delivered to you.

Alternative you can recompile OpenL Tablets from sources on your own.

Table Properties Dispatching

Previously selecting tables that correspond to current runtime context have been processed by java code. Now the rules dispatching is the responsibility of generated Dispatcher Decision table (Such a table will be generated for each group of methods overloaded by dimension properties). The Dispatcher table works like all decision tables so the first rule matched by properties will be executed even if there are several tables matched by properties (Previously in java code dispatching we would had got an AmbiguousMethodException in such a case). So to support both of functionalities we present system property "**dispatching.mode**" that has two possible values:

- **java**: dispatching will be processed by java code. The benefit of such approach is stricter dispatching: if several tables are matched by properties, the AmbiguousMethodException will be thrown.
- **dt**: dispatching will be processed by Dispatcher Decision Table. The main benefit of this approach is performance: decision table will be invoked much faster than java code dispatching performed.

The Java approach will be used by default (if the system property is not specified) or if that "**dispatching.mode**" property has a wrong value.

Tables priority rules

To make tables dispatching more flexible there exists DataTable tablesPriorityRules in `TablePropertiesDefinitions.xlsx`. Each element of this table defines one rule of how to compare two tables using their properties to find more suitable table if several tables was matched by properties. Priority rules will be used sequentially in comparison of two tables: if one priority rule gives result of the same priority of tables then there will be used next priority rule.

Priority rules are used differently in Dispatcher table approach and java code dispatching but have the same sense: select suitable table if there was several tables matched by dimension Properties.

In case of Dispatching table priority rules are used to sort methods of overloaded group. Each row of Dispatcher table represents a rule, so after sorting, high-priority rules will be at the top of decision tables, and in case several rows of Decision table was fired only the first (of the highest priority) will be executed.

In case of java code dispatching priority rules will be used after the selecting of tables that corresponds to the current runtime context: all matched tables will be sorted in order to select one of the most priority. If it is impossible to find the most priority rule (several tables has the same priority and are of more priority than all other tables) `AmbiguousMethodException` will be thrown.

There are two predefined priority rules and possibility to implement java class that compares two tables using their properties. Predefined:

min(<property name>) – table that has lower value of property specified will be of more priority. Restrictions: property specified by name should **be instanceof Comparable<class of property value>**.

max(<property name>) – table that has higher value of property specified will be of more priority. Restrictions: the same as in **min(<property name>)**.

To specify java comparator of tables we should use such expression: **javaclass:<java class name>**. Restrictions: java class should implement Comparator<ITableProperties>.

Validation for Tables

The Validation phase follows after the binding phase that serves to checks all tables for errors and accumulate all errors.

Validators

All possible validators are stored in `ICompileContext` of OpenL class. (The default compile context is `org.openl.xls.RulesCompileContext` that is generated automatically.)
Validators get the OpenL and array of `TableSyntaxNodes` that represent tables for check and must return `ValidationResult`.
ValidationResult:

status (fail or success) and
all error/warn messages that occurred

Table properties validators

Table properties that are described in `TablePropertyDefinition.xlsx` can have constraints. Some constraints have predefined validators associated with them.

Adding own property validator:

1. Add constraint:
   1.1. Define constraint in TablePropertyDefinition.xlsx (constraints field)
   1.2. Create constraint class and add it into the ConstraintFactory
2. Create own validator
   2.1. Create class of your validator and define in method org.openl.codegen.tools.type.TablePropertyValidatorsWrapper.init() constraint associated with validator.
   2.2. If it needed you can modify velocity script RulesCompileContext-validators.vm in project org.openl.rules.gen that will generate org.openl.xls.RulesCompileContext.
   2.3. Run org.openl.codegen.tools.GenRulesCode.main(String[]) to generate new org.openl.xls.RulesCompileContext with your validator.
3. Write unit tests!

Existing validators

Unique in module validator checks uniqueness in module of some property
Active table validator checks correctness of "active" property. Only one active table.

Module Dependencies

Classloaders

Dependency class (datatype) resolution mechanism is implemented using specialized classloading.

Each dependency has its own java classloader. So all classes used while compiling specified module (including generated datatype java classes) are stored in its classloader.

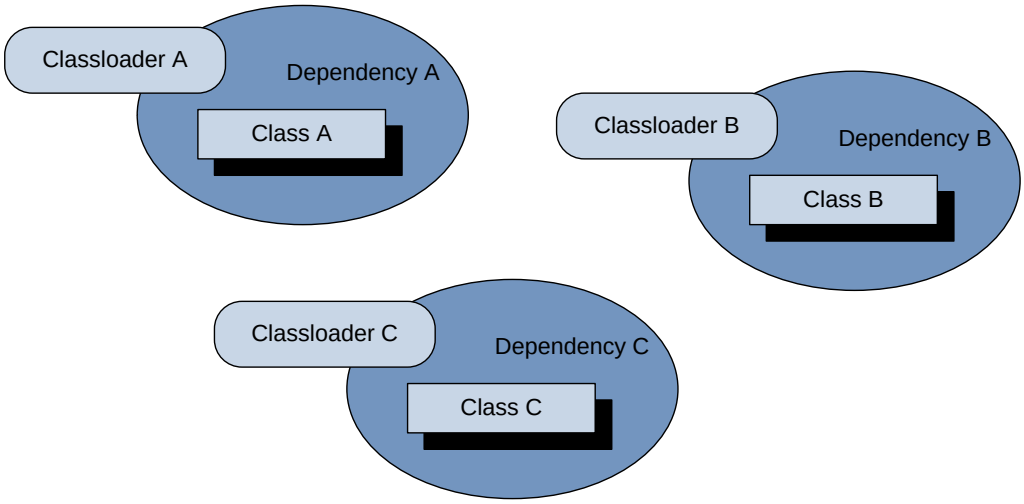

Figure 5: Dependency classloaders

The root module contains links to all his dependencies' (bundles) classloaders. When loading any class, the root module first checks his bundles classloaders if any of them contains already loaded class, as shown below. Algorithm:

1. Get all bundle classloaders
2. In each bundle classloader try to find previously loaded class.
3. If class exists, return it. If no, try to load class by this classloader.
4. Will be returned the class that can be found by any bundle classloader.

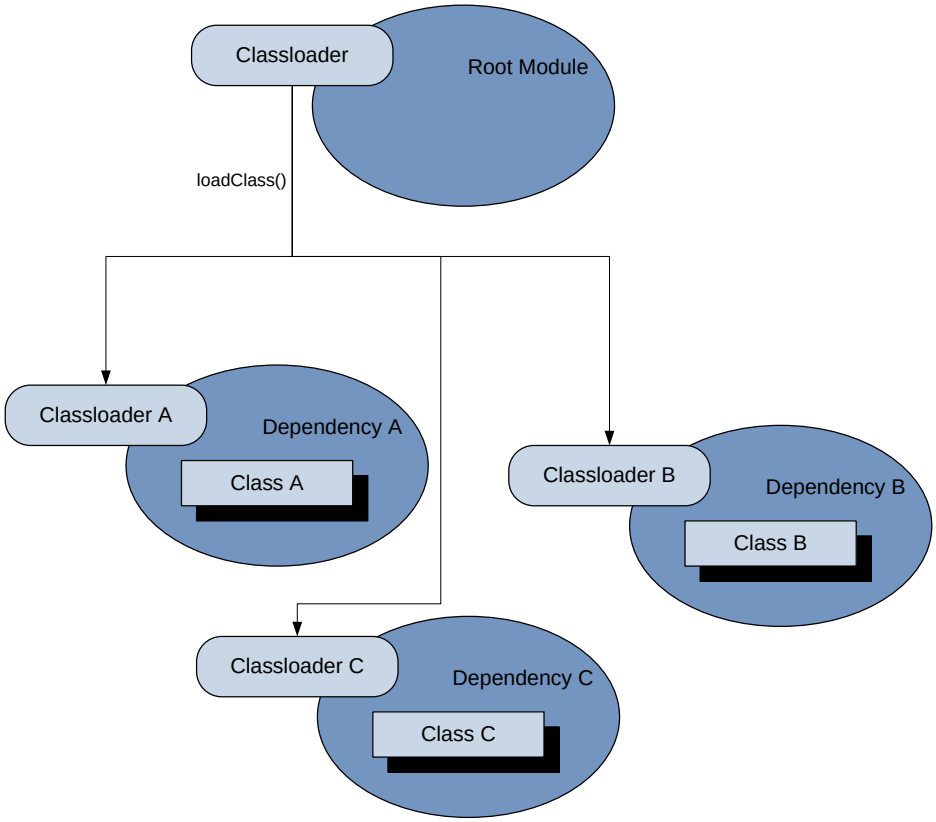

Figure 6: Load class from root module

For dependency management feature please provide appropriate DependencyManager object to the entry point for Openl compilation.

Note: Using the same class in two classloaders can cause a problem, because the class will be loaded by two different classloaders

Peculiarities of OpenL Tablets Implementation

Lookup Tables Implementation Details

At first the table goes through parsing and validation. On parsing all parts of the table such as header, columns headers, vertical conditions, horizontal conditions, return column and their values are extracted. On validation OpenL checks if the table structure is proper.

To work with this kind of table `TransformedGridTable` object is created as constructor parameters it has in original grid table of lookup table (without header) and the `CoordinatesTransformer` that converts table coordinates to work with both vertical and horizontal conditions.

As the result we get a `GridTable` and works with it as a decision table structure, all coordinates transformations with lookup structure goes inside. Work with columns/rows is based on physical (not logical) structure of the table.

Range Types Instantiation

There are 3 ways to create IntRange:

1. `new IntRange(int min_number, int max_number)` – it will cover all the numbers between `min_number` and `max_number`, including borders.
2. `new IntRange(Integer value)` – it will cover only given value as the beginning and the end of the range.
3. `new IntRange(String rangeExpression)`. Borders will be parsed by formats of `rangeExpression`.

The same formats and restrictions are used in DoubleRange.

Index