The MOGLi Code Creator

Just another code generator? â€“ NO! â€“ Itâ€™s THE Lightweight autogeneration tool!
It is a small standalone Tool for a quick start into model based development!
Itâ€™s written in Java but made to generate more than only Java code.

MOGLi stands for the following attributes:

M odel based
O pen for extension
G enerator based
Li ghtweight

Current version: 1.4.0

Content

1. Why is MOGLiCC lightweight?
There are several aspects that illustrate the lightweight character of the MOGLi Code Creator:

a) The programm size which is only a few megabytes,
b) The small number of prerequisites to run it - which is one: a Java 1.6 Runtime,
c) The tooling needed to work with it - which is a simple text editor,
d) The ease of use by simply integrating it into your IDE (Eclipse: RunAsâ€¦), and
e) The main reason: the small knowledge you need to learn to work with it.

With the simplest application form (See 4. How to apply MOGLiCC?) you only need to define your classes by a very simple DSL and you are able to create all java classes for the MOGLiCC JavaBean Group (See 5. Example Case).

Top

2. Why is MOGLiCC open?
Firstly, because it is open for extension by plugins. Secondly, because it is open source. Therefore, feel free to have a look at the moglicc sources at GitHub. The MOGLiCC tool contains a small plugin framework that allows loosely coupled plugins to interact with each other (for more information about the software design see 6. How is MOGLiCC designed?). If the standard set of functionality does not match you purpose, create your own plugins that do. For example, it may be nice to model your domain objects using a graphical tool. To use this

graphical model with MOGLiCC, export the object definitions as, e.g. XHaving written an XMI-Model-Provider plugin you would be able to connect the graphical model tool with MOGLiCC and use it without manually defining a model.

Top

3. How does MOGLiCC work?

Think of MOGLiCC as a reactor for two separate source code ingredients: a model and a generator. The MOGLiCC plugin called StandardModelProvider allows you to define a model in a simple text file. Another MOGLiCC plugin called VelocityClassBasedFileMaker allows you to define a template file. Now MOGLiCC works like this: model + template = code. Therefore, MOGLi is based on two equally important parts: model and generator.

For the actual merging of model and template the current plugins of MOGLiCC use the Velocity Framework. However, you will see no Velocity source code. The Velocity engine is created, configured and started by MOGLiCC. You just need to feed the engine with data, i.e. templates. This means, you have to learn the simple but powerful Velocity Template Language (VTL) if you wish to create your own source code artefacts. If you donâ€™t like Velocity and its template DSL, write your own generator plugin â€“ remember: MOGLi is open for extension.

There are three application forms of MOGLiCC that are worth destinguishing:

a) **Define your own model**: use the default templates of the MOGLiCC JavaBean Group. To do so, you only need to specify your model (class definitions).
b) **Build your own artefacts**: create your own templates to create completely new artefacts.
c) **Develop your own plugin**: write your own MOGLiCC plugins, plug them into the MOGLi Code Creator and let them do their jobs.

There are three abstraction levels on which â€œparallel functionalityâ€ can arise and where code generation can help (see 8. What are the benefits of autogenerating source code?):
a) **Level of attributes**: data classes consisting of many attributes may have to handle each variable in a similar way many times, e.g. when building sql-scripts for the database, when creating dao objects or when converting/mapping to other data structures. For all kinds of modification all source code locations concerned can be modified automatically by MOGLiCC.
b) **Level of classes**: data model consisting of many different data classes may have to generate for each data class a number of other classes, e.g. java bean class, DAO class, validator class, UnitTest classes. For modification of data class all other classes concerned can be modified automatically by MOGLiCC.
c) **Level of projects**: in component oriented architectures a new project has to be created frequently for each new component. The project skeleton is usually similar between and can be generated automatically by MOGLiCC.

Top

4. How to apply MOGLiCC?

The MOGLi Code Creator is released as a zip-file. After unzipping its content, a lib directory containing the MOGLiCC jar files and a start batch file is present. Having set the environment variable JAVA_HOME pointing on a 1.6 JRE, the Code Creator will do its job when calling the start batch. When called the first time, some directories are created and a number of default files are extracted from the jar source files. The most important directories are â€œinputâ€ and â€œoutputâ€ . For each plugin that needs input data, the input directory contains a subdirectories named after the plugin the data belongs to. The same is true for the output dir, but in addition old output files are deleted on every application start. The input directory together with the corresponding output directory can be organized in different workspaces. The â€œapplication.propertiesâ€ file allows you to move the default workspace directory â€“ the application root directory â€“ to an arbitrary file system folder. In this way, you can manage code generation for different projects. For more information how to configure a workspace directory read the default application properties file.

You may wish to deactivate a certain generator plugin, because, for instance, you do not wish its result being generated. For that purpose you can use the workspace properties file. For more information about plugin deactivation read the default workspace properties file.

For each workspace, a log directory exists containing the log output of the last application run. With every application start the old logfiles are deleted. The main log file is named â€œ_MOGLi.logâ€ and contains log output of the core module (See 6. How is MOGLiCC designed?). For any problem analysis, the end of this file is the first place to look. Each plugin has its own log file with additional plugin specific information.

Top

4.1 Defining a model

A model is defined by its model file. Model files are located in the plugin specific input directories of the StandardModelProvider. If only one file is found there, it is selected automatically. Otherwise, the model file must be defined in the model properties file. It can be used to manage more than one model file in one workspace. For more information about model file configuration read the default model properties file.
The whole idea of the metamodel of the StandardModelProvider is, that there are three information levels: model, class and attribute. The only concrete data for each element in each level is only â€œnameâ€ , i.e. for the model and each class in this model and each attribute in all of the classes you only define a name. All other information is defined by MetaInfo elements. MetaInfos can be defined for the model, a class or a single attribute. From the template file, these MetaInfos are addressable. Hence, model file and template must match very well to have reasonable output. The best way to understand this model is to have a look at the default model file â€œMOGLiCC_JavaBeanModel.txtâ€ . It contains both a description of the model definition DSL and an example, the MOGLiCC_JavaBeanModel, how the model DSL is used. If this file has been deleted in your current workspace, define a new workspace in the application property file. Having executed the application with the new workspace, the â€œMOGLiCC_JavaBeanModel.txtâ€ file will be available there again.

Top

4.2 Creating an artefact

Artefacts of a specific generator have to be defined in its specific form. It is common for all generators, that artefact properties must be defined. For the VelocityClassBasedFileMaker and the VelocityModelBasedLineInserter this is done in the header of the main template file. For the VelocityModelBasedTreeBuilder this is done in the artefact properties file. These properties define all necessary artefact metadata such as the generation target directory.

For more detailed information follow the links	Artefact defined in	Artefact consists of	Artefact properties	Max Number

			defined in	of artefacts
VelocityModelBasedLineInserter	subdir of the pluginâ€™s input folder	Number of template files	Header of one or more main templates	No limit
VelocityClassBasedFileMaker	subdir of the pluginâ€™s input folder	Number of template files	Header of the main template	No limit
VelocityModelBasedTreeBuilder	subdir of the pluginâ€™s input folder	Artefact properties file and a file structure	Artefact properties file	No limit

Each artefact is named by the subdirectory containing its input data. For the VelocityClassBasedFileMaker and the VelocityModelBasedLineInserter a main template must be defined if more than one template file exist. A main template has syllable â€œMainâ€ within its name. Other templates may serve as subtemplates used by the main template or by another subtemplate.

Within a pluginâ€™s input directory may exist subdirectories that do not represent artefacts, e.g. meta information of your version control system or a collection of subtemplates used for more than one artefact. For these subdirectories, you can tell the generator to ignore these for code generation (See default generator properties file).

Artefact properties start with the annotation symbol â€˜@â€™. The header of template files contains a list of such artefact properties, e.g. name of file to create or to insert in and where to insert within the target file. The body of a template file contains the actual generation content. Both header and body of the template file are merged with model data. This means, place holder will be replaced by model data both in the header and the body. However, the header does not contribute to the generated content (body) and controls only what the plugin does with the generated content.

The artefact attributes â€œFileNameâ€ and â€œTargetDirâ€ determine the target file. If the target file exists, the attribute â€˜CreateNewâ€™ controls whether the existing file is overwritten or not. If the target file does not exist, the inserter plugin will generate an error, the generator plugin will simply not create a target file. In any case, however, both plugins write the generated content into their plugin specific output dir. But after the next application start, this output will be deleted.

The behaviour of Velocity Template Language (VTL) concerning white space is quarrelsome, intensively discussed and strict: leading white space and empty lines are simply removed. This is troublesome for formatting purpose. To increase the easeness to create well formatted content, the MOGLiCC generators apply a little parser after merging model and templates. This MOGLiCC specific parser removes leading apostrophes <â€™> from each line of the generated content. An apostrophe prevents Velocity from removing the whitespace that follows it. With this parser it is possible to write both well formatted templates and well formatted generation result. Use white space before an apostrophe to format only a template line. Use white space after an apostrophe to format both the template line and the corresponding line of the generation result.

When writing template files or artefact properties files, you should have in mind that not only the generator plugins contribute to the list of functions you can use. See also StandardModelProvider and VelocityEngineProvider for more information.

Top

4.3 The MetaInfo Validation Feature
The advantage of the MOGLiCC Standard Model is its simpleness, its disadvantage the large number of MetaInfo elements and the danger to lose the overview which MetaInfo element is really being used. This is one reason for the MetaInfo Validation Feature. A second reason is the importance specific MetaInfo elements can have, for instance, without a MetaInfo element â€œJavaTypeâ€ it would not be possible to build a JavaBean artefact with a reasonable content. It would be good to assure that for all classes in the model a java type is defined. Therefore the existing generator plugins implement the interface MetaInfoValidatorVendor and its method â€œgetMetaInfoValidatorList()â€ . MetaInfoValidators are defined in the MetaInfo validation file called â€œMetaInfo.validationâ€ . This file is located in the plugin input directory of the generator plugin. For more information how to define MetaInfoValidators, see the default MetaInfo validation file. The StandardModelProvider collects all of these MetaInfoValidators and uses this information for two purposes: 1. to throw an exception if a mandatory MetaInfo element is missing and 2. to create a statistics file that informs which MetaInfo element is used in which context and that warns if a MetaInfo element or a MetaInfo Validator is unused. You find this statistics file in the plugin output directory of the StandardModelProvider.

Top

4.4 VelocityClassBasedFileMaker vs. VelocityModelBasedLineInserter
There are four differences between these two generators. First, only the inserter can insert into existing files, whereas the generator can only create new files. Second, the artefact property â€œTargetDirâ€ is mandatory for the inserter but not so for the generator. Third, â€œclass basedâ€ means, that for each class defined in the model an artefact output file, e.g. a JavaBean, is created. â€œModel basedâ€ means that an insertion is made for the whole model in only one existing file (or only one artefact output file is created). Fourth, for the generator there must exist only one main template, for the inserter more than main templates are possible, but they must reference the same target file, allowing to insert into different sections within the same target file.

The concept of class based and model based plugins may appear difficult. The template of an artefact of a class based generator is merged with the whole model that results into one output artefact. This is useful e.g. for generating a single application wide configuration file that contains for each class in the model some lines with configuration instructions (hibernate configuration for instance). In short: Template + Model definition = one target file. Instead, the template of an artefact of generator is not merged with the whole model but with each class within the model. For 10 classes in the model this results in 10 output artefacts for the artefactâ€™s template. This is useful e.g. for generating java beans for a group of classes. In short: Template + Class Definitions = many target files.

Top

5. Example case: The MOGLiCC JavaBean Group

The probably most typical example for code generation is to create simple data classes, in Java, JavaBeans. Therefore, MOGLiCC comes with a set of default templates for generating a number of artefacts that closely belong together: The MOGLiCC JavaBean Group. This group of artefacts consists of a JavaBean class, a builder class for easy creation of JavaBean instances with example data, a validator class to check a JavaBean instance for valid data and two JUnit test classes to demonstrate that the generated code is runnable and works without errors.

Because these JavaBean Group artefacts need to be created for each class within the model, the templates are default input files of the VelocityClassBasedFileMaker plugin. If you do not like defining your own model, you can use the default model file â€œMOGLiCC_JavaBeanModel.txtâ€ which describes a few arbitrary objects just for illustration purpose. After starting the MOGLiCC application you find the generated artefacts in â€œ<workspaceDir>/output/ VelocityClassBasedFileMakerâ€ . Create a test project and move the generated java files into it. Then, you are able to execute the JUnit test cases.

The most complex class is the JavaBean. It is by default created with the following methods: toString, hashcode, equals and compare. If the Clonable interface is found in the class definition, the clone method is additionally generated. If the Serializable interface is found in the class definition, a SerialVersionUID is generated. To assure maintainablity the JavaBean main template uses a large number of subtemplates. Some of them even use further subtemplates. If you wish to study the Velocity Template Language (VTL), the templates of the MOGLiCC JavaBean Group provide a lot of snippets to learn from. Generally, you find the VTL refererence guide here: http://velocity.apache.org/engine/devel/vtl-reference-guide.html.

Top

6. How is MOGLiCC designed?

The basic principle in designing the MOGLi Code Creator was â€œLoosely couplingâ€ . Following the articles Raus aus der Dependency HÃ¶lle and Dependency-Management von technischen Standardkomponenten the MOGLiCC architecture is component-oriented and the application consists of a number of different components. Some of them represent business logic components â€“ in terms of the articles: â€œfachliche Komponentenâ€ .

The heart of the application is the business logic component â€œcoreâ€ which actually represents a container for MOGLiCC plugins. With each application start, a special plugin folder is scanned for jar-files that contain a file named â€œmogli.propertiesâ€ . From this file a fully qualified name of the plugin starter class is read and used to instantiate the starter class via reflection. The starter classes of all plugins are cast on the common MOGLiCC interface type â€œMOGLiPluginâ€ . After loading the starter classes, the core component analyses the dependencies defined by the â€œgetDependenciesâ€ method each MOGLiPlugin implements. If all dependencies are resolvable the plugins are sorted in the execution order. In this way it is guaranteed that, for instance, the model provider is always executed before the generators.

The four other business logic components are the four plugins StandardModelProvider, VelocityEngineProvider, VelocityClassBasedFileMaker and VelocityModelBasedLineInserter. Before executing these plugins, the core prepares each plugin for execution and sets an infrastructure service object to each plugin. This service provides infrastructure functionality to the plugins such as â€œgetPluginInputDir()â€ or â€œgetModelProvider(id)â€ . Finally, the core calls the MogliPlugin method â€œdoYourJob()â€ for each plugin to execute it.

The core component has two more responsibilities. First, it checks whether a help directory exists for each plugin. If not, the core triggers the corresponding plugin to extract its embedded help files. Second, the core checks whether an input directory exists for each plugin. If not, the core triggers the corresponding plugin to extract its embedded default input files. In this way, a number of directories and files are created when the application was started the first time.

There are four basic MOGLiCC plugin types. 1. ModelProvider, 2. EngineProvider, 3. Generator and 4. DataProvider. The StandardModelProvider is of course a ModelProvider; VelocityEngineProvider an EngineProvider; VelocityClassBasedFileMaker and VelocityModelBasedLineInserter are Generators. â€œInserterâ€ is a subtype of a â€œGeneratorâ€ . A DataProvider is currently not yet implemented.

An EngineProvider typically represents a plugin without an own input and output directory. It does nothing when called directly from the core. The VelocityEngineProvider creates, configures and actually starts a velocity engine automatically. Thus, the generator plugins using the VelocityEngineProvider have nothing to do with any Velocity code. A generator plugin uses an EngineProvider by setting a specific EngineData object to the EngineProvider and triggers it to start the engine. The EngineData gives the EngineProvider all information necessary for the generation, i.e. the model and the template(s). Having done its job the EngineProvider returns a specific ResultData object to the generator which contains all information for the generator to create the output files.

Top

7. How is MOGLiCC implemented?

The author of the moglicc source code followed the principles of the clean code developer (http://www.clean-code-developer.de). One rule of the clean code developers is â€œTest firstâ€ . To apply this rule, the source code author also followed the Test Driven Development method of â€œGrowing Object-Oriented Software, Guided by Testsâ€ by Freeman & Pryce. Thus, the MOGLiCC functionality of the current version is automatically and regularly tested by 384 unit tests (70-90% coverage), 49 integration tests (10-40% coverage) and 40 system tests.

Top

8. What is the generation result of the MOGLi Code Creator?

Each generator has its own type of result:

VelocityModelBasedLineInserter	one or more lines within an existing file or a single new file
VelocityClassBasedFileMaker	one or more complete new files
VelocityModelBasedTreeBuilder	a directory of subdirectories and files

Common for all generators is, that the generatorâ€™s result is written twice. First, in the output directory of the application root directory. This is the central location for all generation results created by a single application run. For each application start this output directory is cleaned, thus, it contains only newly generated results. Second, in the target directory defined in the artefact properties. There are two benefits in this procedure: 1. For many different artefacts, the second generation results may be highly distributed in your file system. For a quick (over-)view, the applicationâ€™s output directory is very convenient. 2. The second generation result creates the code at the very location where you actually want to have it, where you work with it (e.g. your IDE workspace). However, you may not allow the MOGLiCC to overwrite already existing code in your workspace (by using the artefact property CreateNew=false). Anyway, you find the generation result in the applicationâ€™s output directory. In this way, you are able to compare the preserved code in your workspace with the new one.

Top

9. What are the benefits of autogenerating source code?
There is only one final purpose of code autogeneration: speeding up software development. To gain this final benefit, two aspects are important to have in mind:

a) The benefit increases with the amount of â€œparallel functionalityâ€ (see abstraction levels in 3. How does MOGLiCC work?). If a specific design concept is used again and again, parallel functionality exists, for instance, a number of modules or components with a similar internal structure to be embedded in a similar environment. The higher the number of components (repetitions of the parallel functionality) and the higher the complexity of the parallel functionality, the higher is the benefit of autogeneration. Is this complexity low (letâ€™s say, only a few lines in one file are needed to be generated) the number of repetitions must be very high to benefit from autogeneration. Is the complexity high (many lines in many different files) the benefit starts at least with the third â€œparallel componentâ€ (the second repetition). If you foresee sufficient complexity and repetitions of parallel functionality, start using autogenerating with the first repetition.
b) There are three proximate benefits that contribute to the final one:
   I) Reduction of non-creative coding effort.
   Given some minutes to write, extend or modify a model, you generate the code within seconds. In contrast, the manual typing of the code may cost hours of development time even if the programmer(s) exactly know where to create which code.
   II) Reduction of costs for fixing stupid bugs.
   Typically, there are three types of stupid errors caused by programming parallel functionality manually. 1. copy-paste-errors, 2. misspelled expressions and 3. forgotten source locations where an adaptation is necessary. Cost of bug fixing can increase the coding effort dramatically and fixing stupid errors decreases the motivation of the most developers. In contrast, learning to control an autogeneration tool is fun for many developers.
   III) Documentation of â€œHow toâ€ knowledge
   Have you ever heard a statement like â€œSource code is the best documentationâ€ . This may be true for clean code in its best appearance. However, source code is frequently the only documentation. Now imagine a team of developers which has designed and implemented a complex system. The team members change and after a while the knowledge how to set up a new module in the system dies away. Familiar situation? Given the system is developed by the MOGLi Code Creator, the developer simply study the templates for the artefacts generated by the VelocityClassBasedFileMaker and the insertions done by the VelocityModelBasedLineInserter and the knowledge where to create which code is quickly available.

Top

10. Road Map
The following extensions are planned:

a) A new MOGLiPlugin that reads sample data from input files and creates a java class â€˜SampleDataStorage.javaâ€™. This class will allow accessing the sample data programmatically and will belong to the MOGLiCC JavaBean Group.
b) A new artefact for the VelocityClassBasedFileMaker called â€˜JavaBeanFactory.javaâ€™. This Factory will use SampleDataStorage to create instances of the generated Java Bean bases on the sample data and will also belong to the MOGLiCC JavaBean Group.
c) A new own MOGLiPlugin for the responsibility â€œValidation of MetaInfo elementsâ€ which is currently partly done by the model provider and partly by the generator plugins.
d) A new property â€œMetaInfoValidationModeâ€ with the states: OFF, NICE, STRICT
e) A new report file for provider plugins into a new application root subfolder â€œreportâ€ . This will also be the future location of the generator report.
f) A â€œattention.errorâ€ file generated into the application root folder in case at least one plugin has not been executed successfully.