Code generation classes

this document explains the programming behind the code generation

assumed is that you have read these documents

- Metatag reference
- Program class diagram

and have access to the source code

ServiceExport2Projects.java

The service starts with the controller having a database selected.

The linked projects are iterated, and for each project all linked metacode maps are iterated.

Each project / metacode combination is sent to Export2Projects for processing.

Export2Projects.java

in short, this class

- copies all content of the metacode maps
- translates project, table and view metatags
- send each metacode file to Metacodeprocessor

processmetacode(Sourceproject sourceproject, String metacodename)

this method call 2 private methods:

1. processmetacodemapstructure(File metacodedir, String projectdir)

which calls processmetacodemapstructure(File metacodedir, String projectdir, Table tablecontraint)

with null as last argument (tableconstraint)

All content of the metacodedir is read, duplicated and all names with a project metatag are translated in the resulting projectdir.

> **Directories** with table or view metatags are translated as a repeater, meaning that for each table of view a new directory will be made and recursively sent again to processmetamapstructure method, but with the active table/view as tableconstraint.
>
> This tableconstraint means that any table or view metatag will not be handled as a repeater anymore, but will handle only the table referenced in tableconstraint.
>
> Directories without metatags are duplicated “as is'.
>
> **Files**
>
> Metacodeprocessor is initialized with the source and destination file paths, database properties and the program language settings.
>
> *Metacodeprocessor(String metacodefilename, String destinationpath, Xmlreader dbproperties, Programlanguage planguage)*
>
> Files with table or view metatags are translated as a repeater, meaning that for each table of view a new a new file will be made.
>
> This result is sent to the Metacodeprocessor with processTable(String metacodename, Table table)
>
> Files without metatags are sent to the Metacodeprocess with processFile(String metacodename, Table tablecontraint)

2. copydirectory(String source, String destination)

this reads the projectroot map of the metacode and copies all content to the project root map

This method is straight forward and does not need further explanation.

Metacodeprocessor.java

Metacodeprocessor

Metatagsprocessor and repeaters are explained in the next chapter

processTable(String metacodename, Table table)

process tables and views.

If in the destinationpath this file already exists, then this old file is read.

All code blocks enclosed in “//Custom code, do not change this line” is copied and placed in the same place in the metacode.

for **table**

processtablecontent(StringBuilder code, Xmlreader.Table table)

this method runs

RepeattablesProcessor.processTable

processTableViewRepeaters(StringBuilder code)

for **view**

processviewcontent(StringBuilder code, Xmlreader.Table view)

this method runs

RepeatfieldsProcessor

processTableViewRepeaters(StringBuilder code)

processTableViewRepeaters(StringBuilder code)

Repeattables_viewsProcessor

RepeatviewsProcessor

RepeattablesProcessor

Metatagsprocessor.processDatabasetoolname

Metatagsprocessor.processProjectname

Metatagsprocessor.processDate

replacecustomcodeblocks(code)

processFile(String metacodename, Table tablecontraint)

If in the destinationpath this file already exists, then this old file is read.

All code blocks enclosed in “//Custom code, do not change this line” is copied and placed in the same place in the template code.

processTableViewRepeaters(StringBuilder code) is performed. There are use cases on the client side to have all tables/views available in one object.

If this metacode file is already restricted to 1 table or view, meaning it is in a (sub) directory with _table_ or _view_ in the name, processtablecontent or processviewcontent method is performed as described above.

Repeaters (.java)

repeaters are all subclasses of RepeatProcessor and use the same logic

RepeatProcessor & Codefragment

These 2 classes work closely together.

A repeater is defined by it’s repeater metatag (ex. :repeatfields:) and the constraint metatags

For a given Programlanguage instance and code block, a repeater will

- search for enclosed code blocks with the repeater metatag as start and end (codefragment.findtag())
- isolate a the code fragment (first found)
- translate a code fragment as defined in the subclass method processrepeator
- replace the found code fragment with this result

and repeat this process until all occurrences of the repeater metatag are replaced.

RepeatProcessor method process() is the method that all subclass repeaters use to perform described function. This contains the logic to find the initiated repeater metatag value and call the subclass processrepeator.

Codefragment class contains the logic to isolate all constraint metatags, where the processrepeator methods of each subclass processes these code and generates the programming code.

Repeator classes

classes are devided in 2 groups

- RepeattablesProcessor, RepeatviewsProcessor and Repeattables_viewsProcessor are an implementation of RepeatabstracttableProcessor which contains processTable method.
- all other repeaters are called in the processTable method translating all metatags that are by definition related to a table or view.

Pseudotagsprocessor.java

This class translates each metatag into a program instruction, defined by the database properties it represent and the Programlanguage settings.

Pseudotags.java

all possible metatags are defined as constants in this interface.

Expanding repeator functionality

Adding repeator classes

Extra repeater metatags are places in the Metatags interface.

With all the search and replace functionality already programmed in RepeatProcessor, it is easy to create extra repeater classes by defining a repeater metatag, select the needed constraint metatags and implement a custom processrepeator method.

These repeators need to be plugged in at RepeatabstracttableProcessor.processTable method

Adding metatags

Extra metatags require following steps

- define the metatag in the Metatags interface
- add the metatag translation programming in a logical place in the Metatagsprocessor methods
- in case a new method this must be called in 1 or more repeater classes or in an existing Metatagsprocessor method, depending on the functionality