ModelioScribes Documentation

Release 0.1

ModelioScribes

Apr 11, 2017

Contents

1 SQLScribe **1**
- 1.1 Reverse Engineering . . . 1
- 1.2 Relational Profile . . . 3
- 1.3 SQL Forward Engineering . . . 3

2 ParaScribe **5**

3 ClassScribe **7**
- 3.1 Import and export operations . . . 7
- 3.2 SCN Notation . . . 7
- 3.3 SCIN Notation . . . 10
- 3.4 User interface . . . 11

4 UseCaseScribe **13**
- 4.1 Import and export operations . . . 13
- 4.2 SUCN Notation . . . 13
- 4.3 SUCN Examples . . . 14
- 4.4 Interface . . . 15

5 ProjectScribe **17**
- 5.1 Profile . . . 17
- 5.2 Examples . . . 17

6 KMadeScribe **19**

CHAPTER 1

SQLScribe

This script allows the generation of SQL from class models and the other way around the reverse engineering of class models from a database. As shown below the reverse engineering part is based on the SchemaSpy open source tool.

Fig. 1.1: Overview of SQLScribe

Three levels of abstration are supported:

- **Data model**: at this level classes are representing persistent entities. Associations and inheritance are used to model the relationhips between these entities.
- **Relational model**: this level based on "SQL" profile desribe tables with stereotype classes. This level introduce the notion ok primary key and foreign keys are represented by dependencies between columns.
- **SQL Implementation**: this level corresponds to the SQL implementation.

Reverse Engineering

Principles

- from database to modelio "relational model"
  - based on SchemaSpy (http://schemaspy.sourceforge.net)
  - reading the xml file produced by SchemaSpy into modelio thanks to ElementTree library

Tests

Test cases

- "library" example from SchemaSpy
  - Source file directly available from http://schemaspy.sourceforge.net/sample/library.xml
  - Output that should be generated provided as an exemple in CyberBibliotheque project
- simple atomic examples
  - one example per features, e.g.
  - two empty tables,
  - two tables with columns and keys,
  - two tables with a foreign key constraints
  - ...
- both source file (database then .xml or just .xml) and modelio output should be built
- other tests cases to be build
  - open source components with database (e.g. mediawiki, joomla, fusionforge, ...)
  - 1. reverse the database with SchemaSpy to produce a .xml
  - 2. test the SQLScribeRev reverse engineering tool

The following example corresponds to a (subset) of the library example:

```
<database name="library" type="MySQL - 5.1.35-community">
  <tables>
    <table name="address" numRows="9" remarks="Address details" type="TABLE">
      <column autoUpdated="true" digits="0" id="0" name="addressId"
              nullable="false" remarks="" size="10" type="INT">
        <child column="address" foreignKey="borrower_ibfk_1" implied="false"
               onDeleteCascade="false" table="borrower"/>
        <child column="address" foreignKey="library_branch_ibfk_1" implied="false"
               onDeleteCascade="false" table="library_branch"/>
        <child column="address" foreignKey="publisher_ibfk_1" implied="false"
               onDeleteCascade="false" table="publisher"/>
      </column>
      <column autoUpdated="false" digits="0" id="1" name="address1"
              nullable="false" remarks="Address line 1" size="50" type="VARCHAR"/>
      <column autoUpdated="false" digits="0" id="2" name="address2"
              nullable="true" remarks="Address line 2 (optional)" size="50" type=
↪"VARCHAR"/>
      <column autoUpdated="false" digits="0" id="3" name="city"
              nullable="false" remarks="" size="30" type="VARCHAR"/>
      <column autoUpdated="false" digits="0" id="4" name="state"
              nullable="false" remarks="" size="2" type="CHAR"/>
      <column autoUpdated="false" digits="0" id="5" name="zip"
              nullable="false" remarks="Dash req'd for zip+4" size="10" type="VARCHAR
↪"/>
      <primaryKey column="addressId" sequenceNumberInPK="1"/>
      <index name="PRIMARY" unique="true">
        <column ascending="true" name="addressId"/>
      </index>
    </table>
    <table name="author" numRows="9" remarks="" type="TABLE">
```

```
    ...
  </tables>
</database>
```

Relational Profile

Definition of an relational profile within the local module

SQL Forward Engineering

CHAPTER 2

ParaScribe

The ParaScribe plugin is devoted to users that are not expert in UML. This tool paraphrases UML class diagrams in natural language for validation purposes or learning purposes.

Examples of paraphrases are available in some the following slides:

- DiagrammesDeClasses-ConceptsDeBase
- DiagrammesDeClasses-ConceptsAvances

ParaScribe provides some paraphases for the selected elements. It can either describe the meaning of elements or ask questions for validating the elements. In this case a form is generated allowing a non-expert can easily provide feedback. As the result the model is modified or in the case where this is not possible the model is annotated.

CHAPTER 3

ClassScribe

This script aims to simplify the input of class models in Modelio. The “import” operation transforms a text written in simple notation into class elements (classes, attributes, operations, notes, inheritance hierarchies, etc.). On the other way around the “export” operation consists in generating textual representations from an existing model.

Different notations are actually provided by ClassScribe. Each notation suits different purposes:

- Simple Class Notation (SCN): this notation is used to create/modify nested structures of classes, attributes, roles, operations and notes.
- Simple Inheritance Notation (SIN): this notation used nesting to represent hierarchies of classes or interfaces.

Import and export operations

During the import operations if an element with the same name (or the same “uuid”, see below) already exists, it is not created again. Its attributes or nested elements are modified instead. The import operation actually realizes a “merge” operation so the script can be used to modify/extend an existing model but but *NOT* to delete elements. Elements are **NEVER** deleted with this script. Deletions should be realized manually through the regular interface if this is the intention. One possibility is therefore to erase all elements and re-import the text if this is intented.

The export operation generates a file in the textual notation. During the export operation “uuid” can optionally be added to the elements. These “uuid” are unique identifiers generated by modelio when an element is created. They never change. This allows elements to be moved, renamed, etc. while keeping the same uuid. If an uuid is found during the import operation (that follow an export) then the element is identified by this uuid instead of just its name. Using uuid allows to rename model elements in a sequence export - import.

SCN Notation

Context and nesting

The elements created are always created into container element called referred as the “context”. The context defines which (nested) elements can be created in it. For instance in the context of a package, classes can be created just by

giving the list of names. In the context of a class, one can attributes, operations and roles. If only names are given it is assumed that attributes are to be created. Otherwise the syntax of the line differs in order to indicate which kind of elements is to be created. For instance the name of an operation will be followed by "()".

The "root context" corresponds to the first selected elements of modelio. Toplevel elements in the textual notation are going to be created there. Nesting in the notation then indicates the context of each element to be created. The root context can be of different types indicating the type of the elements to be created on the top level. For instance if the import operation is applied on a package then a list of names at the top level will be interpreted as a list of class, but if the import operation is applied on a class, then the list of names at the top level will be interpreted as attributes.

SCN Commands

The syntax of the SUCN notation is based on simple line commands that are nested. Each line is therefore interpreted in a given context. See the examples below for concrete illustrations of the syntax.

The general conventions are the following ones:

- Blank lines are ignored.
- Nesting is done via two spaces.
- Lines starting with "–" are comments .
- If a line is terminated with "..." and then a uuid, then this is the uuid associated with the current element.
- If a name of an element appears for the first time in the text and it does not exist in the model, then the element is created. Otherwise the element is modified. If an uuid is indicated for an element, then the search is based on this uuid instead of its name. Note that elements with uuid stay in their place and are never moved by this script.

The syntax of the SCN notation is line-based, identation based but each line can be specified as shown below. The following conventions are taken:

- <X> introduce a non terminal symbols
- "..." are terminal symbols
- (...) define groups of elements
- ...? represents an optional element
- ...* represents a repetition of zero or more elements
- ...+ represents a repetition of one or more elements
- – are comments

In the context of a **package**, classes or enumerations can be created as following:

```
<DefineClass>           ::= <Stereotype>* <Abstract>? <Name>
    -- Creates/modify the class <Name>
<DefineInheritingClass> ::= <Stereotype>* <Abstract>? <Name> "<" <Name>+
    -- Creates/modify the class <Name>, creates the syper classes if necessary and␣
↪creates
    -- generalization relationships if necessary
<DefineEnumeration>     ::= <Stereotype>* "e" <Name>
    -- Creates/modify an enumeration
```

In the context of an **enumeration**, enumeration literals can be created as following:

```
<DefineEnumerationLiteral> ::= <Stereotype>* <Name>
    -- Creates/modify an enumeration literal
```

In the context of a **class**, attributes, operations or roles can be created as following:

```
<DefineAttribute> ::= <Stereotype>* <Derived>? <Visibility>? <Name> (":" <TypeSpec>)?
    -- Creates/modify an attribute
<DefineOperation> ::= <Stereotype>* <Abstract?> <Derived>? <Visibility>? <Name> "()"
    -- Creates/modify an operation
<DefineRole>      ::= <Stereotype>* <Derived>? <Visibility>? <RoleSpec>
    -- Creates/modify a role
```

In the context of an **operation**, parameters can be defined as following:

```
<DefineParameter> ::= <Stereotype>* <Name> ":" <ParameterType>?
    -- Create/modify a parameter
```

The definitions above are based on the following elements:

```
<RoleSpec>        ::= (<RoleKind>)? <Name> ":" <Name> <Cardinality>? ("inv" <Name>
↪<Cardinality>?)?
    -- The first name is the name of the "source" role to be created or modified in␣
↪the current class.
    -- The second name is the name of the target class. The third name if it exists␣
↪is the name of
    -- the "inverse" role in this target class.
    -- TODO. This specification should be refined to explain what happen if the role␣
↪already exists.

<Composition>     ::= "<#>"
<Aggregation>     ::= "<>"

<TypeSpec>        ::= <TypeName> <Cardinality>?
    -- If not specified the cardinality is set to [1]

<Cardinality>     ::= "[*]" | "[" <Integer> ".." <Integer> "]" | "[" <Integer> ".." "*
↪" ]

<TypeName>        ::= <BasicType> | <Name>  -- an error is generated if
    -- The type name must already exist in the model. Otherwise an error is generated.

<BasicType>       ::= "integer" | "i"       -- i is a shortcut
                    | "float"   | "f"       -- f is a shortcut
                    | "boolean" | "b"       -- b is a shortcut
                    | "date"    | "d"       -- d is a shortcut
                    | "string"  | "s"       -- default if no type is defined yet for␣
↪the element

<Abstract>        ::= "a" | "abstract"

<Derived>         ::= "/"

<Visibility>      ::= "+"                   -- public
                    | "~"                   -- package
                    | "#"                   -- protected
                    | "-"                   -- private

<Name>            ::=   -- a non empty sequence of letters, digits, "_" without any␣
↪space
<Stereotype>      ::=   -- like <Name> but with enclosed in "<" and ">"
```

SCN Examples

The following example provides a complete view of the various possibilities of the "structure" notation. Note that this example is not really realistics: * most of the time only a few of the possibilities will be used * usually the same level of notation will be used for all elements, for instance specifying only the name and the visibility and the type.

Here is a first example:

```
Employee
  salary : i [0..1]
Student < organization.Person, university.Stakeholder
  firstName
  lastName : s
    S: This is a "summary" note (because it startswith #s:)
    D: Here this is a multi line description
     : with multiple line as expected. They are contactenated
     : together. So there are three lines in total.
    D: Now this another "description" note
  +middleName : string [0..1]
  birthDate:d
  <PK> nationalId
  / +age: i
  registeredCourses : Course [*] inv registeredStudents
```

The examples below illustrate the use of the notation in different contexts.

TODO: Create more examples

SCIN Notation

Sometimes it is convenient to create hierarchies of classes and interfaces

Specification (TODO):

```
Person
  Men
  Women
  Professor
  Student a
    FirstYearStudent
    MasterStudent
Serializable i
  Person
  Women
Serializable i
  Interface1 i
    AbstracClass1 a
  Interface2
    AbstractClass2 a
```

Hints:

- "i" stands for "interface"
- "a" stands for "abstract"

User interface

The user interface provides three "commands": one interactive command and two file-based commands:

- **Import class model**. This command must be launched on a selected package or class and a file name in the SCN notation must be specified. The notion of "selection" refers here to modelio selection. Elements are read form the file. The selected element is the place where new elements are going to be created.
- **Export class model**. This command generates a file for all class model elements that are (recursively) in the selected element. The output file name should be specified.
- **Edit class model**. This command open a input window for interactive usage of the SCN notation.
  - The "Clear" button clears the text in the window.
  - The "Import selection" button replaces the content of the text with the content of the selected element (this the interactive application of the import command).
  - The "Apply" button takes the content of the window and apply the changes. This is the interactive version of the export use cases.

CHAPTER 4

UseCaseScribe

This script aims to simplify the input of use case models in Modelio. The “import” operation transforms a text written in a “Simple Use Case Notation” (SUCN) into use case elements (actors, use cases, notes, etc.). On the other way around the “export” operation consists in generating a SUCN text from an existing model.

Import and export operations

During the import operation if an element with the same name (or the same “uuid”, see below) already exists, it is not created again. Its attributes or nested elements are modified instead. The import operation actually realizes a “merge” operation so the script can be used to modify/extend an existing model but but *NOT* to delete elements. Elements are **NEVER** deleted with this script. Deletions should be realized manually through the regular interface if this is the intention. One possibility is therefore to erase all elements and re-import the text if this is intented.

The export operation generates a file in the textual notation. During the export operation “uuid” can optionally be added to the elements. These “uuid” are unique identifiers generated by modelio when an element is created. They never change. This allows elements to be moved, renamed, etc. while keeping the same uuid. If an uuid is found during the import operation (that follow an export) then the element is identified by this uuid instead of just its name. Using uuid allows to rename model elements in a sequence export - import.

SUCN Notation

The syntax of the SUCN notation is based on simple line commands that are nested. Each line is therefore interpreted in a given context. See the examples below for concrete illustrations of the syntax.

The general conventions are the following ones:

- Blank lines are ignored.
- Nesting is done via two spaces.
- Lines starting with “–” are comments.
- If a line is terminated with ”...” and then a uuid, then this is the uuid associated with the current element.

- If a name of an element appears for the first time in the text and it does not exist in the model, then the element is created. Otherwise the element is modified. If an uuid is indicated for an element, then the search is based on this uuid instead of its name. Note that elements with uuid stay in their place and are never moved by this script.

The notation is based on nesting of simple line commands (a name is specified for each command below for futher reference in specification or implementation artefacts). The accepted commands are the following:

- At the ‘‘top-level**:
  - `NameA` (*DefineActor*) : Create/modify an actor.
  - `NameA \< NameB` (*DefineActorInheritance*) : Create/modify an actor NameA inheriting from an actor NameB. Actors are created or reused.
  - `NameA1 NameA2 ... NameAn - NameCU` (*DefineUseCase*): Create/modify a use case and associate it with the corresponding actors. Actors are created or reused. Association between actors and use cases are merged with previous ones if already existing.
- In the context of **use cases**:
  - `NameC` (*DefineCollaboration*): Create/modify a collaboration named NameC
  - `t-> NameC` (*DefineTraceability*): Create a traceability (hence “t->”) dependency towards the model element NameC if it exist. The element is first search in the “analyst project”, and then in the “uml project”.If this model element does not exist or various element with the same name exists, then a warning is issued but the process is not stopped.
- In the context of **actors**, **use cases** or **collaborations**:
  - `#s SomeText` (*DefineSummary*): Create/modify the “summary” note. If an uuid is specified, then the note is identified and its content is changed with the text provided. Otherwise as it is important to avoid creating multiple descriptions each time the model is imported, it is assumed that the definition to be modified is the first one if any and in this case its content is replaced by the text provided.
  - `#d" SomeText` (*DefineDescription*): Create/modify the “description” note. The behavior is analogous to the one described for summary notes.
  - `" SomeText` (*DefineAdditionalText*): Append a text to the previous note. Note that a space should follow the “” character.

SUCN Examples

Here is an example of SUCN text that can be used to create a blank model:

```
-- this is an example of comment
Actor5
Actor1
  S: This is the "summary" note associated with Actor1.
  D: And now a "description" note. Actors does not have
   : to be declared unless notes have to be attached with them
   : or they have inheritance relationships. Otherwise
   : they are declared "online" when found
Actor8
Actor4
  S: The summary of Actor4

Actor6 - UseCase3
Actor6 - UseCase4
-- since various actors can perform a usecase, defining list of actors is usefull
Actor1 Actor2 - UseCase2
```

```
-- alternatively one can use to separate definition like below
Actor1 - UseCase2
Actor2 - UseCase2

Actor1 - UseCase1
  S: This is the "summary" of UseCase1 because "s" stands for summary.
  D: This is the "description" note attached to UseCase1.
   : It has two lines as this is the continuation of the previous note.
  D: This is another description starting here.
  -- here comes a definition of a traceability link towards an existing element
  -- if the element does not exist a warning is issued
  t-> AnNamedElementOfWhateverType
  Collaboration11
    S: This is the "summary" of the Collaboration11

  Collaboration12
    S: This is the "summary" of Collaboration12
    D: This is the "description"
     : and it continues on
     : multiple lines.
  Collaboration13
```

Interface

The user interface provides three "commands": one interactive command and file-based commands: * **Import use case model**. This command must be launched on a selected package and a file name in the SUCN notation must be specified. The notion of "selection" refers here to modelio selection. Elements are read form the file. The selected package will be the place where new elements are going to be created. Existing elements will stay in their place as explained below. * **Export use case model**. This command generates a file for all actors, use cases and collaborations that are (recursively) in the selected package or if no package is selected that are in the UML project. The output file name should be specified. * **Edit use case model**. This command open a input window for interactive usage of the SUCN notation.

- The "Clear" button clears the text in the window.
- The "Import selection" button replaces the content of the text with the content of the selected element (this the interactive application of the import command).
- The "Apply" button takes the content of the window and apply the changes. This is the interactive version of the export use cases.

Note: It is assumed that actors and use cases names are global (this hypothesis is valid for this use case of model). So if existing actors or use cases have already the same names as the ones found in the model (or the same uuid if any) during an import, then these entities will be modified in the place wherever they are. This avoid package qualification which could be quite cumbersome in practice in use case models. It is still possible to create sophisticated structures if needed by first importing use cases and actors in a blank model, then creating various packages and moving elements in these packages. When the elements will be imported again, they will stay in the same package (except newly created elements that will go into the "selected" package).

CHAPTER 5

ProjectScribe

This plugin allows to import a GanttProject file into modelio. Tasks and HumanRessources are made available in the model and can therefore be referenced in modelio.

Profile

The integration of GanttProject entities into modelio implies defining a “Project” profile.

TODO

Define here the list of stereotypes and tag values.

Examples

The content of GanttProject files looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="" company="" webLink="" view-date="2012-12-01"
     ... >
    <description/>
    ...
    <tasks empty-milestones="true">
        ...
      <task id="0" name="Architectural design" color="#99ccff"
            meeting="false" start="2012-12-24" duration="22" complete="78" expand=
↪"true">
        <task id="9" name="Create draft of architecture" color="#99ccff"
              meeting="false" start="2012-12-24" duration="10" complete="100" expand=
↪"true">
          <depend id="10" type="2" difference="0" hardness="Strong"/>
```

```
          <depend id="12" type="2" difference="0" hardness="Strong"/>
        </task>
        <task id="10" name="Prepare construction documents" color="#99ccff"
              meeting="false" start="2013-01-07" duration="12" complete="60" expand=
↪"true">
          <depend id="17" type="2" difference="0" hardness="Strong"/>
        </task>
        ...
      </task>
      <task id="11" name="Interior design" color="#99ccff"
            meeting="false" start="2013-01-07" duration="10" complete="33" expand=
↪"true">
        ...
      </task>
      ...
    </tasks>
    <resources>
      <resource id="1" name="Jack House" function="Default:1"
                contacts="jack.house@myselfllc.net" phone="0044 077345456"/>
      <resource id="0" name="John Black" function="4"
                contacts="john.black@myselfllc.net" phone="+44 0794353567"/>
      ...
    </resources>
    <allocations>
      <allocation task-id="9" resource-id="1" function="Default:1"
                  responsible="false" load="50.0"/>
      ...
      <allocation task-id="1" resource-id="0" function="4"
                  responsible="false" load="100.0"/>
      ...
    </allocations>
    <vacations>
      <vacation start="2009-02-02" end="2009-02-09" resourceid="1"/>
    </vacations>
    ...
    <roles roleset-name="Default"/>
    <roles>
        <role id="0" name="Architect"/>
        <role id="1" name="Bricklayer"/>
        <role id="2" name="Foreman"/>
        ...
    </roles>
</project>
```

CHAPTER 6

KMadeScribe

This plugin makes a bridge between modelio and KMade