tiOPFMapper

Description

tiOPF [http://tiopf.sourceforge.net] is a wonderful framework, but the tasks of writing all of the boiler plate code can be a lot of work. So I wrote the tiMapper utility in the spirit of some existing PHP frameworks that use YML or XML to describle a project's classes and other types which are then transformed into the base class files with all of hard coded boiler plate code already written.

The tiMapper utility uses one or more xml documents to describe a "schema". The schema describes any TtiObject based classes and enumerations as well as mappings that use the tiAutoMap registration mechanism to store meta data about the types described in the schema. There is one main "schema" file which can have INCLUDES pointing to other xml files which describe yet additional classes, enums and mappings if necessary.

Eventually, I'll write a GUI utility front end for it, but for now I am content writing out the xml. A typical 200 line xml schema will produce about 2K lines or more of pascal code for defining the classes, their visitors, registering visitors, mappings, etc and gluing everything up so that you're bascially able to just start writing code.

Inhaltsverzeichnis

Basics

The utility uses the schema to create unit (.pas) files and within those unit files, each class, enumeration mappings, etc described in the schema.

Project schema definition

Sample

Take the following project schema definition:

```
<project
   tab-spaces="2"
   begin-end-tabs="1"
   visibility-tabs="0"
   project-name="MyProject"
   enum-type="int"
   outputdir="../bom"
  >
  <!-- Includes are added to this schema before build-time. -->
   <includes>
      <item file-name="/jobs_schema.xml"/>
   </includes>

   <!-- Units (pas) files that will be created along with defined types, classes, etc. -->
   <project-units>
      <unit name="person_bom">

         <!-- Enumerations defined here -->
         <enums>
            <enum name="TGenderType">
               <values>
                  <item name="gtFemale" value="0"/>
                  <item name="gtMale" value="1"/>
               </values>
            </enum>
         </enums>

         <!-- Classes defined here -->
         <classes>
         </classes>
      </unit>
   </project-units>
</project>
```

<project>

The root <project> node should include the bare parameters required. There are:

tab-spaces, etc:

These control formatting of the code in the pascal units produced by the utility.

enum-type:

The mapping framework takes into consideration whether you are storing
your enumerated types as either the String representation (ala "bsNone") or the
integer representation such as 0. Valid values are "string" or "int".

outputdir:

Determines the directory where the mapping utility should dump the unit files
which it creates. If the outputdir parameter is omitted, the output directory defaults to the
same directory in which the schema file itself resides.

Project units

The <project> contains one node called <project-units> which contains one or more <unit> nodes. These <unit> nodes contain mainly 2 nodes, <enums> and <classes>. A more detailed example of a single <class> node is shown below.

Sample

```
<class
   base-class="TPerson"
   base-class-parent="TtiObject"
   auto-map="true"
   auto-create-list="true">
   <!-- Class properties -->
   <class-props>
      <prop name="FirstName" type="string"/>
      <prop name="LastName" type="string"/>
      <prop name="Age" type="Integer"/>
      <prop name="Gender" type="TGenderType"/>
      <prop name="IsActive" type="Boolean"/>
      <prop name="ActiveDate" type="TDateTime"/>
      <prop name="Email"/>
   </class-props>
   <validators>
      <item prop="Age" type="greater-equal">
         <value>18</value>
      </item>
      <item prop="FirstName" type="required"/>
      <item prop="LastName" type="required"/>
   </validators>
   <!-- Mapping into the tiOPF framework -->
   <mapping table="person" pk="OID" pk-field="OID" oid-type="string">
      <prop-map prop="FirstName" field="first_name" type="string"/>
      <prop-map prop="LastName" field="last_name" type="string"/>
      <prop-map prop="Age" field="age"  type="integer"/>
      <prop-map prop="Gender" field="gender" type="enum"/>
   </mapping>
   <selections>
    <select type="func" name="FindByGender">
       <params>
          <item name="AGender" type="enum" type-name="TGenderType"
            pass-by="const" sql-param="gender_type"/>
       </params>
       <sql>
          <![CDATA[
             SELECT
               ${field_list}
             FROM
                PERSON
             WHERE
                PERSON.GENDER = :gender_type
          ]]>
```

```
        </sql>
      </select>
    </selections>
</class>
```

Resulting class definition:

```
{ Generated Class: TPerson}
 TPerson = class(TtiObject)
 protected
   // getters/setters here
 public
   procedure   Read; override;
   procedure   Save; override;
   function    IsValid(const AErrors: TtiObjectErrors): boolean; overload; override;
 published
   property    PersonType: TPersonType read FPersonType write SetPersonType;
   property    FirstName: string read FFirstName write SetFirstName;
   property    LastName: string read FLastName write SetLastName;
   property    Age: Integer read FAge write SetAge;
   property    Gender: TGenderType read FGender write SetGender;
   property    IsActive: Boolean read FIsActive write SetIsActive;
   property    ActiveDate: TDateTime read FActiveDate write SetActiveDate;
   property    Email: string read FEmail write SetEmail;
 end;
```

<class-props>

These describe the properties of a class. Default is "String" so omitting the "type" attribute will register the property as String. All base type (string, boolean, TDateTime) as well as enumerated types. The types that get registered are written directly out to the pascal class definition.

<validators>

Basic validators are supported including "required" (string types only) which ensures that the there is not an empty string. There is also "greater"/"greater-equal", "less"/"less-equal" and "not-equal". For all Validator types except "required", you must include a <value> node inside the <validator> node indicating the value to compare against.

When the class code is written out to the unit file, the mapper creates an IsValid() override for each class that has <validator>'s defined and writes out the code for is. For the TPerson describted above, the following would be generated:

```
function TPerson.IsValid(const AErrors: TtiObjectErrors): boolean;
var
  lMsg: string;
begin
  Result := inherited IsValid(AErrors);
  if not result then exit;

  if Age < 18 then
    begin
      lMsg := ValidatorStringClass.CreateGreaterOrEqualValidatorMsg(self, 'Age', Age);
      AErrors.AddError(lMsg);
    end;

  if FirstName = '' then
    begin
      lMsg := ValidatorStringClass.CreateRequiredValidatorMsg(self, 'FirstName');
      AErrors.AddError(lMsg);
    end;

  if LastName = '' then
    begin
      lMsg := ValidatorStringClass.CreateRequiredValidatorMsg(self, 'LastName');
      AErrors.AddError(lMsg);
    end;

  result := AErrors.Count = 0;
end;
```

The ValidatorStringClass is a class which you can set that will take care of supplying strings to the framework for errors, etc so that you can use your existing model for language translations, etc.

<mapping>:

The <mapping> tag allows you to setup the tiOPF related ORM mappings between your class and the tiOPF framework. The same base types are supported such as "string", "boolean", etc and should match what you indicated in the <prop> tag to define the class property that you are mapping. The EXCEPTION is enumerated types. If the type being mapped is an enumerated type, the value that you indicated should be "enum". This will allow the mapping utility to account for how to write out functions for dealing enumerated types such as when overriding a visitor's MapRowToObject method. When the type is "enum", the mapper will lookup the type indicated for the property by it's corresponding <prop> tag and use that.

For each class that you define with a populated <mapping> tag, the mapper will automatically create
hard coded CRUD visitors including Read, Update, Delete, Create and write out their code as well as register them with gTIOPFManager.

<selections> and Automatically generated lists:

The mapper can automatically create TtiFilteredObjectList descendants for each class that you define by setting the "auto-create-list" attribute of the <class> node to "true". The mapper will then create a TYourClassNameList object for the class and give it a FindByOID. The mapper will also created hard coded CRUD visitors for each List its creates for a class that you define as well as register those visitors.

A quick note on Auto-Mapping...

If you are using the auto map feature of tiOPF, the mapper will write code to make calls to register your class's properties. Additionally, the mapping utility uses the mappings that are registered to create a visitor that uses the mappings to perform queries using the familiar automap/criteria interface such as myList.Criteria.AddEqualTo() which the hard coded visitor uses to flesh out the WHERE and ORDER BY clauses of the SQL statement.

OK, back to <selections>. <select> tags allow you to define methods that get written into the TtiFilteredObjectList class that gets created when the <project> tag's "auto-create-list" is set to "true".

```
<selections>
  <select type="func" name="FindByFirstName">
    <params>
      <item name="AName" type="string" pass-by="const" sql-param="user_first"/>
    </params>
    <sql>
      <![CDATA[
        SELECT
          ${field_list}
        FROM
          PERSON
        WHERE
          PERSON.FIRST_NAME = :USER_FIRST
```

```
        ]]>
      </sql>
    </select>
</selections>
```

The <select> above would result in the following list class:

```
  TPersonList = class(TtiMappedFilteredObjectList)
  public
    // typical overriden methods for list here

    { Returns Number of objects retrieved. }
    function    FindByFirstName(const AName: string): integer;
  end;
```

It's FindByFirstName implementation would be written out as:

```
  function TPersonList.FindByFirstName(const AName: string): integer;
  begin
    if self.Count > 0 then
      self.Clear;

    Params.Clear;
    AddParam('AName', 'user_first', ptString, AName);
    self.SQL :=
      ' SELECT PERSON.OID , PERSON.FIRST_NAME, PERSON.LAST_NAME,  ' +
      ' PERSON.AGE, PERSON.GENDER FROM PERSON WHERE PERSON.FIRST_NAME  ' +
      ' = :USER_FIRST';
    GTIOPFManager.VisitorManager.Execute('TPersonList_FindByFirstNameVis', self);
    result := self.Count;
  end;
```

Notice that the class definition for TPersonList has a built in method called FindByFirstName which takes a const AName: string param and returns and integer indicating the number of objects that got populated into the object list. All of the sql and code to use the sql is written automatically.

The utility also creates a specialized visitor and fleshes it out.

```
  TPersonList_FindByFirstNameVis = class(TtiMapParameterListReadVisitor)
  protected
    function    AcceptVisitor: Boolean; override;
    procedure   MapRowToObject; override;
  end;
```