DDL generator

Design Document

Version: 1.0

Version: 0.9
Date: May 29, 2006
Author: Le Duc Bao
Email: leducbao@gmail.com
Mentor: Ralf Joachim

Table of Contents

Introduction

Purpose

This document describes the architecture of the DDL generator tool.

Scope

This use case document applies to the project DDL generator in sense of Google Summer of Code 2006.

Environment

Castor DDL generator requires:

- Java 1.4+
- Castor 1.0

High-level architecture

The generator takes a mapping file and 2 configuration files (optional) from commandline.

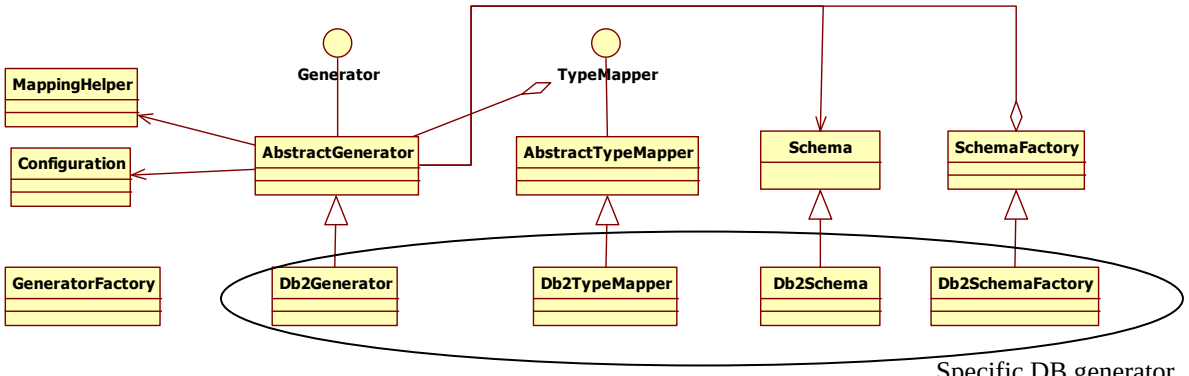

Figure 1: High-level class diagram of DDL Generator

The program is divided into several different packages:

- `org.castor.ddl`: contains skeleton classes for the generator including AbstractGenerator, AbstractTypeMapper, Configuration, GeneratorFactory, MappingHelper…
- `org.castor.ddl.schemaobject`: contains classes dedicated to manage schema element like Table, Field… and to generate DDL
- `org.castor.ddl.typeinfo`: contains several type's categories
- `org.castor.ddl.[database]`: contains implementation classes for specific database
- `org.castor.ddl.[database].schemaobject`: contains implementation classes of schema object classes for a specific database.

Detail design

Generator

AbstractGenerator is the base class for various DDL generator of specific DB and handles following tasks:

- Extract essential information from Mapping to Schema.
- Loop through the schema and provide a skeleton for DDL creation

AbstractGenerator will automatically extract necessary information for DDL creation. That information is handled by Schema. There are some highlights for schema extraction:

- The KeyGenDef defines the key-generator and shares among tables. So, we must first use list to store the KeyGenDef. During fetching tables, the key-generator will be added to table in which it requires.

- The m-n relationship creates an additional table (resolving table). It will be added to a list _resolveTable and manipulated after going through the mapping. A resolving table has its parents' identities, foreign keys.

```
   <field name="products" type="myapp.Product" required="true"
collection="vector">
         <sql name="prod_id" many-table="category_prod" many-
key="category_id" />
   </field>
```

- The 1-1, 1-n relationships create the foreign key. It is added into table which handles the key. In this case, the type is not a JDO's supported type:

```
<field name="product" type="myapp.Product">
      <sql name="prod_id" />
</field>
```

It should resolve the field's type by referring the type of the identity of the class which is defined in the field's type.

```
<class name="myapp.Product" identity="id">
      <map-to table="prod" xml="product" />
      <field name="id" type="integer">
            <sql name="id" type="integer" />
      </field>
</class>
```

In this example, the field `"product"` has type of `"integer"` (by referring to class `"myapp.Product"`, its identity is `"id"`, and type of `"id"` is `"integer"`).

The field that is defined as an identity (`<field name="id" type="integer" identity="true">`) is used as the primary key for a table. In the case the identity is defined in the class (`<class name="Product" identity="id">`), the field associated with the identity's value is the primary key. The identity

definitions at class and field are alternative syntax. If both are specified the one at field should take precedence over the class one.

- There is no index definition, no unique key definition.

To create new generator for a DBMS, you should:

- Overwrite this class to create new generator for a DBMS.
- If the syntax of DBMS is different to standard DDL syntax, you should overwrite SchemaObject (Table, Field, KeyGenerator, Index, ForeignKey,...) classes. The class SchemaObjectFactory who handles the SchemaObject creation must be also overwritten.
- You must overwrite the TypeMapper if mapping between JDBC types and specific DBMS's types is different among various DBMS.

The example bellow shows how to create a generator for DB2:

- **Generator for DB2**

```
public class Db2Generator extends AbstractGenerator {

    // constructor
    public Db2Generator(final String globConf, final String dbConf)
            throws GeneratorException {
        super(globConf, dbConf);

         // type mapper implementation
        setTypeMapper(new Db2TypeMapper(getConf()));
    }
}
```

- **TypeMapper for DB2**

```
public final class Db2TypeMapper extends AbstractTypeMapper {

    // constructor
    public Db2TypeMapper(final Configuration conf) {
        super(conf);
    }

    // define type mapping
    protected void initialize(final Configuration conf) {
        // numeric types
        this.add(new NotSupportedType("bit"));
        this.add(new NoParamType("tinyint", "SMALLINT"));
        this.add(new NoParamType("smallint", "SMALLINT"));
        this.add(new NoParamType("integer", "INTEGER"));
        this.add(new NoParamType("bigint", "BIGINT"));
    }
}
```

- **Field for DB2**

```
public class Db2Field extends Field {

    // constructor
    public Db2Field() {
```

```
        super();
    }

    // generate ddl for field
    public String toDDL() throws GeneratorException {
        StringBuffer buff = new StringBuffer();
        buff.append(getName()).append(" ");
        buff.append(getType().toDDL(this));

        if (isIdentity()) {
            buff.append(" NOT NULL");
        }

        // process key generator
        KeyGenerator keyGen = getKeyGenerator();
        if (keyGen != null && isIdentity()) {

            if (KeyGenerator.IDENTITY_KEY.equalsIgnoreCase(keyGen.getNa
me())) {
                buff.append(" GENERATED BY DEFAULT AS IDENTITY ").
                    append("START WITH 1 INCREMENT BY 1");
            }
        }

        return buff.toString();
    }

}
```

- **SchemaFactory for DB2**

```
public class Db2SchemaFactory extends SchemaFactory {

    // constructor
    public Db2SchemaFactory() {
        super();
    }

    // create field
    public Field createField() {
        return new Db2Field();
    }

}
```

The GeneratorFactory class handles the specific database generator creation. For example:

```
Generator generator = GeneratorFactory.
     createDDLGenerator("mysql", "ddl.properties", "mysql.properties");
```

SchemaObject & SchemaFactory

SchemaObject is a set of classes which encapsulate the schema elements, such as Schema, Table, Field, Index, Key Generator, and Foreign Key and generate DDL for

themselves. The figure 2 shows the class diagram of schema objects and schema factory.

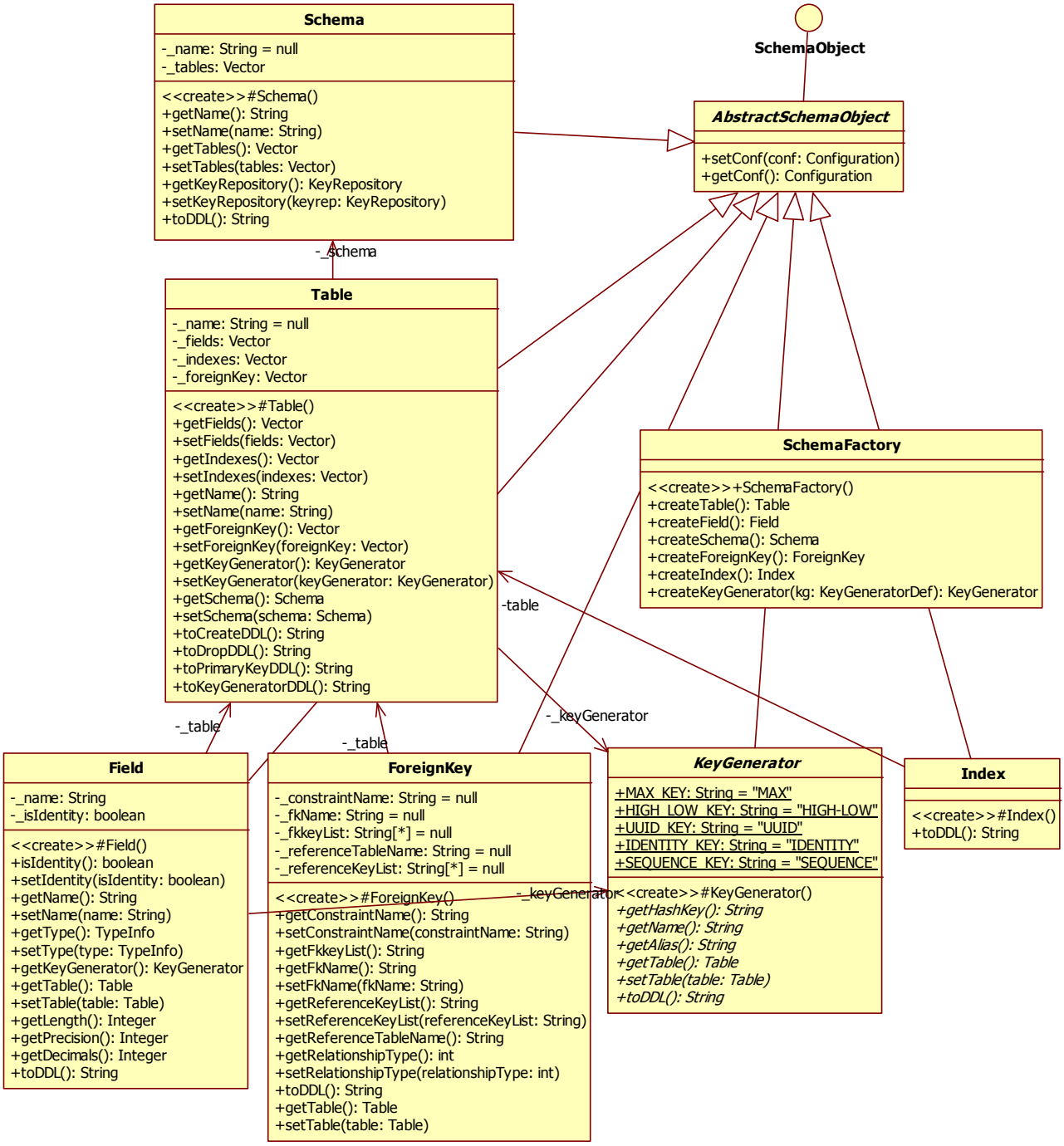

Figure 2: The schema object's class diagram and the schema factory

The SchemaFactory class handles the creation for various schema objects. It helps the AbstractGenerator to dynamically extract schema's information for specific database.

SchemaObject handles necessary information for DDL creation which is extracted from Mapping. This extraction makes the generator be independent with Mapping. In most of SchemaObject, the method toDDL() is used to generate default DDL for most of databases. We must overwrite SchemaObject and SchemaFactory if its format is different with the default.

Castor supports 5 key generators including SequenceKey, HighLowKey, IdentityKey, MaxKey, and UUIDKey and using the KeyGenerator class as the base class. The HighLowKey, MaxKey, and UUIDKey do not generate DDL script.

- The IdentityKey is used to create the "auto-increment" style column for a table and is supported by MySQL, PostgreSQL, Derby, Microsoft SQL Server, DB2, Sybase, HSQL, and PointBase.
- The SequenceKey is used to define the sequence for the identity of a table. It is supported by Oracle, PostgreSQL, SapDB, Db2, and HSQL. The SequenceKey creates the DDL for SEQUENCE. If the trigger is true in the KeyGenDef element, the trigger DDL will be generated. The generator uses the trigger's template (trigger_template) which is defined in the [database].properties file. The example below shows a template for Oracle:

```
trigger_template=CREATE TRIGGER <trigger_name>\n\t\
BEFORE INSERT OR UPDATE ON <table_name>\n\t\
FOR EACH ROW\n\t\
DECLARE\n\t\t\
   iCounter <table_name>.<pk_name>%TYPE;\n\t\t\
   cannot_change_counter EXCEPTION;\n\t\
BEGIN\n\t\t\
    IF INSERTING THEN\n\t\t\t\t\
        Select <sequence_name>.NEXTVAL INTO iCounter FROM
Dual;\n\t\t\t\t\
        :new.<pk_name> := iCounter;\n\t\t\
    END IF;\n\t\
\n\t\t\
    IF UPDATING THEN\n\t\t\t\t\
         IF NOT (:new.<pk_name> = :old.<pk_name>) THEN\n\t\t\t\t\t\
             RAISE cannot_change_counter;\n\t\t\t\t\
         END IF;\n\t\t\
    END IF;\n\t\
\n\t\
EXCEPTION\n\t\t\
     WHEN cannot_change_counter THEN\n\t\t\t\t\
         raise_application_error(-20000, 'Cannot Change Counter
Value');\n\t\
END;
```

There are 5 pre-defined keywords being used in a template:

- <sequence_name>: sequence's name, refer to http://castor.codehaus.org/key-generator.html#SEQUENCE-key-generator for more information.

- <trigger_name>: trigger's name, auto-generated by using <sequence_name>.
- <table_name>: table's name of the trigger.
- <pk_name>: primary key.
- <pk_type>: primary key SQL's type.

TypeMapper

TypeMapper is the interface to define the mapping between JDBC's types and SQL's types. Each database has its own data types. We must implement TypeMapper for each database.

The AbstractTypeMapper id the base class for specific database data type mapping. It uses a Map to handle a list of TypeInfo which defines the association of JDBC's type and SQL's type. There are several different TypeInfo types: RequireLengthType, OptionalLengthType, RequiredPrecisionType. The Configuration is used to define the default value for TypeInfo. For example,

```
# define the property's key for default precision of bitint
default_bigint_precision=19
```

For a specific database, it must create a class inherited the AbstractTypeMapper and overwrite the *initialize()* method.

MappingHelper

MappingHelper is a utility class which provides essential methods for manipulating Mapping document.

Configuration

Configuration class manages the configuration for Generator. The configuration is formatted as property file and divided into 2 file:

- ddl.properties: for all databases.
- [database].properties: for specific database.

Users can edit those files.

Conclusion

This paper has described the DDL generator architecture. It is not a guide to running the program. For information on running the DDL generator, please refer to the User Guide.

Reference

- Castor project: http://castor.codehaus.org/index.html
- JDO mapping/schema: http://castor.codehaus.org/jdo.html
  http://castor.codehaus.org/schema.html