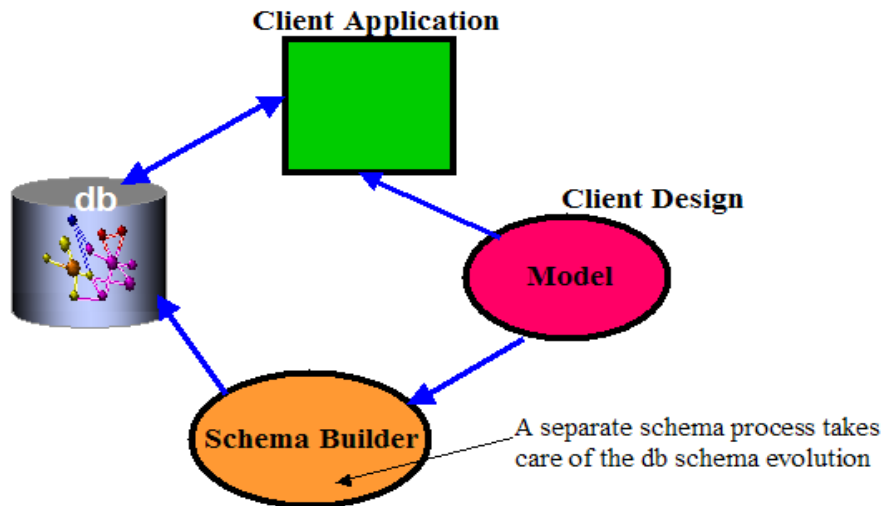


# Schema Builder

By Richard Lingeh: Principal Consultant Versant

## Definition

This pattern allows any further evolution of a database schema after a complex modification of the client application model to be done by a separate schema process and not by the client application or by the schema it generates. This separate process generates a database schema that matches the complex evolving client application model. The complex evolving application model comes from changes in the inheritance structure of the application. This means the remodelling of an application by adding a super class and moving previous attributes in previous defined classes to this super class. It could also mean the insertion of an intermediate class between a super class and its sub classes. The database schema generated by the Schema Builder pattern should match the new client application schema without loss of data.



This pattern is similar to the *GoF Builder* pattern in the creational pattern.

## Motivation

Many business applications undergo continuous and constant modifications. Most of the changes occur because of customers' demands. In the software these changes in the simplest form can be the addition of new data members (attributes) to existing classes, addition of new classes, deleting existing class members, deleting existing classes. In this situation the database schema is automatically evolved and use - the so call *lazy schema evolution*.

Another situation is when the inheritance structure of an application changes. This involves adding a new super class, moving attributes from existing classes to this super class and making these existing classes now sub classes of the newly created super class. As another example new classes are inserted between an existing super class and its sub classes (intermediate classes). Doing these on the client application side involves code modifications. The already deployed application has an underlying database with thousands or millions of objects based on the old schema. The business unit doesn't expect any loss of data in the database after installing the new software

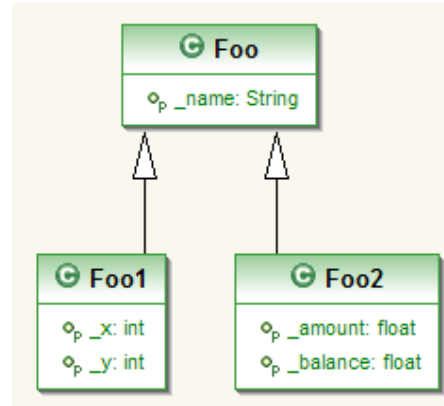
version. Versant Object Database has to offers a way to take care of this to make sure that there is no data loss after the installation of the new software release. With the APIs that Versant offers, the Schema Builder pattern can be created. Two examples of complex modifications in an application model are shown below.

**Scenario 1**

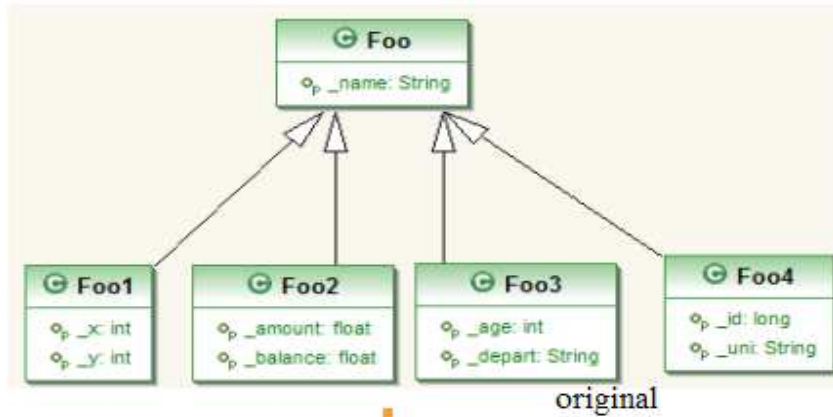
*Original application*



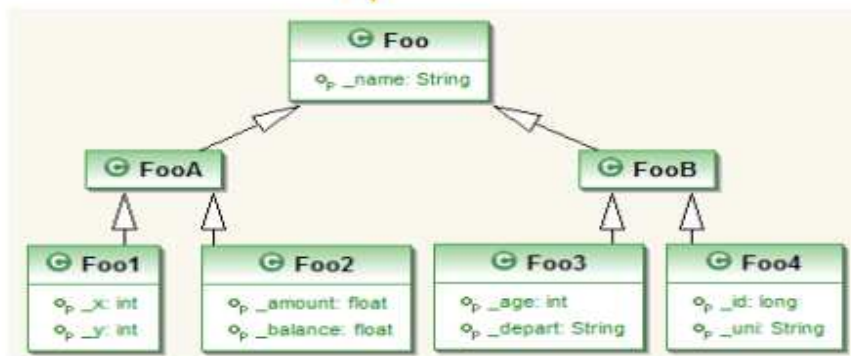
*Modified application*



**Scenario 2**



original



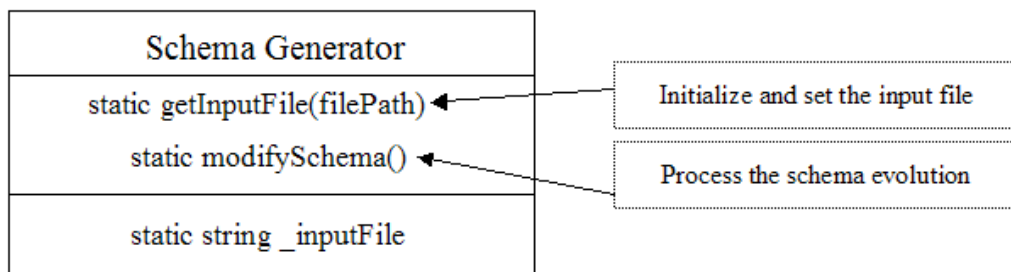
modified

**Applicability**

Use the Schema Builder pattern when

- The application model changes in such a way that the entire inheritance structure changes through the addition of super classes and will involve the movement of attributes from the sub classes to the newly added super classes.
- The application is expected to be flexible and extensible and its model changes in a way that the entire inheritance structure changes through the addition of intermediate classes between existing super classes and their sub classes.
- There is possibility to release a new version of the application code to accommodate inheritance structure changes
- OSS/BSS applications that are based on the TMF/SID framework. These applications are of independent separate sub applications that have some classes in common and are expected to be extensible. Versant Object Database schema builder pattern creates a general database schema and it is possible for any application to use the sub set of this database schema.

### Structure



### Participants

- Schema Generator
  - Has a *getInputFile()* method which takes as argument a path to an input file. It checks if this file exists and then uses it to set the member *\_inputFile*
  - Provides a schema evolution method *modifySchema()*. This method uses the input file *\_inputFile* to create a new super class (with or without attributes) if it doesn't exist. It creates an inheritance relationship between the classes (super classes and sub classes) and the attributes.
  - Has a data member *\_input file*. This file can be of any format (mostly ascii or xml file). If it's an ascii file, each line can have for example 3 entries: super class, subclass, attribute

Supcls1	subcls1	attribute1
Supcls1	subcls2	attribute1
Supcls2	subcls3	attribute2

If it's an xml file each section has 3 entries: super class, subclass, attribute

```
<Section>
  <supclass> supcls1 </supclass>
  <subclass> subcls1 </subclass>
  <attribute> attribute1 </attribute>
</section>
```

Attribute types can also be written into this file.

## ***Collaborations***

The process for the evolution of the database schema depends on the input file `_inputFile`. This file is set by a some sort of an initialization function `getInputfile()`. The method `modifySchema()` now uses this file to process the schema evolution in the database.

## ***Consequences***

The consequences of the Schema Generator pattern are as follows:

1. *Complex schema evolution with no data loss.* Schema evolution in case of changes in the inheritance structure of the client application is possible without loss of data. One of the most complex modifications in any existing application model is to change its inheritance structure. If this happens and the schema generated by the client application is used to evolve the database schema, there will be loss of data. With the Schema Builder pattern, there is a guarantee that no data will be lost.
2. *Facilitate complex schema evolution process.* With this pattern complex schema evolution can be resolved in minutes. Other options are to stream (dump) all the data from a database into a binary or xml files, capture the new schema in the database and then stream the data from the file back into the database. This process can take hours or days depending on the size of the database. This time window is too long and most companies with not leave their production databases offline for that long. With this pattern most of the critical problems facing business units are resolved.
3. *Facilitate the notion of extensibility in today's applications.* Applications are expected to have good performance, flexible and easy to extend. This pattern makes it possible to extend any Versant supported application to any degree.
4. *Facilitate distributed software development.* In a distributed software development where the module are independently developed and expected to work on the same database, the decoupling of the database schema from being evolved by the client application code facilitates development. Since the schema evolution is now done by own module, each of the client application module concentrate only on their own portion of software which will use a subset of the db schema (C++).

## ***Sample code***

An application made of 2 classes Foo1 and Foo2 is redesigned to contain a super class Foo. A common attribute `_name` defined in Foo1 and Foo2 is now moved to Foo. The application programmer takes care of modifying the code and the possibilities offered by Versant make it easy to use the Schema Builder pattern here to modify the

database schema without loss of existing data. Typical changes in the application code are shown below.

```
Class Foo1                                Class Foo2
{                                           {
    string _name;                          string _name;
    int _x;                                 double _p;
}
```

*Before remodelling*

```
Class Foo                                Class Foo1 : Foo                Class Foo2 : Foo
{                                           {                                           {
    string _name;                          int _x;                                       double _p;
}
```

*After modification*

Now to modify the database schema to match the new application schema using the Schema Builder pattern code below, simply call its *modifySchema()* method with the following parameters:

```
SchemaBuilder::modifySchema("Foo", "Foo1", "_name", "String");
```

```
SchemaBuilder::modifySchema("Foo", "Foo2", "_name", "String");
```

An input file can instead be used especially when the number of classes to be created is large. In the above case such a file (ascii ) will look thus:

```
Foo    Foo1    _name    string
Foo    Foo2    _name    string
```

*Typical code snippet for the Schema Builder*

```
Class SchemaBuilder
{
    private o_dbname db;
    static o_err defineSupCls(char *clName, char* attr,
    char* attrType)
    {
        o_err err = O_OK;
        o_vstr vstr;
        o_object classObj;
        //defined further attributes to be moved from
        the subclasses here
        int refactor = 0;
        //get refactor from the subclass and use here
        o_attrdesc attrDescAtr = {attrName, {"", ""},
        attrType, refactor, NULL, 0, NULL};

        o_attrdesc *attrDescP[] = {&attrDescAtr};
        vstr = o_newvstr(&vstr, sizeof(attrDescP),
        (o_ulb *)attrDescP);
        if (vstr == NULL)
            return o_errno;
    }
}
```

```

classObj = o_defineclass((o_clsname)className,
db, NULL, vstr, NULL);
if (classObj == NULL)
    return o_errno;
/* 1. Locate the class object */
o_clsobj *clsObj = (o_clsobj
*)o_locateobj(classObj, RLOCK);
/* 2. Get the attribute that we just defined */
o_attrobj *attr = (o_attrobj *)o_locateobj
(((o_object *)clsObj->attrs))[0], WLOCK);
/* 3. Make the initval vstr empty */
o_adjustvstr(&(attr->initval), 0);
/* 4. Make the object dirty */
o_setdirty(((o_object *)clsObj->attrs))[0]);

return err;
}

void modifySchema(char *supCls, char* subCls, char*
attr, char* attrType)
{
    o_object supObj, subObj;
    o_clsobj *super = NULL, *sub = NULL;
    o_err err = O_OK;
    supObj = o_locateclass(supCls, db);
    if (supObj == NULL) {
        err = defineSupCls(superCls, attr,
attrType);
        if (err != O_OK)
            exit (4);
        superObj = o_locateclass(supCls, db);
        if (superObj == NULL)
            exit (4);
    }
    //check and define the superclass attr
    super = (o_clsobj *)o_locateobj(supObj, RLOCK);
    //Now get the superclass attribute
    int supAttrIdToDirty = 0;
    //1. Use versant api to check the super class
    attr to dirty
    //2. modify sub and super class attribute
    properties
    //3. modify the subclasses and super classes
    properties
    //4. rename attributes
    //5. change attribute types
}
}

```

There is a complete code for this pattern in C++ and Java for the Versant ODBMS.

### ***Example***

Ericsson's OSS/BSS application running on Versant used this pattern to modify their database schema after inserting intermediate classes in their existing inheritance structure. The reason for remodelling the application was to improve performance.

INITplan, a German logistic company uses this pattern in remodelling their application. For their new software version they have to insert new super classes, intermediate classes, move and rename attributes to make the application stable, easy to extend and more flexible.