CLIENT: CEA **PROJECT:** PAL/SALOME

PROJECT ID: 4 0001 26 209 **TASK:**

PROJECT

PAL/SALOME

LIVRABLE

DELIVERABLE

TYPE

PERSISTENT FORMAT SPECIFICATION

STUDY

Dump Python Extension

Historical				
Ver.Indice	**Date**	**Author**	**Short description**	**Status***
0.1	20/12/05	SRN	Initial document creation	D
0.2	20/06/07	JFA	Update for Salome 3.2.6: minor syntax correction	D

* **D**: Draft, **P**: Proposed for validation.

Customer comment			
Ver.Indice	**Date**	**Name**	**Function**

Table of content

1. INTRODUCTION

The goal of this document is a description of how to extent the existent Salome component so that it can save its state in form of Python script with a usage of DumpPython mechanism implemented in the Salome platform.

2. GENERAL DESCRIPTION

The current implementation of DumpPython mechanism consists of two parts. The first is DumpStudy method of SALOMEDS_Study interface. The user hasn't any interaction with this method. The second part is a redefinition of virtual method DumpPython of SALOME_Component interface. The signature of this method is:

sequence<octet> DumpPython(in SALOMEDS::Study theStudy,
in boolean isPublished,
out boolean isValidScript)

The implementation of this method for the custom component is a goal of the extension of DumpPython mechanism (DPM). The way in which the custom component implements DumpPython method mostly depends on complexity of the component's data model. In general DumpPython method has to generate a Python script, which contains at least one function with predefined name RebuildData. The signature of this function is:

def RebuildData(theStudy)

For detailed description of DPM, please, look at the specification "Dump Salome study to Python".

3. DIFFERENT APPROACHES

Generally there are two main approaches for implementation of the method DumpPython.

1. Each method of the custom component that publishes in the Salome Study adds to published objects some additional data (for example a string presentation of Python command that being executed reproduced the published object) that allows the method DumpPython by iteration of the component's objects create a Python script, which presents the component data model. This approach is used in GEOM component.

   Advantages:
   The method DumpPython can be trivially implemented.
   Disadvantage:
   This approach requires a significant modification of the implementation of the custom component.

2. In this case the method DumpPython analyzes a content of the Salome Study and generates a Python script basing on the information retrieved from the study and the corresponding data stored in the component data model. This approach is implemented in PostPro component.

   Advantages:
   There is no need to modify the existent data model of the custom component.
   Disadvantage:
   For complex data model of the custom component implementation of the method DumpPython can be rather complicated task.

Thus, taking into account advantages and disadvantages of both ways it may be advised to use the first approach when creating a new component and the second when adding support of DPM to the existent component.

In fact as one can see it's possible to mix both approached when the most complicated objects in the custom component store the additional information thus simplifying the implementation of the method DumpPython and less complex objects is written in a Python script basing on analysis of their structures, which leads to decrease of modifications of the existent code.

In some cases a component is initially able to export its data model in the form of Python script, for example Salome Supervisor component. In this case implementation of DumpPython is trivial.

4. EXAMPLES OF IMPLEMENTATION

As an example of implementation of DumpPython method let's consider an imaginary Salome component, which can put in the Salome Study a component's host name.
The component has the following IDL interface:

```
module Test {
  interface HostObject : SALOME::GenericObject
  {
    string getHostName();
    void setHostName(in string theHostName);
  };

  interface testComp : Engines::Component, SALOMEDS::Driver
  {
     HostObject generateHostObject(in string theHostName);
     void addInStudy(in SALOMEDS::SObject SO, in HostObject HO);
     HostObject getHostObject(in SALOMEDS::SObject SO);
  };
};
```

Here is a possible implementation of a method addInStudy, which puts given object in a study (other methods don't play a part in the implementation of the method DumpPython)
Note: the code given below cannot be compiled and is written in pseudo C++ style.

```
void testComp::addInStudy(SALOMEDS::SObject SO, HostObject HO)
{
   SALOMEDS::Study_var Study = SO->GetStudy();
   SALOMEDS::StudyBuilder_var SB = Study->NewBuilder();
   SALOMEDS::AttributeIOR IOR= SB->FindOrCreateAttribute(SO,
"AttributeIOR");

   char* ior = orb->object_to_string(HO);
   IOR->SetValue(ior);

   return;
}
```

Now, let's consider a modification of the custom component testComp to add a DPM support to it. As it was stated above it could be done in two main ways, the first when each object stores some extra data allowing the method DumpPython to easily generate a Python script and the second approach when the method DumpPython generates the Python script basing on the content of the study and the component internal data.

4.1. Adding supplementory data to component's objects

First, let's modify the HostObject declaration to add to it an ability to store auxiliary data.

```
interface HostObject : SALOME::GenericObject
   {
     string getHostName();
     void setHostName(in string theHostName);

      void setDescription(in string theCommand);
      string getDesctription();
   };
```

Now lets' modify the method addInStudy of the component testComp to enable it to add a Python command presented as string to published in the Salome Study HostObjects.

```
void testComp::addInStudy(SALOMEDS::SObject SO, HostObject HO)
{
   SALOMEDS::Study_var Study = SO->GetStudy();
   SALOMEDS::StudyBuilder_var SB = Study->NewBuilder();
   SALOMEDS::AttributeIOR IOR= SB->FindOrCreateAttribute(SO,
"AttributeIOR");
```

```
    char* ior = orb->object_to_string(HO);
    IOR->SetValue(ior);

     //Add a Python command, which being executed reproduce the HostObject
     if(strlen(HO.getDescription()) == 0)  { //No Python script is associated
        string hostname = HO->getHostName();
        string cmd = "obj = testComp.generateHostObject(";
        cmd+=hostname;
        cmd+=")\n";

        //Associate a Pthon command with HostObject
        HO->setDescription(cmd.c_str());
      }

      return;
}
```

Of course the method addInStudy is not the only place, which can be modified to add to the component a support of DPM. In the example testComp it's possible, for example, to modify a method generateHostObject, so any generated HostObject would contain an associated Python command. It's up to a creator of the component to choose the right place to add auxiliary information to produced by the component objects.

Here is implementation of the method DumpPython for the testComp component.

```
TMPFile* DumpPython(CORBA::Object Study, bool isPublished, bool& isValid)
{
  IsValid = false;

  //Start the Python script generation
  string aScript = "import testComp\n";
  aScript += "def RebuildData(theStudy):\n";

  if(isPublished) {
    aScript+= "\t SB = theStudy.NewBuilder()\n";
    aScript += "\t comp = theStudy.FindComponent(\"testComp\")\n";
  }

  SALOMEDS::SObject aSO;
  SALOMEDS::Study ST = SALOMEDS::Study::_narrow(Study);
  aSO = ST->FindComponent("testComp");

  //Iterate all SObjects under the testComp
  SALOMEDS::ChildIterator itr = ST->NewChildIterator(aSO);
  for(; itr->More(); itr->Next()) {
    aSO = itr->Value();
    HostObject ho = HostObject::_narrow(aSO->GetObject());
    aScript += "\t";
```

```
    aScript += ho->getDescription();
    if(isPublished) {
      aScript += "\t SO = SB.NewObject(comp)\n"
      aScript += "\t testComp.addToStudy(SO, obj)";
    }
  }

 aScript +="\t pass\n";

  isValid = true;
 TMPFile* tmpFile = new tmpFile(aScript.size(), aScript.size(), aScript, 1);
   return tmpFile;
}
```

The result of the method DumpPython is a Python script like the following:
(if isPublish variable is set to "true")

```
import testComp
def RebuildData(theStudy):
        SB = theStudy.NewBuilder()
        comp = theStudy.FindComponent("testComp")
        obj = testComp.generateHostObject("myhost")
        SO = SB.NewObject(comp)
        testComp.addToStudy(SO, obj)
        pass
```

4.2. Generation basing on the content of the study

In this case there is no need to modify the existent interfaces. It's only necessary to implement the method DumpPython.

```
TMPFile* DumpPython(CORBA::Object Study, bool isPublished, bool& isValid)
{
  IsValid = false;

  //Start the Python script generation
  string aScript = "import testComp\n";
  aScript += "def RebuildData(theStudy):\n";

  if(isPublished) {
    aScript+= "\t SB = theStudy.NewBuilder()\n";
    aScript += "\t comp = theStudy.FindComponent(\"testComp\")\n";
  }

  SALOMEDS::SObject aSO;
  SALOMEDS::Study ST = SALOMEDS::Study::_narrow(Study);
  aSO = ST->FindComponent("testComp");

  //Iterate all SObjects under the testComp
  SALOMEDS::ChildIterator itr = ST->NewChildIterator(aSO);
```

```
  for(; itr->More(); itr->Next()) {
    aSO = itr->Value();
    HostObject ho = HostObject::_narrow(aSO->GetObject());

    string hostname = HO->getHostName();
    aScript += “obj = testComp.generateHostObject(”;
     aScript += hostname;
     aScript +=”)\n”;

    if(isPublished) {
      aScript += “\t SO = SB.NewObject(comp)\n”
      aScript += “\t testComp.addToStudy(SO, obj)”;
    }
  }

 aScript +=”\t pass\n”;

  isValid = true;
 TMPFile* tmpFile = new tmpFile(aScript.size(), aScript.size(), aScript, 1);
   return tmpFile;
}
```

The resulting Python script is the same as in the first case.

5. CONCLUSION

This document demonstrated two possible approaches to add a DumpPython functionality to the custom component. An approach to choose depends on several points. The main reasons to take into account are: a complexity of component’s objects, whether the component is just created or it’s a modification of the existent component, what is a level of support of Python in the component.