The Dynamic Template Pattern

Fernando D. Lyardet

LIFIA - Dto. de Informática,
Facultad de Ciencias Exactas, UNLP
fer@sol.info.unlp.edu.ar

DYNAMIC TEMPLATE (Object Structural)

Intent

Decouple instances from their classes so that those classes can be implemented as instances of a class.

The Dynamic Template allows the creation of new object templates dynamically and the establishment of single inheritance relationships among these templates at runtime. A Template and instance attributes may also be added at runtime.

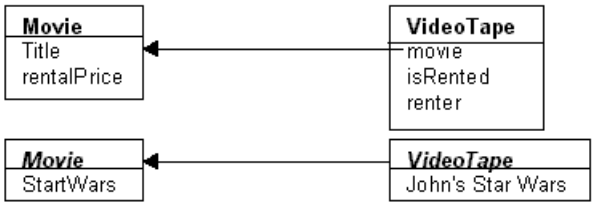
Motivation

As an introduction consider the videotape example presented for the Type Object [Woolf 96]. Since many of the videotapes are very similar, the videotape's instance share a lot of redundant information. Following this example, all copies of "Star Wars" would be similar: they have the same title, rental price, and other data. Thus, the repetition of that information throughout all copies of "Star Wars" would be redundant.

There are many ways to overcome this problem. One approach is to subclass VideoTape for each movie. Each time a movie is added to the system, it would be necessary to add a new subclass and recompile the system. This approach is neither practical nor flexible.

They key problem is how to deal with the fact that each type of videotape needs to be an instance of a class of type videotape.

As expressed in Woolf[96], the key step is to implement two classes, one of whose instances represents application instances and the other one whose instances represents what would normally be the application classes. Each application instance has a pointer to its corresponding class-instance. So, to implement this solution two classes should be involved: one to represent the VideoTape and other to represent the Movie.



The last solution adds the necessary flexibility to allow changing the class of a VideoTape at runtime by simply changing its class reference. However, there are a number of possibilities that would be interesting to analyze: for instance, to protect tapes from video piracy, the industry adds a special code to their movies so that each new movie would have a special code/identification.

We could solve this problem with the techniques described above; but even by using the Type Object pattern, it would be necessary to recompile the system, we are unable to change or add a class variable to support the Movie code.

With the Dynamic Template pattern, we can overcome that problem by adding a class attribute dynamically:

```
{Borland Delphi code example}
var     Movie:DynamicTemplate;
        MyVideo:videotape;

begin
  Movie:=DynamicTemplate.create;      {Create a movie class}
  With movie do
    Begin
       name:='Star Wars' ;    {set movie name}
       price:=$5.75;   {set movie price}
    end;
  MyVideo:=Movie.CreateInstance;
Movie.AddAttribute('Code' );
{access the attribute…}
Movie.attributeByName('Code').value:=17893;
End;
```

A second example is the design and implementation of CASE environment [Lyardet96] to support an OO hypermedia design methodology (OOHDM [Schwabe96]).

A Hypermedia is composed of nodes and links. Under this methodology, you model the domain (Abstract design) using well-known object-oriented modeling principles. The next stage defines the navigational structure (Takes into account user profile and task and focuses on the cognitive and architectural aspects of a hypermedia system), establishing the Node and Link classes.

Finally you also have the interface design for each node class, where the designer sets the appearance of each node class. Once you finish with the design, you still don't have a hypermedia, because you don't have either nodes or links, but descriptions of their classes, their interface, and navigational contexts.

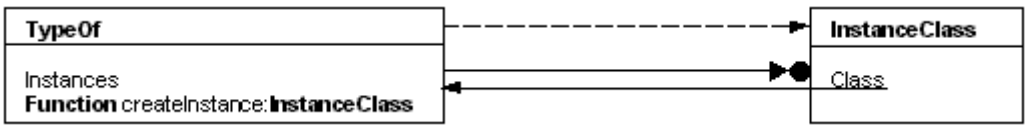
With that representation, you can create an instance of any node class and fill it with data (for example, an instance of a painter: "DaVinci", and after that many other instances of the class "paintings" for each Leonardo´s artwork).

Now you have a hypermedia that is "platform independent" that can be mapped to html, Asymetrix´s Toolbook, Director, etc. The tool must be able to model every design component as a class and support any kind of operations upon itself, as if it were an actual class. Operations on a class, can involve adding class and instance attributes, inheritance (is-a relationships) and composition (part-of relationships).

Another goal of the system is to provide the ability to generate a working hypermedia application for a given platform (html, Toolbook, etc). Such automated implementation should be generated from the OO model given by the user. Thus, there is an implicit need for supporting instances of model classes, such as node and link instances.

As a Final example, this pattern could be in the context of an Object Oriented Database (OODB). I order to define its scheme, an OODB must be able to define classes dynamically and store instances of them. Furthermore, it should be possible to modify an OODB scheme, which may imply class versioning or class modification.

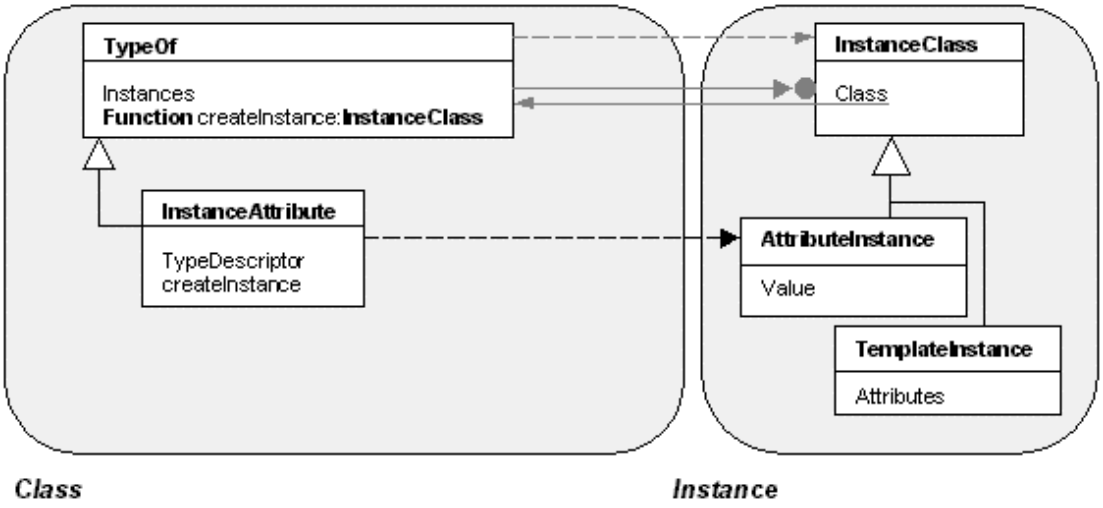
At this point, we borrow from the Type Object the strategy of having separate classes whose instances behave as both classes and instances. Since it is very difficult (or impossible) to add behavior to a class at runtime, what we create are not "real" classes but template objects. The structure of these templates is defined dynamically, and they are able to create instances of them.



The Dynamic Template pattern describes how dynamically created classes can be extended through inheritance or by adding class and instance attributes, and the way the instances of these classes are generated. It also describes how the latter behave as actual instances whose type and number of attributes may vary at runtime.

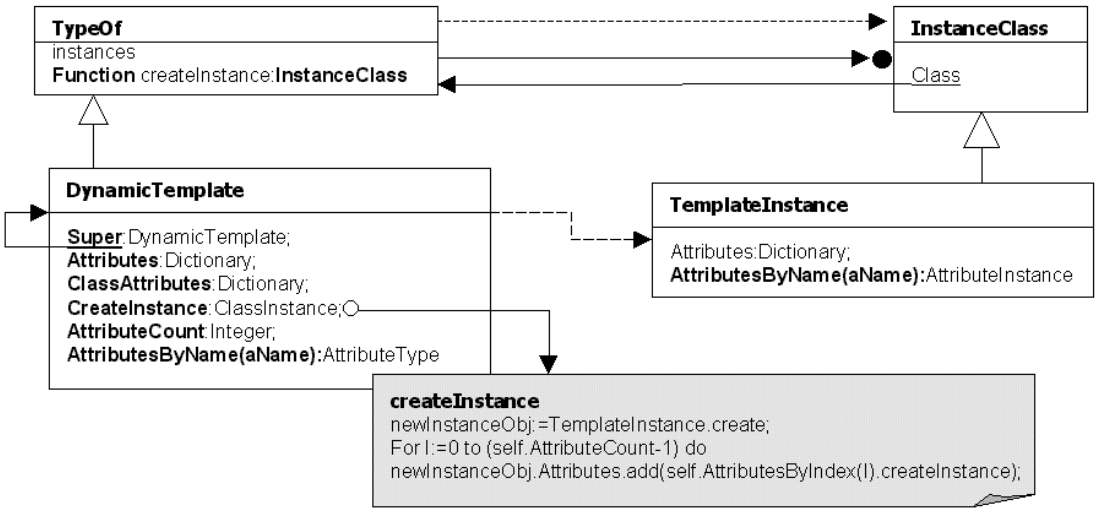
To provide a dynamic mechanism for adding attributes to a class, we need to describe them in a way such that the creation of class instances with the specified attributes

could be possible. These descriptions of the attributes could be further specialized to specify whether the attributes would be class or instance attributes at the moment of instantiation.



The classes, whose instances represent actual system instances, are all descendants of the InstanceClass, such as AttributeInstance and TemplateInstance. The former is the AttributeType counterpart and holds the actual value of an instance attribute value, while the latter represents the instances of a class.

The AttributeType class provides a way to describe a class attribute and works as a factory method depending on the typeDescriptor value. When an instance of a dynamic class is being created, the class asks to its attributes for an instance of themselves that will belong to the new instance.



The idea of supporting single inheritance through a dynamic subclassing mechanism relies on the ability of creating a virtual concatenation of a class attributes to its superclass attributes.

To provide this feature, each DynamicTemplate instance has:

- Its instance and class attributes stored in dictionaries.
- A method that tells how many attributes the class have (no matter whether they are inherited or not) (-attributesCount-)
- A reference to its superclass.

When the superclass of one of the dynamic classes is set, the new subclass sets its "super" reference to the given class. At this step, we could setup the observer pattern[GHJV95] that would notify its instances and subclasses.

All dynamically created classes and instances of these classes have access to their attributes through a method to specify whether the attributes are to be fetched by name or through an index.

```
{Example}
var AttributeValue:String;
…
AttributeValue:=MyObject.attributesByName('MovieRenter').value;
```

Or

```
var AttributeValue:Integer;
…
AttributeValue:=MyObject.attributesByIndex(indexNumber);
```

This approach allows hiding whether the attributes are inherited or not by providing a uniform access mechanism. Both DynamicTemplate class and TemplateInstance implement the AttributeBy… methods to get access to their attributes, with some differences:

- In the DynamicTemplate class these methods provide access to class attributes searching through its superclasses if necessary, and InstanceAttrBy… methods to access instance attributes definitions.
- TemplateInstance provides access to its attributes without the need of extra searching since as an instance, it already holds all its attributes.

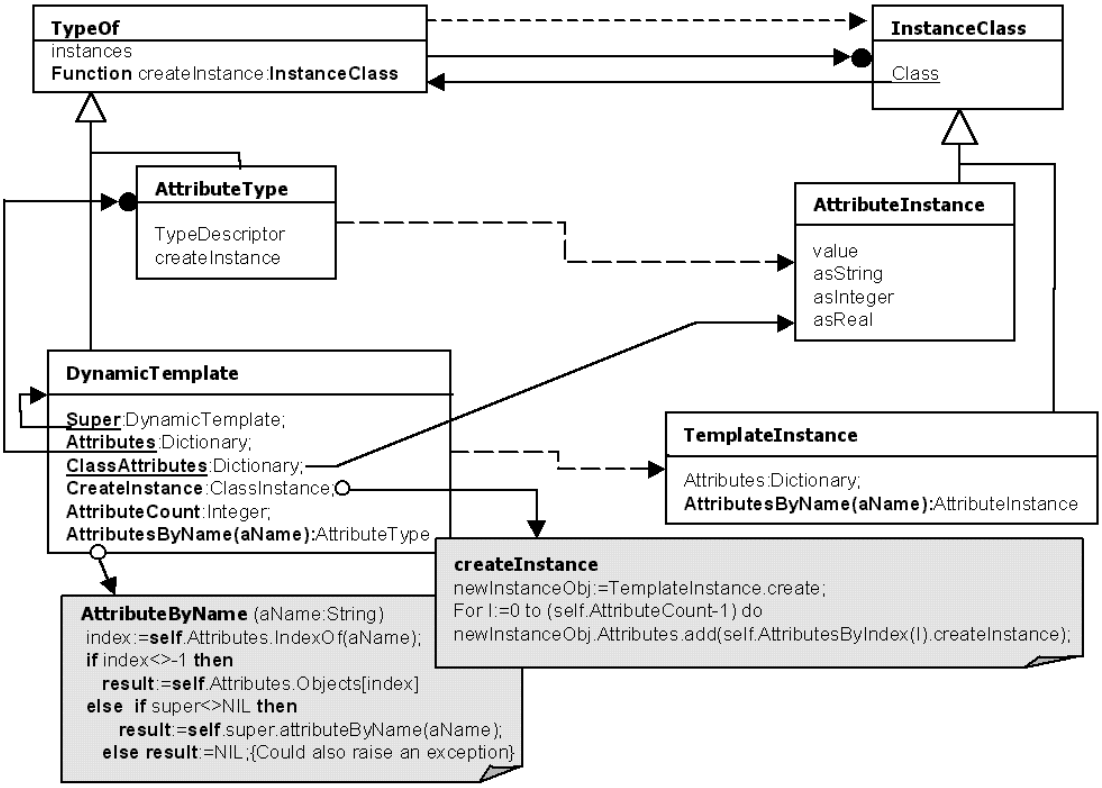
```
{The following example shows how to build a dynamicTemplate, define its structure}
and then creating instances from it.}

    Var
      MovieTemplate:DynamicTemplate;
      VideoTape1,VideoTape2:TemplateInstance;
    Begin

     MovieTemplate:=DynamicTemplate.create;    {Create a new template class}
     with MovieTemplate do                     {Cascade messages to ListTemplate}
      begin
       AddClassAttribute('MovieName');       {Add an attribute to hold the name}
       AddClassAttribute('MoviePrice');      {Another for the movie price      }
       AddInstanceAttribute('isRented');    {Now we define a couple of instance}
       AddInstanceAttribute('renter');      {attributes                         }

      {Now we set the template's class attributes values}
      ClassAttributesByName(´MovieName´).Value:= 'Men In Black';
      ClassAttributesByName('MoviePrice').Value:= $4.95;

      end;

     VideoTape:= MovieTemplate.CreateInstance;       {We create an instance}
     with VideoTape do
      begin
       AttributeByName('isRented').Value:=True;
       AttributeByName('renter').Value:=´John´;
      end;

     {Now, We create another instance from de template,}
     { to hold another videoTape…                      }
     VideoTape2:=MovieTemplate.CreateInstance;{We create another instance}
     with VideoTape2 do
      begin
       AttributeByName('isRented').Value:=True;
       AttributeByName('renter').Value:= 'Alicia';
      end;
    end;
```

Applicability

Use the DynamicTemplate pattern when:

- Instances of a class need to be grouped together according to their common attributes, and be able to organize their grouping into further subclasses at any depth.
- You want to be able to add class or instance attributes at runtime.
- In addition, it can be used when the Type Object pattern does not provide the required flexibility.

Structure:



Participants

- DynamicTemplate (TypeOf)
  - Is the class of Dynamic Template.
  - Has a separate instance for each new dynamic class created.
  - Supports single inheritance between instances.
  - Supports getting access to the attributes by their names or index.

- TemplateInstance (InstanceClass)
  - Is an instance of DynamicTemplate.
  - Implements the mechanism to support accessing to the attributes by their names or index.

- AttributeType(TypeOf)
  - Specify the name and describes a DynamicTemplate attribute;
  - Overrides the createInstance method and returns an instance of AttributeInstance.

- AttributeInstance (InstanceClass)
  - Its instances represent the attributes of a class instance and hold values.

Consequences

The advantages of the Dynamic Template pattern are:

- Allows adding templates dynamically as well as adding or deleting template or instance attributes, and establishing its superclass at runtime.
- Allows the Object to change its DynamicTemplate.
- Allows a DynamicTemplate to change its superclass.
- Gives to the system a very flexible capability to handle and enhance the information management.

- Since Dynamic Template pattern provides the same separation between objects that behave as classes and objects that are to be instances of classes, the Type Object pattern is "included" within the Dynamic Template pattern. Therefore, it would be possible to use this pattern as a Type Object in similar contexts to provide extra functionality. Nevertheless, the main use of this pattern is in the context of highly dynamic systems, as those presented in the motivation section.

The disadvantages of the Dynamic Template Pattern are:

- The Dynamic Template pattern shares the same disadvantages of the Type Object pattern plus certain discomfort to handle attributes through a method instead of just sending a message. This limitation is due to the need of doing some computation to fetch the desired attribute.
- Inheritance mechanism works only with DynamicTemplate objects. That is, you cannot set an arbitrary class to become its superclass. It must be a DynamicTemplate descendant.
- Attribute access is slower than normal variable access.
- Although part of the structure of a DynamicTemplate objects can be set statically, the attributes specified this way cannot be inherited. Thus, to exploit the inheritance feature and to access the attributes in a uniform way (without having to distinguish between statically and dynamically defined attributes), all attributes should be defined dynamically.
- Following the above disadvantage, the DynamicTemplate and TemplateInstance objects cannot statically determine their structures.
- It is difficult to determine if a variable in a dynamic template is being used since we cannot simply search the source code.

Implementation

There are a number of issues to consider when implementing the Dynamic Template pattern:

1. Class instance references DynamicTemplate: since the programming language does not know the relationship between a TemplateInstance object and a DynamicTemplate object, the relationship and meaning of the class reference of classInstances must be established by the software designer.
2. *Dynamically change the class of instances*: to change the class of an instance may take an extra effort, because the idea of creating a full class does not allow us to assume the new class will have a similar structure (as it happens with the type object). We have to ensure instance attributes are consistent in meaning and structure with the class. Mutation of the attributes could be a possibility, matching instance attributes with the new class attributes by name, and adding/deleting remaining attributes.
3. *Reflecting Class changes to their instances*: reflecting class changes to its instances is a class responsibility. In languages such as Delphi or C++ do not provide their basic Object classes with the ability of sending a “self changed:” message. Thus, the notification capability should be implemented either by the DynamicTemplate itself or by delegating this functionality to specialized objects that behave as Observers or DependencyTransformers[Woolf94] (both also should be programmed).
4. *Attributes ability for holding values*: AttributeInstance instances have the ability of holding the actual attribute’s data (value). The problem arises with statically typed OO languages such C++, where type information is needed at compile time. An easy way to overcome this problem could be the following:
    - Adding a symbol (identifier) to AttributeType that specifies the type of the attribute.
    - Adding the methods asString, AsInteger, etc to AttributeInstance Class and derive new subclasses to support value for all specified types.
    - The createInstance method of Attributetype would work as a factory method, to match the creation of the attribute instance with the attribute’s type.

A word about Delphi implementation: Borland's Delphi incorporates a new type called **Variant**[1] whose type does not need to be specified at compile time. This feature allows a cleaner and simpler implementation (much like a Smalltalk implementation), since the problem of typing is eliminated.

Now we present an outline of the implementation. A complete implementation of this pattern is freely available on the web in both Borland's Delphi, and Smalltalk (C++ version coming soon!) versions at: ***http://www-lifia.info.unlp.edu.ar/~fer/plop97/codeSamples.html*** .

Sample Code

```
type
TypeDescriptor=(vSmallint, vInteger, vSingle, vDouble, vCurrency, vDate,
                vOLEStr, vDispatch, vError, vBoolean, vUnknown, vString);
TypeObject= Class(TObject)
              public
              instances:Tlist;
              Function CreateInstance:InstanceClass; Virtual;
            end;

AttributeType=Class(TypeObject)
                public
                 Name:String; {the attribute´s name}
                 ValueType:TypeDescriptor;
                 Function CreateInstance:InstanceClass; Override;
              end;

DynamicTemplate= Class(TypeObject)
           Private
            PropStartIndex:Integer;
            Attributes,ClassAttributes:TStringlist; {TstringList works much like}
                                                    {a Dictionary in Smalltalk  }
           Public
             Super:DynamicTemplate;
             Function attributeCount:integer;
             Function ClassAttributesByName(aName:String):AttributeInstance;
             Function attributesByName(aName:String):AttributeType;
             Function attributesByIndex(index:Integer):AttributeType;
             Function AddInstanceAttr (name:String):Integer; {Error Codes}
             Function AddClassAttr    (name:String):Integer;
             Function CreateInstance:InstanceClass;  Override;
           end;


InstanceClass= Class(TObject)
                Class:TypeObject;
                Value:Variant;
               end;

TemplateInstance  = Class(InstanceClass)
                  Private
                   attributes:TStringlist; {Works much like a Dictionary in Smalltalk}
                  Public
                  Super:DynamicTemplate;
                  Function attributesCount:integer;
                  Function attributesByName(aName:String):AttributeInstance;
                  Function AttributesByIndex(anIndex:Integer):AttributeInstance;
            end;


{The following method, creates a template's instance}
{It simply iterates over the AttributeType objects (no matter whether they're}
{inherited or not) and generates their AttributeInstance counterpart, and adds the}
{latter to the newly created instance                                          }

Function DynamicTemplate.CreateInstance:InstanceClass;
```

[1] The Variant type is capable of representing values that change type dynamically. Whereas a variable of any other type is statically bound to that type, a variable of the Variant type can assume values of differing types at run-time. The Variant type is most commonly used in situations where the actual type to be operated upon varies or is unknown at compile-time.

```
 var newInstance:TemplateInstance ;
     i:Integer;
     AttribDef:AttributeType;     { Attribute definition }
     AttribInst:AttributeInstance;{ Attribute instance   }
 begin
  newInstance:=TemplateInstance.create;
  For i:=0 to self.Attributes.Count-1 do
   begin
     AttribDef:=self.AttributesByIndex(i);
     AttribInst:=AttributeInstance(AttribDef.CreateInstance);
     newInstance.Attributes.AddObject(AttribDef.Name,AttribInst);
   end;
 Result:=newInstance;
 end;
```

Each dynamic class is able to determine whether a required attribute belongs to it or its superclass through the following methods:

```
{searches the desired attribute by name}
Function DynamicTemplate.AttributesByName(aName:String):AttributeType;
var index:Integer;
begin
 index:=self.Attributes.IndexOf(aName);
 if index<>-1 then
    Result:=AttributeType(self.Attributes.Objects[index])
 else if super<>NIL then
         Result:=self.super.attributesByName(aName)
      else Begin
            raise EInvalidAttributeName.Create('Invalid attribute name');
            Result:=nil;
          End;
end;
```

{for complete implementation, see the **URL** specified above or contact the author. The code presented above is incomplete and may be not enough to understand some details. Please, do NOT hesitate contacting me.}

Related Patterns

A DynamicTemplate could notify its instances as the aSubject does to its Observers in the Observer pattern (page 237, [GHJV95]). This can be implemented by inheriting the notification mechanism from an Observer (not very nice), or it may delegate the notification of its changes to an observer or a more specialized subclass of it that would work the same way aDependencyTransformer does.

All other patterns related to the TypeObject pattern are also related to the DynamicTemplate pattern.

References

[Bor96] Borland Delphi 2.0 User Reference. Visual Form Inheritance http://www.borland.com/

[GHJV95] Gamma, Helm, Johnson, Vlissides. Design Patterns: *Elements of Reusable Object Oriented Software*. Addison-Wesley, Reading, MA, 1995;

[Lyardet96] Lyardet, Fernando & Rossi, G. *"Enhancing Productivity in the Development of Hypermedia Applications*". Proceedings of the Workshop on Next Generation CASE Tools (NGCT'96) held in CaiSE'96. Crete, Grece.

[Parc96] VisualWorks 2.5.1 Documentation. 1996, ParcPlace-Digitalk http://www.parcplace.com/

[Schwabe96] Schwabe D. And Rossi G. "*Systematic Hypermedia Design with OOHDM*" Proceedings ACM Hypertext '96.

[Woolf94] Woolf, Bobby. "*Improving Dependency Notification*". The SmallTalk Report (a Sigs publication), Vol4 No3,1994.

[Woolf96] Woolf, Bobby. "*The Type Object Pattern*". 1996.