TRANSLATOR PATTERN

Class Behavioral

Intent

Given an object class that represents an abstract concept, defines the rules for translating this abstraction into predefined concrete forms.

Motivation

It often happens that you use classes and objects to represent abstract concepts. All the processing can be expressed in terms of this concept, but at some point you might need to map it to a some concrete form of the concept, if only to communicate it to or from some user.

The details of this concrete form can vary widely from user to user. For example, if I'm thinking of a common domestic canine, I'll say "dog" to an English-speaking user, but I'll say "chien" to a French-speaking user and "Hund" to a German-speaking user. I always refer to the same abstract canine, but the concrete form varies according to the person that I'm talking to.

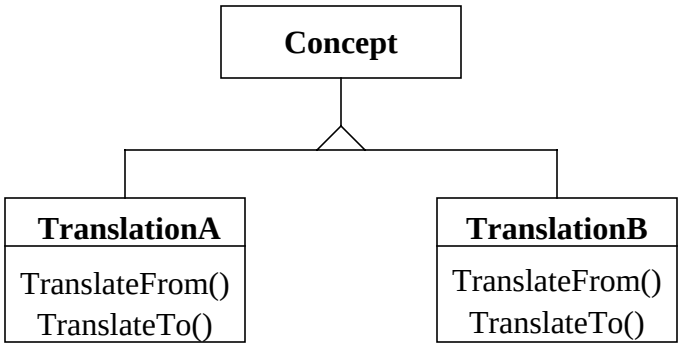
The Translator pattern describes how to organize a series of translation rules around an abstract concept. In the above example, the pattern described how an abstract concept (a canine) was expressed as different string objects depending on some translation rules.

The Translator pattern uses a class to represent the abstract concept and subclasses for each of the possible concrete classes. The subclasses only add the translation rules to the parent class.

I would call the parent class "abstract" and the subclasses "concrete", but that would introduce unwanted confusion with abstract and concrete classes. I will therefore use the term "Concept" to mean the abstract concept to by translated, and the term "Translation" to mean some concrete form of the Concept. The actual conversion will be termed a "Translation Rule".

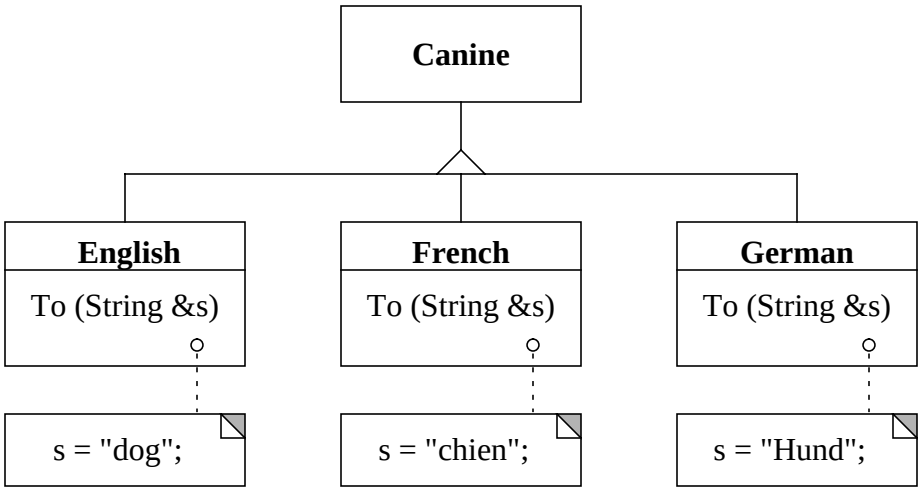
Using polymorphism, a Translation instance can be used in place of a Concept instance, but some way must be defined so that we can initialize a Translation instance with a Concept instance. This can be achieved in C++ through constructor methods.

A typical hierarchy might look like the following:



Note that Concept, TranslationA, and TranslationB are all concrete classes. Concept can be used for processing within a system. But when comes the time to pass the resulting Concept instance to some other system, either TranslationA or TranslationB instances must be used.

Part of the above canine example would therefore look as follow:



We can create a Translation for each possible output format without ever touching the Concept class. A user of the Concept needs only know what forms he wants to translate it to and use the proper Translation (or implement his own).

To actually translate a concept from form A to form B, a system could use an object `aTranslationA` to read the Concept (using the `TranslateFrom()` method), initialize an object `aTranslationB` with `aTranslationA`, and then use `aTranslationB` to write back the result (with the `TranslateTo()` methof). This would produce the desired translation of the Concept from form A into form B.

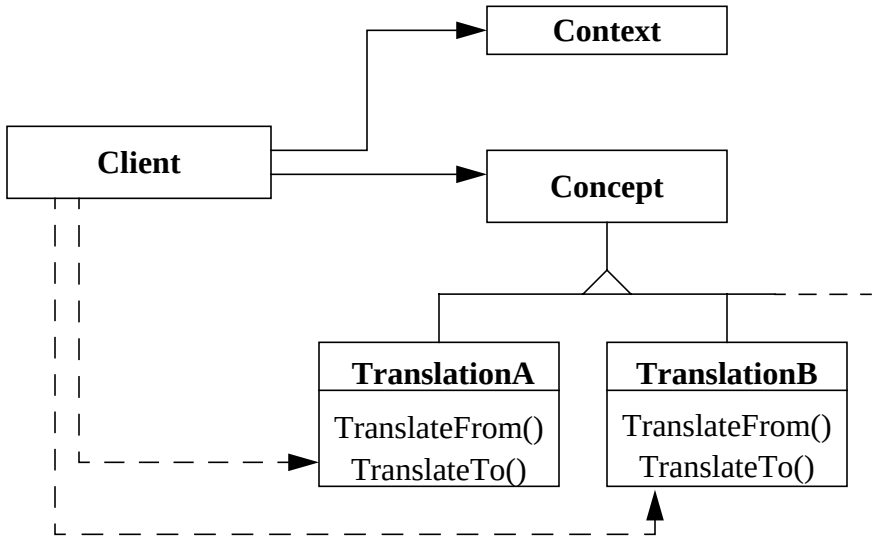
Applicability

Use the Translator pattern when there is an abstract concept to be represented in a variety of similar forms. The Translator pattern works best when:

- There is a large number of possible Translations, and new ones can be expected to be added at various points of the development cycle. To merge the Translations into the Concept (or even simply merging them together into some form of Translation manager) would produce a large class that would be difficult to maintain. It might also obscure the main purpose of the Concept.
- The Translations have a high level of similarity. Some semi-automatic mechanism can then be used to facilitate their generation.
- The concrete forms of the Concept are some lower-level data formats on which the designer has no control, for example the parameters to some API calls.
- The set of Concepts is much smaller than the set of possible Translations, or when specific concrete forms are not meaningful to many Concepts. It would be unthinkable to build a Concept out of every word in the dictionary and then English and French Translations for every Concept in order to implement an automatic translator. The explosion of classes would be staggering. In such a case, some other

tool might be more appropriate.

Structure



Participants

- **Concept**
  - An instance is required for each abstract concept that requires translation.
  - Offers no TranslateTo/From interface, only whatever interface is required for the proper processing.
- **Translation**
  - Adds (and implements) a TranslateTo/From interface to the Concept
  - Maintains the conversion rules for this particular conversion.
- **Context**
  - Contains the translation rules (which Translation to use and when).
- **Client**
  - Uses Concept for whatever processing that needs to be done.
  - Instantiates the required Translations as the need for them occurs.
  - Applies the translations as instructed by the Context.

Collaborations

- The Client instantiates the proper Translation according to the rules found in the Context
- Each Translation performs its TranslationFrom of TranslateTo method as requested, following the proper Translation Rule.

Consequences

The Translator pattern has the following benefits and liabilities:

1. *It's easy to add new Translations.* Because the pattern uses subclasses to represent specific translation rules, you can duplicate the process to add new rules and therefore enlarge the set of Translations about a given Concept.
2. *Implementing a Translation is easy too.* Since it deals with only one conversion, the subclass is rather small and therefore easier to implement and less error-prone. In

addition, Translations are often similar to one another, which facilitates code reuse. But this is a mixed blessing, as it also means that the implementor must be careful of the differences and treat them accordingly.

3. *It keeps the Concept focused.* The Concept only deals with operations that pertain to its role. It doesn't need to deal with the details of specific concrete forms, that work is left to the Translations. This helps further encapsulation.

Implementation

Consider the following issues when implementing the Translator pattern:

1. *Is the Translation simply a concrete form or a new Concept?* A Translation should be kept as simple as possible and should only provide conversion methods. If it requires new data fields within the object, then a new Concept should be defined instead (as a subclass of the original Concept).
2. *Mind the differences between Translations.* Some Translations may differ only slightly and great care must be given in implementing Translation Rules correctly.
3. *Translations are leaves in the inheritance tree.* Translations should not be subclassed. Their purpose is solely to put the Concept in a specific concrete form and I cannot think of a case where this behaviour could be extended upon.
4. *Use conversion operators whenever possible.* In C++, classes can define implicit conversions by defining special constructors and conversion operators. This makes for Concepts that are easy to type-cast into Translations in order to operate a given conversion. You must be careful, though, that if you define a conversion constructor for a Translator, you must also defines appropriate constructors that correspond to all the other constructors from the Concept.

Sample Code

Here is an example in C++ of how the Translator pattern might be implemented. In OSI network management, an attribute-value assertion (AVA) is a coupling between an object identifier (a string of digits identifying a given object, here a data type) and a value denoted as an ANY (equivalent to C's void pointer). It is defined in ASN.1 (a data type specification language) as:

```
AttributeValueAssertion ::= SEQUENCE {
      attributeType  OBJECT IDENTIFIER,
      attributeValue ANY DEFINED BY attributeType}
```

Depending on the context where the AVA is used, a number of aliases have been defined to make the type name more significant:

```
Attribute   ::= AttributeValueAssertion
ActionInfo  ::= AttributeValueAssertion
ActionReply ::= AttributeValueAssertion
EventReply  ::= AttributeValueAssertion
```

When the ASN.1 is compiled to produce C structures for some application, most compilers produce a separate data type for each alias. The result is that various AVAs may not be assignment compatible, depending on their source.

The solution here is to use the Translator pattern. The Concept is the AVA and the Translations are the various aliases. We first define a class CAVA to represent the Concept. It uses multiple inheritance to fetch the properties of both object identifier and ANY types which were defined before hand:

```
class CAVA : public COID, public CAny {
public:
      CAVA ();
      CAVA (const COID &o, const CAny &a);
      CAVA (const CAVA &a);

      // Other members...
};
```

Then we define the Translations that are required. Suppose that the first alias listed above yielded the data type `ANS1_Attribute`, we would obtain the following Translator:

```
class CAttribute : public CAVA {
public:
      CAttribute ()                                          {; }
      CAttribute (const COID &o, const CAny &a) : CAVA(o, a){; }
      CAttribute (const CAVA &a)                : CAVA(a)   {; }

      CAttribute (const ANS1_Attribute &a);
      operator ANS1_Attribute () const;
};
```

And so on for `ASN1_ActionInfo`, `ASN1_ActionReply`, `ASN1_EventInfo`, and even `ASN1_AttributeValueAssertion`.

The first three constructors in the example are there simply to transmit the full set of constructors to the subclass. Their behaviour is rather straightforward in that they simply falls back on the equivalent constructor in the parent CAVA class.

The two other members are more significant. It is them that implement the actual conversion:

```
CAttribute::CAttribute (const ASN1_Attribute &a)
{
      // Initialize the COID and CAny parent classes
      // from the data contained in "a".

      return;
}


CAttribute::operator ASN1_Attribute () const
{
      ASN1_attribute temp;

      // Fill in "temp" with the values from the
      // data in the COID and CAny parent classes.

      return (temp);
}
```

Now the application can use the classes to deal with various forms of AVAs. Let's suppose the function `API_funcA (ANS1_ActionInfo *ptr)` fetches a value from some API and puts it in the variable being pointed to by `ptr` and that `API_funcB (ANS1_ActionReply val)` hands the value in `val` back to the API. A possible interaction might look like:

```
CAVA              ava;        // A Concept variable
ASN1_ActionInfo   info;       // A concrete form
ASN1_ActionReply  reply;      // Another concrete form

API_funcA (&info);            // Fetching the value
ava = (CAttribute) info;      // and translating it to a CAVA

      // Processing the action based on "ava" ...

reply = (CEventReply) ava;    // Translating the CAVA into a reply
API_funcB (reply);            // and sending it back
```

Known Uses

The Translator pattern is currently being used in a C++ framework for building CMIS interfaces. CMIS is the network management architecture used in OSI, the Open System Interconnection, standardized by ISO and ITU-T (formerly CCITT).

Related Patterns

Conversion: Each Translation Rule can be viewed as an application of the Conversion pattern.

Interpreter: Can be coupled to a Translator to produce a cross-compiler.

Adapter: The Adapter pattern might used this pattern in order to implement translation of parameters between object methods.

Contacts

http://www.iro/umontreal.ca/~keller/Layla

Jean Tessier, AT&T Laboratories, Advanced Technologies Division, Holmdel, NJ.

Jean.Tessier@att.com

Rudolf K. Keller, Département d'informatique et de recherche opérationnelle, Université de Montréal, C.P. 6128, succursale Centre-ville, Montréal (Québec) H3C 3J7, Canada.

keller@iro.umontreal.ca