Florian Hoffmann

# Enumeration Classes

## Design Pattern Suggestion

June 20th, 2016

### Introduction

One of the major gaps in the programming language ABAP is the lack of *enumerations*. Enumerations have been part of Java from the beginning on, and have been successfully adapted by a variety of other programming languages, such as C#. Enumerations make it easy to write clean and correct code and are generally preferred to all other means of specifying constants.

Unfortunately, the ABAP language group does not seem to have plans to introduce enumerations in the foreseeable future. Through trials and discussions with other groups I tried to identify a design pattern that anyhow gives us something similar.

This document describes the best pattern I found and explains why I think it is superior to other, similar patterns, especially our current best practice of specifying constants in interfaces.

### At a Glance

CLASS ce\_fra\_umml\_filter\_option DEFINITION

PUBLIC FINAL ABSTRACT.

PUBLIC SECTION.

INTERFACES if\_fra\_enumeration. " marker interface

CONSTANTS:

BEGIN OF sc\_value,

equal TYPE if\_fra\_umml\_types=>ty\_filter\_option VALUE 'EQ',

between TYPE if\_fra\_umml\_types=>ty\_filter\_option VALUE 'BT',

END OF sc\_value.

ENDCLASS.

CLASS ce\_fra\_umml\_filter\_option IMPLEMENTATION.

ENDCLASS.

### Cohesion

Our best practice for grouping constants in interfaces produces objects with low [*cohesion*](https://en.wikipedia.org/wiki/Cohesion_(computer_science)). Cohesion describes how good things that are grouped together really fit together. In general, our method produces only the unfavorable [*coincidental cohesion*](https://en.wikipedia.org/wiki/Cohesion_(computer_science)#Types_of_cohesion), meaning that the things are only grouped because someone more or less accidentally put them together.

For example, the constants interface *IF\_FRA\_COMMON\_CONSTANTS* lists *SC\_AUTHORIZATION\_RANGE* next to *SC\_SELECT\_OPTION*, things that have nothing in common. The name *common* alone suggests that this is a loose collection of general stuff, like a *utilities* or *helper* class.

The suggested enumeration class pattern in contrast forms objects with good cohesion, because it builds logical groups clearly specified by the class’s names. This also strengthens the single-responsibility principle.

### Discovery

One of the effects of bad cohesion is that it is hard to find things. For example, would you search for the constants that identify our solutions Fraud Management / Audit Management / Business Partner Screening in the interface *IF\_FRA\_STRATEGY\_CONSTANTS*? Usually the only way to find those is by a *where-used* search on the data type.

The suggested pattern makes it easier to find things because each enumeration has its own name and we can search things in transactions like SE80 by the class’s name.

We can further facilitate discovery by introducing a new prefix for enumeration classes, such as the *CE* in the example above. This way, typing *CE\_FRA* makes the code completion automatically suggest enumerations. I suggested “CE” because it is already used by a small number of other groups in SAP for the same purpose. I found regular class names and suffixes such as *CL\_FRA\_SOLUTION\_ENUM* inferior because they waste too many characters and don’t support auto-completion.

### No Realization

With interfaces, it is alluring to make classes implement constants interfaces to get shorter access to their constants. For example, the class *CL\_FRA\_TC\_HANA\_OBJECT\_MANAGER* “implements” the interface *IF\_FRA\_TC\_CONSTANTS* for the sole reason that it was shorter to write *CS\_DELIVERY\_UNIT* than *IF\_FRA\_TC\_CONSTANTS=>CS\_DELIVERY\_UNIT*.

This pseudo-realization can cause havoc in programming languages such as C++ because it forces the compiler to build huge dependency trees and recompile a lot of code if only a minor constant changes. ABAP seems to be less fragile here, but still this is not something that should be encouraged.

The suggested pattern prevents this because it does not use interfaces. The used classes are *ABSTRACT* and *FINAL* and cannot be inherited. Their constants can only be accessed in the one intended way.

### Validation

One of the major disadvantages of missing enumerations is that methods have to do a lot of validation to ensure that their input is really valid. For example, a CHAR02 input parameter can legally receive values such as “XX” although its enumeration only recognizes “01” and “02”. As a result, the first few lines in a method are usually dedicated to checks such as *IF ( iv\_value = sc\_const-01 OR iv\_value = sc\_const-02 )*.

These checks are fragile. Add a third constant and you need to adjust all of these checks, anywhere in the application. Usually, you would make a central method that checks this, but where would you put it?

Enumeration classes solve this in a natural way because they allow you to add methods that do the validation:

* IS\_VALID( iv\_value ): abap\_bool

Check whether the input is a valid constant value.

* ASSERT\_IS\_VALID( iv\_value ) RAISING cx\_fra\_illegal\_argument

Throw an exception if the input isn’t a legal constant value

As all enumeration classes have the same structure, these calls can be further delegated to more general, type-independent implementations such as *CL\_FRA\_ABSTRACT\_ENUM*. See the older trial *CL\_FRA\_DETECTION\_TYPE\_ENUM* for an example.

### Adding Value

Enumeration classes also allow adding other methods that add value to an enumeration. For example, you could add a *TO\_STRING* method that makes constant values printable for logs, or a *COMPARE* method that compares constant values in enumerations with an ordered character such as message severity.

### Synch with Domains

Usually, our constants cover data types that also have an underlying domain. Our constants interfaces have the issue that they don’t recognize changes to the related domains, and vice versa. For example, changing the domain value ‘03’ to ‘A4’ doesn’t automatically adjust the constant interface.

Automation is not getting better with enumeration classes, but another thing is: We can add unit tests that compare the domain’s values with the enumeration class’s constant values and fail if the two deviate. You still have to realign the two by hand, but at least the system tells you that it has to be done.

If we add a marker interface such as *IF\_FRA\_DOMAIN\_BASED\_ENUM* to all enumeration classes that also provides the name of the domain, we could write one global unit test that checks all enumeration classes in one go.

### Generate Classes

Building one abstract enumeration class that serves reuse methods for enumerations that inherit it is not possible because the data types of the constant values change and the method signatures need to adjust to that. However, as all enumeration classes look the same, we could implement a generator that produces them from domains in an automatic fashion.

### No Instances

Usually, we use only instantiable classes because it allows mocking them in unit tests. I don’t see a need for that with enumeration classes. We cannot mock our constants interfaces today, so we don’t lose anything with the switch to classes. When we add methods, mocking could become interesting, but this piece of functionality is usually coupled very tightly to the surrounding code and probably not required. Forcing instantiation would work in any way, but bloat the code unnecessarily.

Another solution developed at SAP builds enumeration classes in a different fashion: Here, each constant value is an object, i.e. an instance of the class:

<https://wiki.wdf.sap.corp/wiki/download/attachments/1252693218/Enumeration_Assistant.pps?api=v2>

(See also the [pattern description on Stack Overflow](http://stackoverflow.com/questions/20637925/is-it-possible-to-create-an-enumeration-enum-in-abap).)

CLASS ce\_filter\_option DEFINITION

PUBLIC CREATE PRIVATE.

PUBLIC SECTION.

CLASS-DATA:

sc\_equal TYPE REF TO ce\_filter\_option,

sc\_between TYPE REF TO ce\_filter\_option.

DATA mv\_value TYPE string.

CLASS-METHODS class\_constructor.

METHODS constructor IMPORTING iv\_value TYPE string.

ENDCLASS.

CLASS ce\_filter\_option IMPLEMENTATION.

METHOD class\_constructor.

sc\_equal = NEW ce\_filter\_option( ‘EQ’ ).

sc\_between = NEW ce\_filter\_option( ‘BT’ ).

ENDMETHOD.

METHOD constructor.

mv\_value = iv\_value.

ENDMETHOD.

ENDCLASS.

This reflects Java’s solution even more closely. The primary advantages are that this automatically restricts the values that can be passed to the allowed enumeration constants, avoiding the need for subsequent validations, and that the object gives immediate access to its functionality. On the downside, this requires “serialization” and “deserialization” in all places: for example, directly after reading a BOPF node you would first have to turn all character-like constants into object references. Another disadvantage is that this has to instantiate dozens to hundreds of objects for the constant values when loading an enumeration class. My trials didn’t suggest this is worth it. Although superior in theory, this pattern only makes sense if it’s provided by the platform itself.

### Summary

The suggested pattern is a pretty small change to our best practice. We make classes instead of interfaces. And we make one class per enumeration instead of grouping things.

My suggestion is to introduce the alternative class prefix, present developers with this possibility, and prefer this new pattern for new enumerations.