Design Pattern in OO ABL

Klaus Erichsen, IAP GmbH

Introduction

This PDF is the compiled and reworked version of a series of 3 blog entries.
It will have a look on 7 Object Oriented Design Pattern in the context of the OpenEdge OO ABL.

The original blog entries and the whole **sample source code** are available on line, **see links at end of this PDF**.

Wikipedia says: 'In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design.'

Patterns are conceptual work, the idea has been known to the public ever since the publication of the book 'Design Patterns: Elements of Reusable Object-Oriented Software', in 1995.

In a larger project, design patterns help to keep the code reusable, readable, and to avoid problems in the long term.

This article will describe how to implement some of the patterns in OpenEdge Object Oriented ABL – shortly named OO ABL. The chosen patterns are very common and often used. The view to the patterns is driven by daily, practical work.

Table of Contents

Pattern 1 – Builder (Creational Pattern)

The Builder Pattern is a Creational Pattern, it uses a second object to create and set up an object.

Imagine you have an object like this:

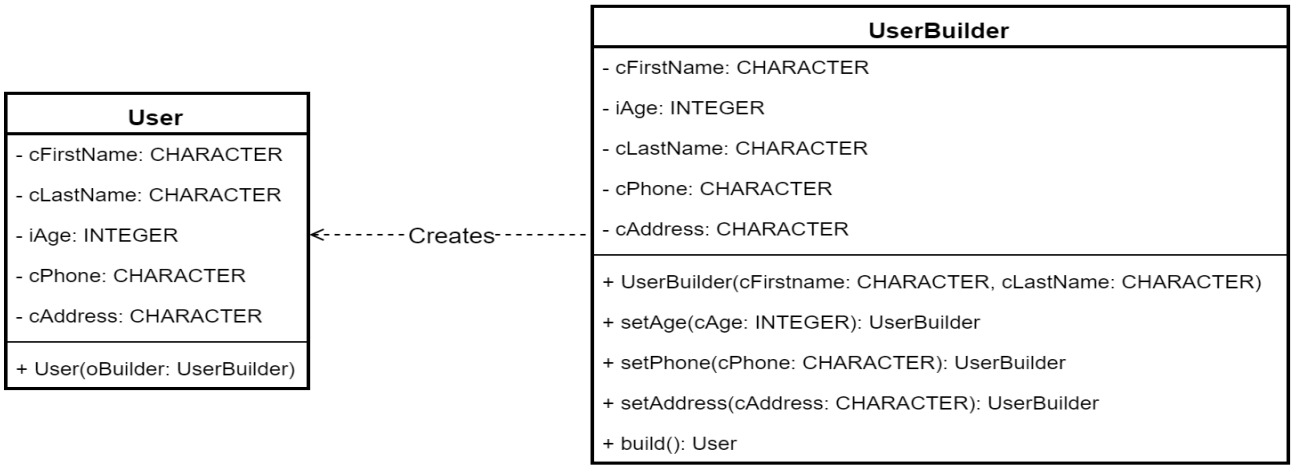
User
- cFirstName: CHARACTER - cLastName: CHARACTER - iAge: INTEGER - cPhone: CHARACTER - cAddress: CHARACTER
+ User(cFirstName: CHARACTER, cLastName: CHARACTER) + User(cFirstName: CHARACTER, cLastName: CHARACTER, iAge: INTEGER) + User(cFirstName: CHARACTER, cLastName: CHARACTER, iAge: INTEGER, cPhone: CHARACTER) + User(cFirstName: CHARACTER, cLastName: CHARACTER, iAge: INTEGER, cPhone: CHARACTER, cAddress: CHARACTER)

For several ways to create an instance of the ‘user’ there are several constructors in the class with more or less parameters.

So if you would like to create a user object with all parameters, the code looks like this:

```
RUN StatusCreate IN l-Import-Library-Handle
  ( INPUT l-DB-Cust,
    INPUT "",
    INPUT 150,
   ...
  INPUT
    "QtyType=" + OrderQtyQualifier + "{&T}"
    + "UTCTime=" + l-UTCTime
    + "{&T}"
    + "ConC-ID=" + SSCO-Ord.ConC-ID
  ...
  ) NO-ERROR.
  ) NO-ERROR.
```

With builder, the object diagram looks like this:



Besides the User object there is an UserBuilder object. This object has all the properties from the User object, but the constructor only has the mandatory properties (First and Last name). For every other property there is a method with a descriptive name, e.g. for iAge there is SetAge.

The secret is hidden in the setter methods:

```
CLASS UserBuilder:
...
    METHOD PUBLIC UserBuilder setAge(iAge AS INTEGER):
        THIS-OBJECT:iAge = iAge.
        RETURN THIS-OBJECT.
    END METHOD.
...
END CLASS.
```

The setter method sets the according property and returns itself (THIS-OBJECT). That means, the output is the builder object and this can be used to set another property or to complete the object.

```
DEFINE VARIABLE oUser AS User NO-UNDO.
oUser =
    (NEW UserBuilder("Michael", "Barfs")
  :setAge(23)
  :setPhone("+49 40-30 68 03-26")
  :setAddress("Valentinskamp 30, 20355 Hamburg")
  :build()).
  :build()).
```

This structure is called chaining. By the way, the created UserBuilder object is thrown away immediately and will hopefully be removed by the garbage collector.

Advantages of Builder Pattern:

- Readable – the property is named by the setter method
- Less confusion with parameters – they are named
- Auto-complete suggests property methods
- Shorter than setting properties in user object
- Simple to change the base object without changing existing calls

To make the advantages clearer, here is an example of a procedural code with a lot of parameters.

The code is a shortened version of the full call.

```
RUN StatusCreate IN l-Import-Library-Handle
  ( INPUT l-DB-Cust,
    INPUT "",
    INPUT 150,
   ...
  INPUT
    "QtyType=" + OrderQtyQualifier + "{&T}"
    + "UTCTime=" + l-UTCTime
    + "{&T}"
    + "ConC-ID=" + SSCO-Ord.ConC-ID
  ...
  ) NO-ERROR.
  ) NO-ERROR.
```

The parameters are in a given order (like a very large constructor). If a parameter is added, removed or changed, every existing calls needs to be changed, too.

The programmers were so desperate, that they added a universal parameter (List of key=value{Delimiter}key=value...). That was done to allow changing the parameters without changing existing code. These kinds of lists are very good as a short term fix, but bad in long-term maintenance.

So that is was working solution, but not a good one.

Here is the approach with the Builder Pattern (excerpt).

```
DEFINE VARIABLE oStatusAnlage AS StatusAnlage NO-UNDO.

oStatusAnlage =
(NEW StatusAnlageBuilder()
:setCustCode(l-DB-Cust)
:setStatusNumeric(150)
...
:setQtyType(OrderQtyQualifier)
:setUTCTime(l-UTCTime)
:setConCID(SSCO-Ord.ConC-ID)
...
:build()).
```

The code is much more readable and you can add parameters without changing existing code.

Conclusion:

The Builder Pattern will make the code more readable, better working due to auto-complete and allows changes later in a project.

Pattern 2 – Singleton (Creational Pattern)

The Singleton Pattern is a Creational Pattern, it is a substitution for global objects in OO.

There are no global objects in OO languages, but many people see the need for kinds of global structures, e.g. read only setup information.

Some people disagree with the idea of the Singleton Pattern, since they see it as an Anti Pattern. Here is a good discussion about the pros and cons of using Singleton and what to consider.

If you use a Singleton for read-only content and Dependency Injection for dependent state information, it should be OK.

The Singleton is using the effect, that static elements (object, properties) are kind of global. They are instantiated during first access and they exist as long as the session is running. They can not be deleted.

Here is the base code.

```
CLASS Configuration:
...
    DEFINE PUBLIC STATIC PROPERTY oInstance AS Configuration
        PUBLIC GET():
            IF NOT VALID-OBJECT(oInstance) THEN
                oInstance = NEW Configuration().
            RETURN oInstance.
        END GET.
        PRIVATE SET.


    CONSTRUCTOR PRIVATE Configuration():
        loadConfig().
    END CONSTRUCTOR.
...
END CLASS.
```

Note the PUBLIC STATIC property and the PRIVATE constructor.

Note also that the configuration object is created during the first call of the object instance (oInstance).

Here is the usage of the configuration object.

```
DEFINE VARIABLE oConf           AS Configuration NO-UNDO.
DEFINE VARIABLE cBaseExportPath AS CHARACTER     NO-UNDO.

oConf = Configuration:oInstance.

cBaseExportPath = oConf :getValue("BaseExportPath")
...
```

Advantages of Singleton Pattern:

- Solves problem of global settings
- Inheritance is possible (which is not possible from a static object)
- Has some logic during instantiation
- Can be re-instantiated (which is not possible with a pure static object)

Conclusion:

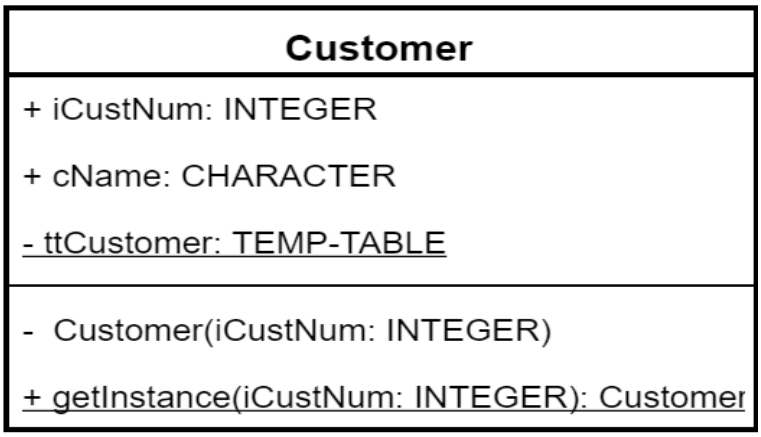
The Singleton Pattern will make kinds of global settings available.Programmers must care about dependencies! It would be best to use Singleton for read only and Dependency Injection for a local active program state.

Pattern 3 – Multiton (Creational Pattern)

The Multiton Pattern is a Creational Pattern, like a bunch of Singletons. It is using a static element for singular data access.

The Multiton returns an object based on an index value. It will guarantee that every call using this index value will return the same object. This is useful, when you have multiple run time objects that are accessing one piece of data.

Here is the object structure.



Sample code of the declaration part. Note that the temp-table is static (global).

```
DEFINE PUBLIC PROPERTY iCustNum AS INTEGER NO-UNDO GET.
    PRIVATE SET.
DEFINE PUBLIC PROPERTY cName AS CHARACTER NO-UNDO GET.
    PRIVATE SET.
DEFINE PRIVATE STATIC TEMP-TABLE ttCustomer
    FIELD custNum AS INTEGER
    FIELD obj AS Progress.Lang.ObjectINDEX ID custNum.....
END CLASS.
```

The access method is public, but static.

When an instance is asked for an entry, an existing object or new object will be returned.

The reference in the temp-table is permanent, so garbage collection will not happen.

```
METHOD PUBLIC STATIC Customer getInstance(iCustNum AS INTEGER):
    FIND FIRST ttCustomer WHERE ttCustomer.custNum = iCustNum NO-LOCK NO-ERROR.
    IF NOT AVAILABLE ttCustomer THEN
    DO:
        CREATE ttCustomer.
        ASSIGN
            ttCustomer.custNum = iCustNum
            ttCustomer.obj     = NEW Customer(iCustNum)
            .
    END.RETURN CAST(ttCustomer.obj, Customer).
END METHOD.
```

The constructor is private, it will be called from the code above and not from the code using the Multiton.

This sample code is loading data from the database.

```
CONSTRUCTOR PRIVATE Customer(iCustNum AS INTEGER):
    DEFINE BUFFER bCustomer FOR Customer.
    FIND FIRST bCustomer WHERE bCustomer.CustNum = iCustNum NO-LOCK NO-ERROR.
    IF AVAILABLE bCustomer THEN DO:
        THIS-OBJECT:cName = bCustomer.Name.
        THIS-OBJECT:iCustNum = bCustomer.CustNum.
    END.
END CONSTRUCTOR.
```

Finally, here is the usage.

We give a customer number to the Multiton and receive a customer object.

```
DEFINE VARIABLE oCust AS Customer NO-UNDO.
    oCust = multiton.Customer:getInstance(1537).
```

Remarks:

- The basic idea of Multiton is to have thread safe objects
  (which is at the moment not important for OOABL)
- The objects are like global, so they accumulate
  (the programmer may delete them)
- Creating/Accessing 10000 objects is OK, 100000 is not (much slower)
  (Performance test in OE 11.7x)

Disadvantages

- Vulnerable to side effects, e.g. like global shared buffer
- Unit tests becomes more complex
- OOABL performance problems with lots of objects

The Multiton can be used without any considerations for reading data, like in reports or display only.

Conclusion:

The Multiton Pattern will allow program-wide access to unique resources. This could be streams, WebServices, ESB access, DB connections and data.

But for updating data, the programmer is highly responsible to make sure to avoid side effects! And the OOABL may be too slow for large numbers of instantiated objects.

Pattern 4 – Lazy Loading (Creational Pattern)

The Lazy Loading Pattern is a Creational Pattern, it delays the instantiation of a class, the search for data or calculation of value until the moment, it is used. The opposite of Lazy Loading is Eager Loading.

Often, data and objects may be used or not, depending on the program flow. Lazy Loading delays the load until the moment, an object (or data) will be used. Lazy Loading is something you may have used in OpenEdge for years, without knowing it is a pattern ;)

Lazy loading has four main variants:

- Lazy Initialization – Delay getting data for a field until it is needed
- Virtual proxy – Delay instantiating an object until it is needed. This may also increase security, as the proxy must not expose all methods and properties.
- Ghost – Here an object will held only partial information (e.g. id of record). When record data are accessed, missing data will be loaded. Ghost makes start up faster.
- Value Holder – Object with a get value method, getting data on demand from real object.

Popular Lazy Loading scenarios:

- Access aggregated data
- Load images / data when they appear in view (infinite scroll)
- Tab widget is selected
- Initialize a service (ESB, log system, rpc…) when first used

In this OpenEdge example (Lazy Initialization) we calculate the invoice summary of the last three month for a customer.

```
CLASS cMyInvoice:
...
    DEFINE PUBLIC PROPERTY iInvSum AS INTEGER NO-UNDO INITIAL ?
        PRIVATE SET.

        PUBLIC GET:
            IF iInvSum = ? THEN
            DO:
                DEFINE VARIABLE iCn AS INTEGER NO-UNDO.
                iCN = THIS-OBJECT:iCustNum.
        //loop through invoices of customer
        //accumulate invoices ...
            END.
            RETURN iInvSum.
        END GET.
...
END CLASS.
```

Lazy Loading – Advantages

- Delaying or avoiding work
- Starting screen faster

Lazy Loading – Disadvantages

- Extracting code
- May increase overall calls to DB
- May show inconsistent data
- (cause some parts are loaded later, first loaded parts may not refreshed)

Conclusion:

In OpenEdge daily work Lazy Loading is most useful for service level objects (logging, ESB) as well as giving the user fast screen load and 'infinite scroll' (data browse, tabs).

When using kind of 'Business Objects' (collecting data on AppServer and sending a bunch of data via DataSet or Temp-Table) delayed load of data can be too expensive. Later calls are time consuming.

Pattern 5 – Adapter (Structural Pattern)

The Adapter Pattern is a Structural Pattern and also know as 'wrapper'. It converts an interface into another interface, to make the interfaces compatible, wrap an old class or change components.

The adapter is often used in a project, when something changed and you need to make the existing system running with a new subsystem. Examples for this are new scanning hardware, switching from Google Maps to OpenStreetMap, switching from Sonic ESB to Apache ESB, using Business Entities from multiple vendors...

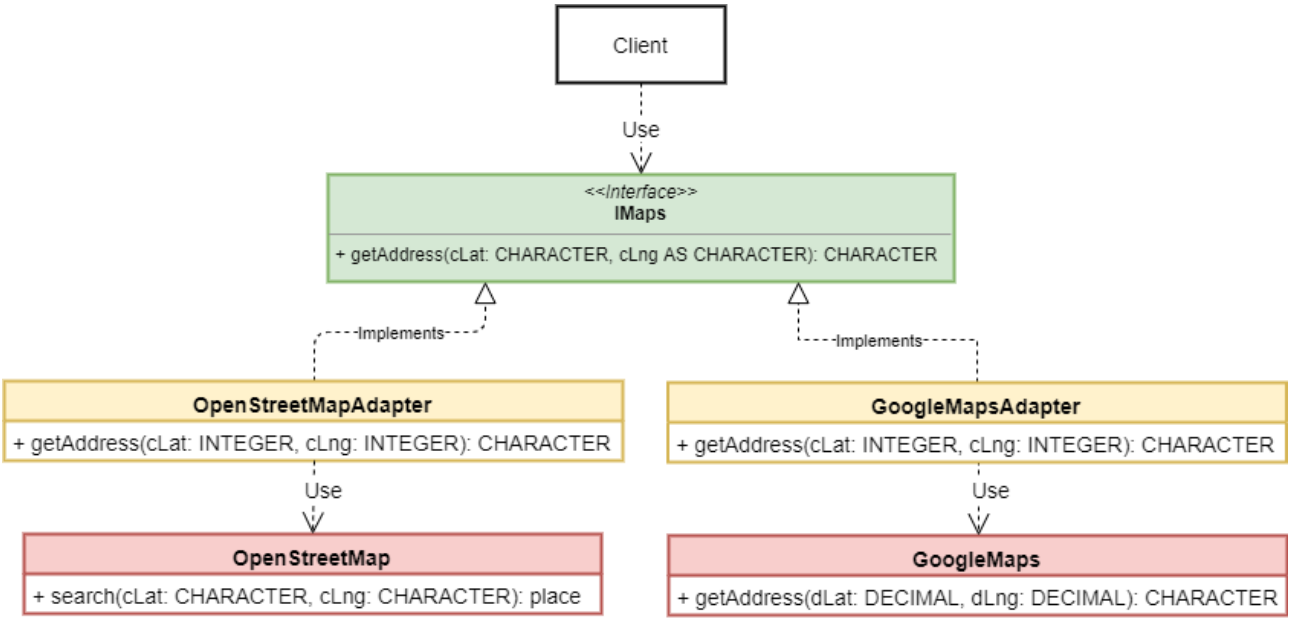
Advantages

- Allowing change of subsystems/libraries
- Reusing objects
- Creating abstract application tiers (e.g. use adapter for Business Entities)
- Adopting 3rd party objects
- Making usage simpler (more convenient)
- (hide parts of API or adapt to project style. Look also for facade pattern.)

Disadvantages

- More code
- Small run-time overhead

The OO feature of interfaces is a static feature but it allows to use various objects without dynamic code. This diagram shows classes included in a use case, where the client can use Google Maps or OpenStreetMap.



Level 1 Interface class: It defines every method of the adapters.

```
INTERFACE pattern_example.adapter2.IMaps:

  METHOD PUBLIC CHARACTER GetAddress (dLat AS DECIMAL, dLng AS DECIMAL).

    //...other method declarations

END INTERFACE.
```

Level 2 – Adapters: Next level are the adapters, in this case the adapter to Google.

```
CLASS pattern_example.adapter2.GoogleMapsAdapter IMPLEMENTS IMaps:
    CONSTRUCTOR PUBLIC GoogleMapsAdapter ():
        // Create session connect to Google Maps.
    END CONSTRUCTOR.

    METHOD PUBLIC CHARACTER GetAddress(INPUT dLat AS DEC, INPUT dLng AS DEC):
        // Call method in Google Maps...
        RETURN "The address found by GOOGLE".
    END METHOD.
    // ...other
END CLASS.
```

Level 3 – The client: The constructor loads one of the adapters (Google, OSM). The call (mGetAdrFromCoord) will call the method in the adapter, it does not matter which one.

```
CONSTRUCTOR PUBLIC MyGeoApi ( INPUT cSearchTool AS CHARACTER ):
    DEFINE VARIABLE oMapApi AS pattern_example.adapter2.IMaps.

    CASE cSearchTool:
        WHEN "OSM" THEN
            DO:
                oMapApi = NEW OpenMapsAdapter().
            END.

        // Google is default

        OTHERWISE
        DO:
            oMapApi = NEW GoogleMapsAdapter().
        END.
    END.
END CONSTRUCTOR.

METHOD PUBLIC CHARACTER mGetAdrFromCoord (dLat AS DECIMAL, dLng AS DECIMAL):
    RETURN oMapApi:GetAddress(dLat, dLng).
END METHOD.
```

It is easy to see that multiple adapters can exist, hiding real calls and call interfaces.

Conclusion:

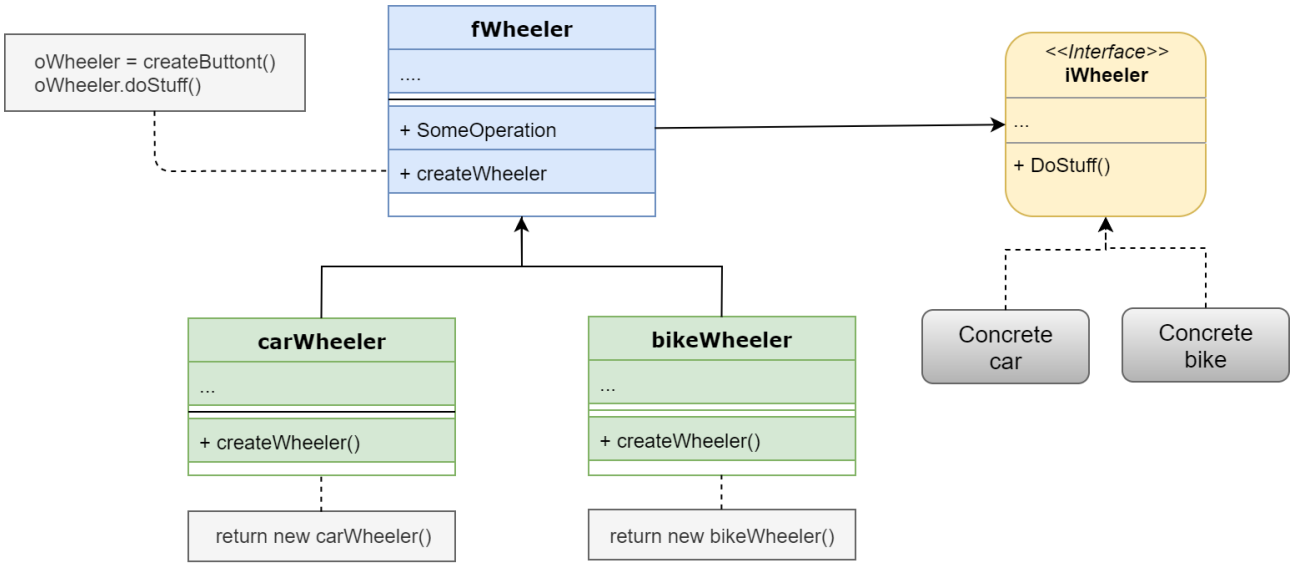
The idea of an adapter is very common. It is a practical approach to simplify API/interface access. It was used in OpenEdge prior to OO, but OO makes it much more elegant.

Pattern 6 – Factory/Fabric (Creational Pattern)

The Factory Pattern is a Creational Pattern, that allows to create an object without specifying the exact class that will be created.

Sometimes an application will create objects, which are similar but not exact the same..

To make the code more universal, the factory add a layer to create these classes. You order something in the same area of interest and the fabric is producing this on demand, that is pretty simple :)



Fabric Pattern – Class diagram

The creation specific logic and the details are hidden behind the factory. During run-time the created objects are accessed through an interface.

So the object is abstract during creation and during use. This diagram shows an example, where a factory is used to create transports with a wheel (car, bike).

In my example I have a factory creating some wheeler (car, bike) classes. I will not show the interface here, this is simple. It will be in the source package for download.

fWheeler.cls – Factory class: Instantiate class depending on wheeler type. The factory gives back the instance of a wheeler object of type iWheeler (the interface).

```
CLASS 06_Factory.fWheeler:

  ... conctructor etc.

  // Create a car, bike or what else...
    METHOD PUBLIC iWheeler createWheeler ( cWheelerType AS CHARACTER ):
        DEFINE VARIABLE oWheeler AS iWheeler NO-UNDO.

        /* Instantiate various object types (which implement wheeler interface) */
        CASE cWheelerType:
            WHEN "Car"  THEN
                oWheeler = NEW carWheeler().
            WHEN "Bike" THEN
                oWheeler = NEW bikeWheeler().
            OTHERWISE
            DO:
                MESSAGE "Uups, unknown wheeler type: " cWheelerType
                    VIEW-AS ALERT-BOX INFORMATION.
                RETURN ERROR.
            END.
        END CASE.

    // Start the wheeler building process
        oWheeler:buildWheeler().
        RETURN oWheeler.
    END METHOD.
...
```

bikeWheeler.cls – Implementation of bike wheeler: Create bike with common stuff and some bike specific methods. A similar class for cars is existing.

```
CLASS 06_Factory.bikeWheeler IMPLEMENTS 06_Factory.iWheeler:

  … properties from interface

    METHOD PUBLIC VOID buildWheeler(  ):
    // Call some bike specific stuff
        addPedal().
        cBuildResult = "I build a bike".
        cColor = "green".
        cType  = "bike".
    END METHOD.

    METHOD PUBLIC CHARACTER driveTest( INPUT cLevel AS CHARACTER ):
    // Run various tests somewhere in the area
        RETURN "Did bike test drive on level: " + cLevel.
    END METHOD.

    METHOD PUBLIC LOGICAL addPedal(  ):
    // Add a pair of glowing pedals to the bike
        cFeatures = "Professional races pedals".
        RETURN TRUE.
    END METHOD.
...
```

start.p – test program: This program (a .p file for my convenience) is using the factory for creating two wheelers. The handling of the different wheeler types is very simple for the user.

```
DEFINE VARIABLE ofWheeler AS fWheeler NO-UNDO.
DEFINE VARIABLE oWheeler1 AS iWheeler NO-UNDO.
DEFINE VARIABLE oWheeler2 AS iWheeler NO-UNDO.

// First, create the factory
ofWheeler = NEW fWheeler().
// Then create two wheelers
oWheeler1  = ofWheeler:createWheeler("bike").
oWheeler2  = ofWheeler:createWheeler("car").


// Get some information from the wheelers
MESSAGE "Result of building wheeler: " oWheeler1:cBuildResult SKIP
        … other
    "Result of building wheeler: " oWheeler2:cBuildResult SKIP
        … other
    VIEW-AS ALERT-BOX INFORMATION TITLE"Build wheeler result"
```

Factory – Advantages

- Loose coupling between creator and created objects
- Same creation code for every case (UI in this example)
- Extensible, adding new UI types is simple
- Testing (mock) is simple
- Increase abstraction level (reduce maintenance)

Factory – Disadvantages

- Add some complexity and code

Abstract Fabric:
A fabric is a class, creating objects of same or similar type. Often a fabric is used to create fabrics, then we talk about the Abstract Fabric pattern.

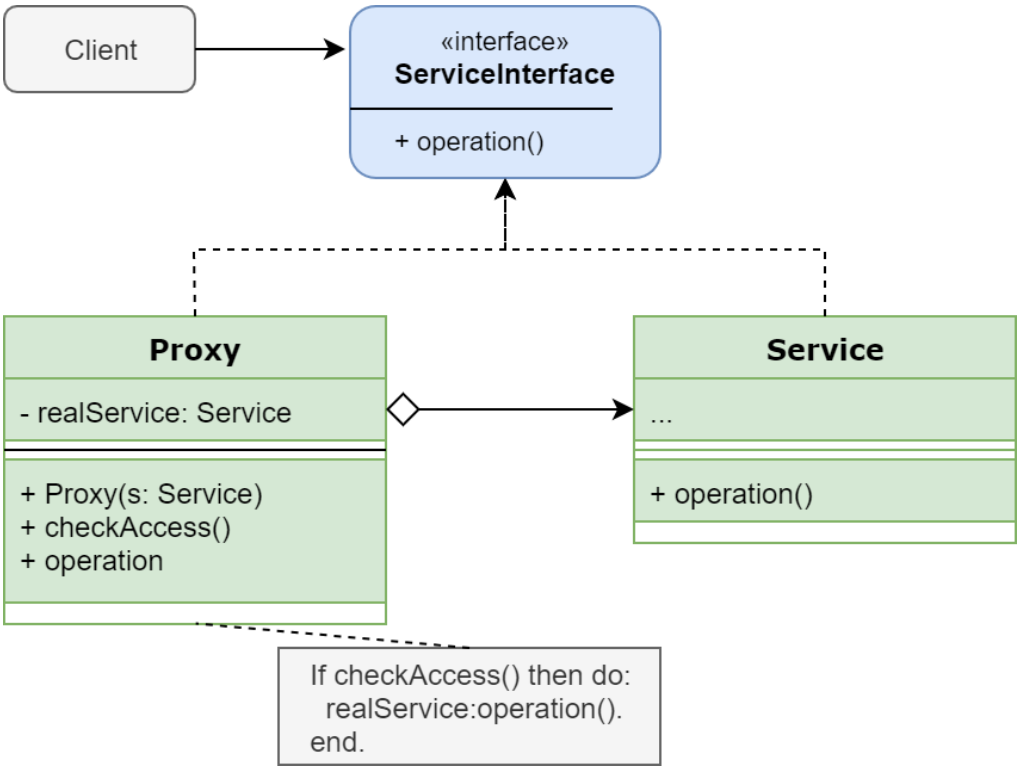
Conclusion:

Abstract Fabric is one of the top used design pattern, and useable without any problems in OpenEdge.

Pattern 7 – Proxy (Structural Pattern)

The Proxy Pattern is a Structural Pattern, which will use one object as proxy for a second object, to mask something (complexity, security, location, instantiation time...) in another object.

The proxy is related to Adapter and Decorator (which I did not talk about in this series). The proxy is different to these, as it implements the same interface as the service object).



Proxy Pattern – Class diagram

There are several scenarios, where proxies are used:

- Smart proxy – Control load/unload/use of a central object or handle locking for a given to a unique resources object (e.g. wait until a transfer is ready before starting next).
- Protective proxy – Control access to an object (security).
- Remote proxy – Have a local object controlling a remote version of the same object (RMI).
- Virtual proxy – Delay or avoid creation of a resource hungry (time, resources) object, so that the real creation is done during first functional access.
- Simplify API to access a complex object.
- Cache or log requests
- Proxy Pattern – Class diagram

proxySoapService.cls – Proxy class: This proxy create an instance of the SOAP class. It will also check rights before and force HTTPS.

```
CLASS 07_Proxy.proxySoapService:

  … define interface properties

    DEFINE VARIABLE oSoapServices AS 07_Proxy.soapService NO-UNDO.

  // Create the service object and force https
    CONSTRUCTOR PUBLIC proxySoapService ():
        IF testUserRights() = TRUE THEN
        DO:
            oSoapServices = NEW 07_Proxy.soapService().
            oSoapServices:cProtocol = "SOAP".
            oSoapServices:lUseHttps = TRUE.
        END.
        ELSE RETURN ERROR.
    END CONSTRUCTOR.

  // Connect target URL
    METHOD PUBLIC VOID connectURL ( ).
    oSoapServices:cURL = THIS-OBJECT:cURL.
    IF oSoapServices:connectURL() THEN
    DO:
        cConnectResult = "Everything OK, connected : " + THIS-OBJECT:cURL.
    END.
    ELSE RETURN ERROR.
    END METHOD.

    METHOD PRIVATE LOGICAL testUserRights ():
    // Do some sophisticated right checks
        RETURN TRUE.
    END.
END CLASS.
```

SoapService.cls – Service class: This class handles the SOAP protocol level. It has no idea of security (user rights and https).

```
CLASS 07_Proxy.soapService IMPLEMENTS 07_Proxy.iSoapService:

  … define interface properties

    DEFINE PUBLIC PROPERTY lUseHttps       AS LOGICAL   NO-UNDO GET. SET.


  // Connect target URL
    METHOD PUBLIC LOGICAL connectURL ( ).
    /* Use best method to connect remote SOAP URL... */
    RETURN TRUE.
END METHOD.

  // other functionality here... send, receive, decode...

END CLASS.
```

start.p – test program: Quite simple test program to start the proxy and initiate a call.

```
DEFINE VARIABLE oMySoapService AS proxySoapService NO-UNDO.

oMySoapService = NEW proxySoapService().

oMySoapService:cURL = "https://my.service.com".
oMySoapService:connectURL().

MESSAGE oMySoapService:cConnectResult
    VIEW-AS ALERT-BOX INFORMATION.
```

Proxy – Advantages

- Add control and security
- Avoid duplication of 'huge' objects and manage their life cycle
- Allow handling of remote objects

Proxy – Disadvantages

- Additional complexity
- RMI calls are expensive/slow

Conclusion:

Most of the numerous use cases for the proxy pattern can be used with the OpenEdge OO ABL. The OOABL has no direct RMI capabilities build in. As a workaround use AppServer calls.

OO Design Pattern In OOABL – Conclusion

The language of the OpenEdge platform (called ABL or 4GL) is Today a full fledged OO programming language.
Pattern are helpful concepts how to organize common scenarios in a project. When used in the same style in a project they became part of the 'documentation', because they make things standardized.

Some general points:

- Pattern are useful for organisation, documentation and maintenance
- For small projects pattern do not help really
- Some patterns are similar to others
- The team should choose a set of patterns for a project
- Be careful, a pattern could mutate to an anti-pattern
  (e.g. use Singleton for state transfer between objects)
- Using a pattern just for using a pattern is more worse then never use them.

Some OO ABL specific points:

- More or less all patterns will technically work
- OO speed in ABL is lower than in other languages, so create large swarms of objects may be slower than expected
  (Progress is working on this)

A design pattern is a defined way to solve a problem. Not more and not less. They are a way to keep a project on the road and to give the team a common idea of solutions. If something has a name, you can talk about it.

If you program in the OO ABL, find your pattern and start using them!

Klaus Erichsen
Hamburg, May 2019

Loosely based on the work of Michael Barfs.

The articles are availabe here on line:
https://www.iap.de/design-pattern-in-oo-abl-part-1-of-3/
https://www.iap.de/design-pattern-in-oo-abl-part-2/
https://www.iap.de/design-pattern-in-oo-abl-part-3/

Contact me: ke@iap.de

Sample source code for the pattern is available for Download.

Any code is given as example and "as is". Usage is allowed, but IAP is not responsible for the outcome.