# Using the Factory Pattern in OOABL: How, when and why

# Lutz Fechner

- Project Consultant at Consultingwerk
  Project- and Delivery Management
  „Special“ projects / non ABL

- 20+ years experience in C# (Framework and Core), Java, C++, C, JavaScript, ABL

- Full Stack Development
  eProcurement Systems
  Catalog Search Engines

# Consultingwerk Software Services Ltd.


- Independent IT consulting organization
- Focusing on **OpenEdge** and **related technology**
- Located in Cologne, Germany, subsidiaries in UK, USA and Romania
- Customers in Europe, North America, Australia and South Africa
- Vendor of developer tools and consulting services
- Specialized in GUI for .NET, Angular, OO, Software Architecture, Application Integration
- Experts in OpenEdge Application Modernization

# Services Portfolio, Progress Software

- OpenEdge (ABL, Developer Tools, Database, PASOE, …)
- Telerik DevCraft (.NET, Kendo UI, Angular, …), Telerik Reporting
- OpenEdge UltraControls (Infragistics .NET)
- Telerik Sitefinity CMS (incl. integration with OpenEdge applications)
- Kinvey Plattform, NativeScript
- Corticon BRMS
- Whatsup Gold infrastructure-, network- and application monitoring
- Kemp Loadmaster
- …

# Services Portfolio, related products

- Protop Database Monitoring
- Combit List & Label
- Web frameworks, e.g. Angular
- .NET
- Java
- ElasticSearch, Lucene
- Amazon AWS, Azure
- DevOps, Docker, Jenkins, ANT, Gradle, JIRA, …
- …

# Recap from yesterday

- Patterns in general
- Decorator pattern

# Software Design Patterns

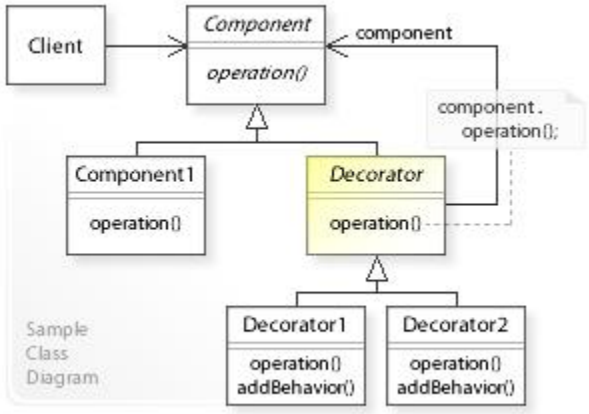
- Well known “ways of doing”, solving common, reoccurring problems
- Easier to understand und maintain clean code
- Prevents reinventing the wheel and “too creative” code

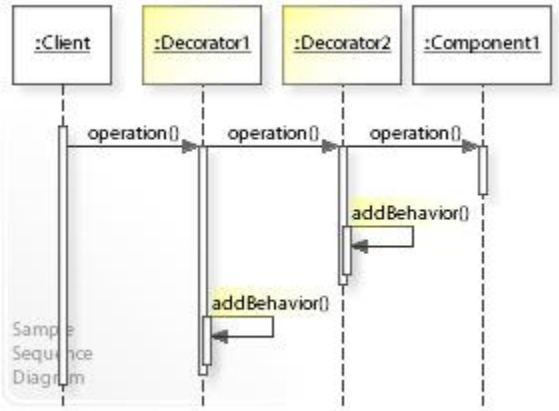
# Decorator pattern

- Allows functionality to be divided by concern (Single Responsibility)
- Allows extension without modification (Open Closed Principle)
  - This is the actual decoration
- Flexible, efficient way of extending an object without creating a new object
  - No Casting, Extending or Overwrites needed

# Decorator pattern

- Interface, Decorator(s), Decorated
- Decorator implementes Interface of the to be decorated
- Decorator holds reference to the decorated object (Wrapper)



# Software Design Patterns

- Popular through the GoF (Gang of Four)
  - Erich Gamma (IBM/Rational/Microsoft – Developer of Eclipse, Junit and VS Code)
  - Richard Helm (IBM/Boston Consulting)
  - Ralph Johnson (worked on Smalltalk)
  - John Vlissides (IBM)

- Examples: Factory, Builder, Singleton, Facade, Adapter, Iterator, Lazy Initialization, and many more….

# Agenda

- Example
- Factory patterns
- Fluent Interface
- Examples

# Example

- We want to create houses
- With different capabilities
  - With Insulation
  - With Solar
  - With Heat Pump
- A house may have one or more of these capabilities
  - Capabilities can be upgraded over the lifetime of a house
  - Different houses have different features in place

# How do we specify the capabilities?

- Constructor arguments
  - Does allow required values to be set
  - Optional values may be set
  - Can end up with vary many constructors, with very many parameter combinations
  - Can end up with overly-broad constructors, with too many parameters for the required capabilities

***Which constructor is a developer supposed to call?***

- Settable properties, public methods
  - Caller must somehow know that they are supposed to call these

# Bad Example

```
oHouse = NEW ClassWithUglyConstructor(FALSE, FALSE, "", 5, 12.0, 6, TRUE, TRUE, 5, 6, "WTF", FALSE, FALSE).
```

- Constructor arguments
  - Not really comprehensive (What does the values given mean, and why?)
    - Intellisense the parameters to at lease see what they might mean.
  - Need a parameter more for some processing inside the class?
    - New Constructor with meaningful default values
    - Change all code pieces that used the old one

# Introducing Factories & Builders

- Abstract factory Provide an interface for creating *families* of related or dependent objects without specifying their concrete classes
- Builder Separate the construction of a complex object from its representation, allowing the same construction process to create various representations
- Factory method Define an interface for creating a *single* object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses

https://en.wikipedia.org/wiki/Abstract_factory_pattern

https://en.wikipedia.org/wiki/Builder_pattern

https://en.wikipedia.org/wiki/Factory_method_pattern

# Factories & builders

```
CLASS Consultingwerk.Demo.Factory.HouseBuilder
    IMPLEMENTS IHouseBuilder ABSTRACT
    :

    DEFINE PUBLIC PROPERTY House AS IHouse NO-UNDO
    GET():
        RETURN THIS-OBJECT:GetInstance().
    END GET.

    METHOD ABSTRACT PROTECTED IHouse GetInstance().

    METHOD STATIC PUBLIC IHouseBuilder Build (pcHouseCategory AS CHARACTER):
        IF pcHouseCategory = "modern" THEN
            RETURN NEW ModernHouseBuilder().
        ELSE
            RETURN NEW DefaultHouseBuilder().
    END METHOD.

END CLASS.
```

Abstract Factory

Factory method

Builders

# Factories & builders

- Developer doesn’t need to know specific type that is created
- Factory might have more meaningful method names
  - A Constructor is always called like the class name
- You can change the Factory providing new implementations without having to change the client code
- Abstract Factory is kind of magic black box. You don’t know who builds you what specific type.

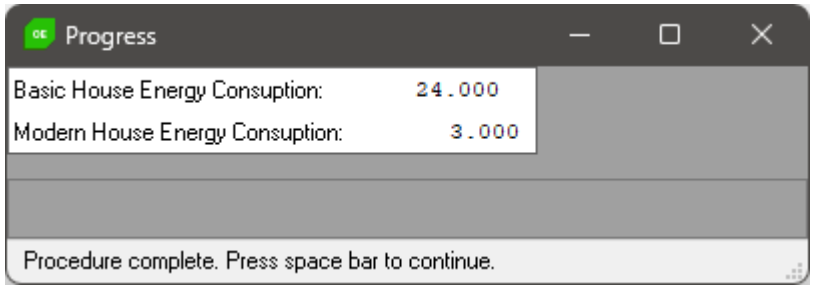
# Abstract Factory usage

```
DEFINE VARIABLE oHouse AS IHouse NO-UNDO.

/* ************************  Main Block  ************************ */

oHouse = HouseBuilder:Build(""):House. //you don't see who build the object for you
DISPLAY "Basic House Energy Consuption: " oHouse:GetEndEnergyConsumption().

oHouse = HouseBuilder:Build("modern"):House. //you don't see who build the object for you
DISPLAY "Modern House Energy Consuption: " oHouse:GetEndEnergyConsumption().
```



# The Builder builds the object for you

```
CLASS Consultingwerk.Demo.Factory.DefaultHouseBuilder INHERITS HouseBuilder:
    METHOD PROTECTED OVERRIDE IHouse GetInstance():
         DEFINE VARIABLE oHouse AS IHouse NO-UNDO.
            oHouse = new BasicHouse().
         RETURN oHouse.
     END METHOD.
END CLASS.
```

```
CLASS Consultingwerk.Demo.Factory.ModernHouseBuilder INHERITS HouseBuilder:

    METHOD PROTECTED OVERRIDE IHouse GetInstance():
        DEFINE VARIABLE oHouse AS IHouse NO-UNDO.
           // create a house
           oHouse = new BasicHouse().
           //decorate it
           oHouse = new InsulatedHouse(oHouse).
           oHouse = new HeatPumpHouse(oHouse).
           oHouse = new SolarPoweredHouse(oHouse).
        RETURN oHouse.
    END METHOD.

END CLASS.
```

# Classical Builders

- Help to create complex objects without long parameter lists in telescopic constructors ( see the bad example )

- Create object in steps:

```
oHouseBuilder = NEW ClassicalHouseBuilder().

oHouseBuilder:AddInsulation(TRUE).
oHouseBuilder:AddHeatPump(FALSE).
oHouseBuilder:AddSolar(TRUE).

oHouse = oHouseBuilder:Build().
```

- Doesn’t fit all use cases (Not even the one in our example here ☺ )
- Don’t use for immutable objects (Can’t be changed at runtime)

# Fluent interface

A fluent interface is an object-oriented API whose design relies extensively on method chaining. Its goal is to increase code legibility by creating a domain-specific language (DSL).

```
using OpenEdge.Net.HTTP.*.

define variable oRequest as IHttpRequest no-undo.

oRequest = RequestBuilder:Post("https://example.com/", oJsonData )
                          :ContentType( "application/json" )
                          :AcceptJson()
                          :SetHeader("X-API-Key", "abc123")
                          :Request.
```

https://en.wikipedia.org/wiki/Fluent_interface

# Fluent House Builder

This is the trick!

```
INTERFACE Consultingwerk.Demo.Fluent.IFluentHouseBuilder:

    METHOD PUBLIC IFluentHouseBuilder AddInsulation(plInsulation AS LOGICAL).

    METHOD PUBLIC IFluentHouseBuilder AddHeatPump(plHeatPump AS LOGICAL).

    METHOD PUBLIC IFluentHouseBuilder AddSolar(plSolar AS LOGICAL).

    DEFINE PUBLIC PROPERTY House AS IHouse NO-UNDO
    GET.

END INTERFACE.
```

# Fluent interface: example

```
DEFINE VARIABLE oHouse AS IHouse.

//basic house
oHouse = FluentHouseBuilder:Build()
                           :House.

DISPLAY "Basic House energy consumption:" oHouse:GetEndEnergyConsumption() SKIP.


//insulated house
oHouse = FluentHouseBuilder:Build()
                           :AddInsulation(TRUE)
                           :House.
DISPLAY "Insualted House energy consumption:" oHouse:GetEndEnergyConsumption() SKIP.


//insulated house with heat pump
oHouse = FluentHouseBuilder:Build()
                           :AddInsulation(TRUE)
                           :AddHeatPump(TRUE)
                           :House.
DISPLAY "After heat pump upgrade:" oHouse:GetEndEnergyConsumption() SKIP.

//insulated house with heat pump and solar
oHouse = FluentHouseBuilder:Build()
                           :AddInsulation(TRUE)
                           :AddHeatPump(TRUE)
                           :AddSolar(TRUE)
                           :House.
DISPLAY "After solar installation:" oHouse:GetEndEnergyConsumption() SKIP.
```

Configure your object as needed.

Use variables instead of hard-coded TRUE to dynamically build the right thing for you.

# Conclusion

- Never write a NEW again! Factories and builders give us a single-responsibility class for instantiating objects

- Application developers don't need to think about any complexities of constructing objects

# Additional info

- More examples are available at https://github.com/4gl-fanatics/airplane-seat-patterns


lutz.fechner@consultingwerk.de

- We have a OpenEdge quiz running at our booth or available at consultingwerk.com/quiz

- Callenge your OpenEdge knowledge to earn a chance to win an Amazon Echo Studio, Echo or Echo Spot

- Best entries will be presented on Friday here at the PUG!