Consultingwerk
software architecture and development

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

Peter Judge

- Senior Architect at Consultingwerk
- Writing 4GL since 1996, working on a variety of frameworks and applications. More recently have worked on a lot of integration-y stuff: Authentication Gateway, HTTP Client, Web Handlers. Dabble in PASOE migrations.
- Active participator in Progress communities, PUGs and other events

© 2023 Consultingwerk Software Services Ltd. All rights reserved.

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
- …

© 2023 Consultingwerk Software Services Ltd. All rights reserved.

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, …
- …

© 2023 Consultingwerk Software Services Ltd. All rights reserved.

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

© 2023 Consultingwerk Software Services Ltd. All rights reserved.

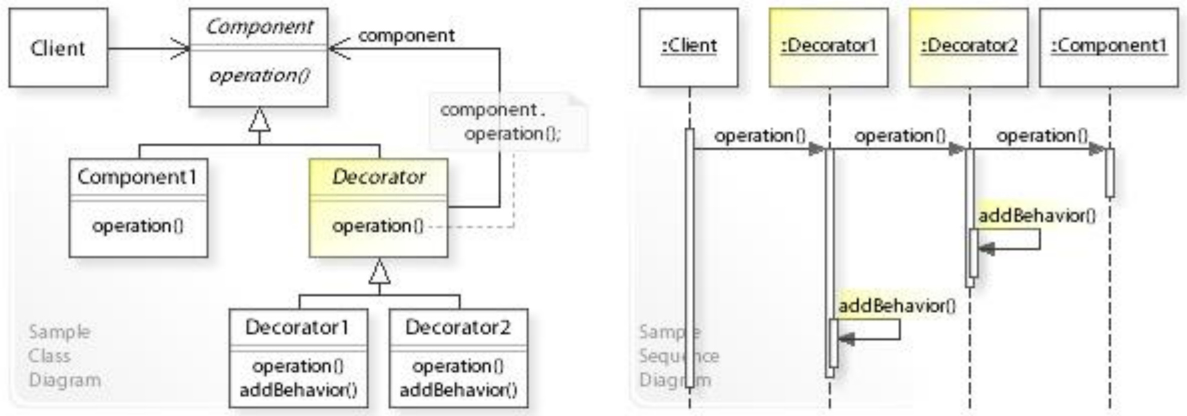
Decorator pattern

- 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

© 2023 Consultingwerk Software Services Ltd. All rights reserved.

Decorator pattern

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



© 2023 Consultingwerk Software Services Ltd. All rights reserved.

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….

© 2023 Consultingwerk Software Services Ltd. All rights reserved.

Agenda

- Example
- Factory patterns
- Fluent Interface
- Examples

© 2023 Consultingwerk Software Services Ltd. All rights reserved.

Example

- We want to a class to represent a House
  …and want to know how much Energy it consumes over the year
  - How does that change if we change something on the house?
  - We want to change that at Runtime!
    - Not at compile time

- Houses may have solar panels, insulation, a battery, heat pumps, etc
  - Not all houses have all of these
  - Some houses may have multiple
  - Capabilities can be upgraded over the lifetime of a house

© 2023 Consultingwerk Software Services Ltd. All rights reserved.

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

© 2023 Consultingwerk Software Services Ltd. All rights reserved.

Bad Example

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

- Constructor arguments
  - Not really comprehensive (What do the values given mean, and why?)
  - Intellisense or documentation can help decipher their purpose
- 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

© 2023 Consultingwerk Software Services Ltd. All rights reserved.

The Old MacDonald approach

… A new-new here, a new-new there, here a new, there a new, everywhere a new-new …

- What happens if you need to add mandatory data to the class?
  - Use sensible defaults
  - New subtype
- Typically results in changes to existing NEWs

  You have how many?

There should be only one place responsible for the creation of object for a type or family of types

© 2023 Consultingwerk Software Services Ltd. All rights reserved.

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

© 2023 Consultingwerk Software Services Ltd. All rights reserved.

Factories & builders

```
class HouseBuilder abstract implements IHouseBuilder :

  define public property House as IHouse no-undo
  get():
    /* Abstract method, supports overriding pre-12.5 */
    return this-object:GetInstance().
  end get.
  method abstract protected IHouse GetInstance().

  method static public IHouseBuilder Build (pcCategory as character):
    case pcCategory:
      when "modern" then return new ModernHouseBuilder().
      when "basic"  then return new BasicHouseBuilder().
      otherwise          return new DefaultHouseBuilder().
    end case.
  end method.
```

Abstract factory

Factory method

Concrete builders

Abstract Factory

```
interface IHouseBuilder:

    define public property House as IHouse no-undo
    get.

    method public void AddInsulation(plInsulation as logical).

    method public void AddHeatPump(plHeatPump as logical).

    method public void AddSolar(plSolar as logical).

end interface.
```

Specify desired capabilities

© 2023 Consultingwerk Software Services Ltd. All rights reserved.

Abstract factory : example

```
class HouseBuilder abstract implements IHouseBuilder :

    /* Removed Factory method, static Build() method to fit on the slide :) */

    define protected variable lHasHeatPump as logical no-undo.
    define protected variable lHasInsulation as logical no-undo.
    define protected variable lHasSolar as logical no-undo.

    method public void AddHeatPump( input plHeatPump as logical ):

        lHasHeatPump = plHeatPump.

    end method.

    method public void AddInsulation( input plInsulation as logical ):

        lHasInsulation = plInsulation.

    end method.

    method public void AddSolar( input plSolar as logical ):

        lHasSolar = plSolar.

    end method.
```

- The variables store the desired capabilities for use by the builder classes

  … could be a temp-table, JSON object or other more complex data structure

© 2023 Consultingwerk Software Services Ltd. All rights reserved.

Demo – Abstract Factory & Builders

© 2023 Consultingwerk Software Services Ltd. All rights reserved.

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

© 2023 Consultingwerk Software Services Ltd. All rights reserved.

Enabling a fluent interface

```
INTERFACE 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.

CLASS FluentHouseBuilder IMPLEMENTS IFluentHouseBuilder:

  METHOD PUBLIC IFluentHouseBuilder AddHeatPump( plHeatPump AS logical ):

    lHasHeatPump = plHeatPump.
    RETURN THIS-OBJECT.

  END METHOD.
```

© 2023 Consultingwerk Software Services Ltd. All rights reserved.

Fluent interface: example

```
DEFINE VARIABLE oHouse AS IHouse.

// Insulated house with heat pump
oHouse = FluentHouseBuilder:Build()

                           :AddInsulation(TRUE)
                           :AddHeatPump(TRUE)

                           :House.
```

© 2023 Consultingwerk Software Services Ltd. All rights reserved.

Builders

- The factory aspects represent the logical view of what's being built; builders create a physical representation of that view
- Builders are the key to extensibility, flexibility
  - Some form of extensible configuration specifying them is important

    E.g config file, Service Manager, class registry

```
method static public IHouseBuilder Build (pcCategory as character):
   case pcCategory:
     when "modern" then return new ModernHouseBuilder().
     when "basic"  then return new BasicHouseBuilder().
     otherwise          return new DefaultHouseBuilder().
   end case.
 end method.
```

© 2023 Consultingwerk Software Services Ltd. All rights reserved.

Demo

- Builders
- Fluent interface

© 2023 Consultingwerk Software Services Ltd. All rights reserved.

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

- For maximum effect, they should have configurable builders

  … via configuration file or class registries

© 2023 Consultingwerk Software Services Ltd. All rights reserved.

Additional info

- Code shown today is available at https://github.com/4gl-fanatics/house-energy-patterns

- The ***Implementing and using the Decorator pattern in ABL*** session was ~~is~~ on Monday 13 Nov at 17:00. Come see where the requirements for building complex objects comes from (or download the slides after the conference).

© 2023 Consultingwerk Software Services Ltd. All rights reserved.

peter.judge@consultingwerk.com

Consultingwerk
software architecture and development

© 2023 Consultingwerk Software Services Ltd. All rights reserved.