Implementing and using the Decorator pattern in ABL

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, …
- …

Agenda

- Software design patterns
  - General
  - Decorator
- Example

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

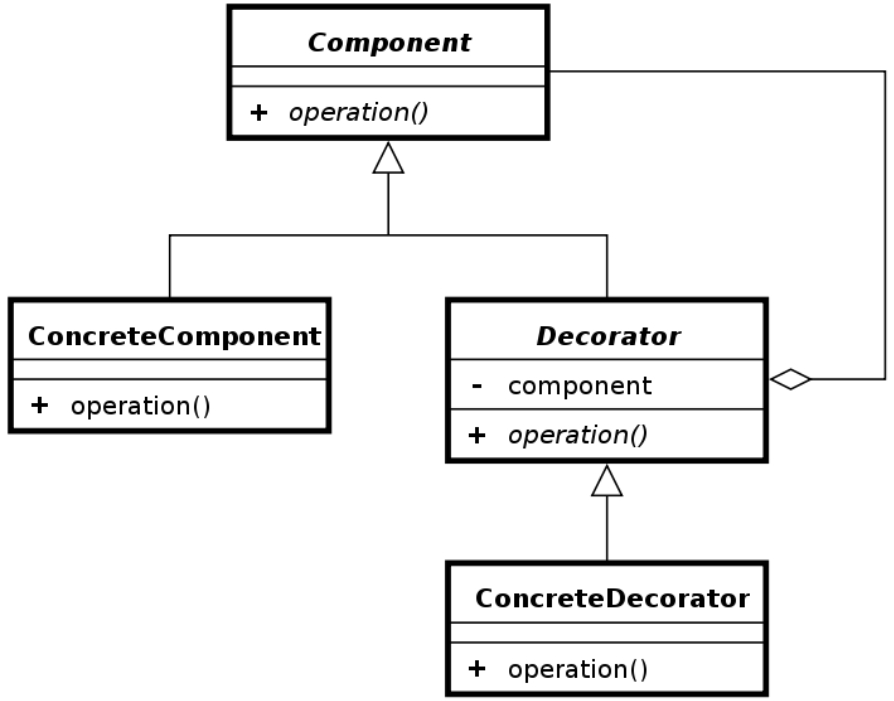
Software Design Patterns

- 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….

Decorator pattern

In object-oriented programming, the decorator pattern is a design pattern that allows behavior to be added to an individual object, dynamically, without affecting the behavior of other objects from the same class.



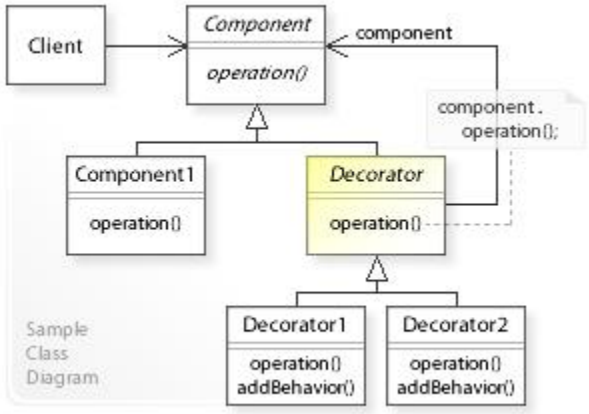
https://en.wikipedia.org/wiki/Decorator_pattern

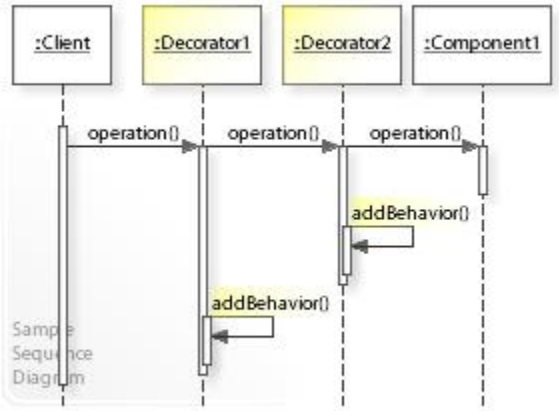
Decorator pattern

- 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)





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.

Example

- Define an Interface

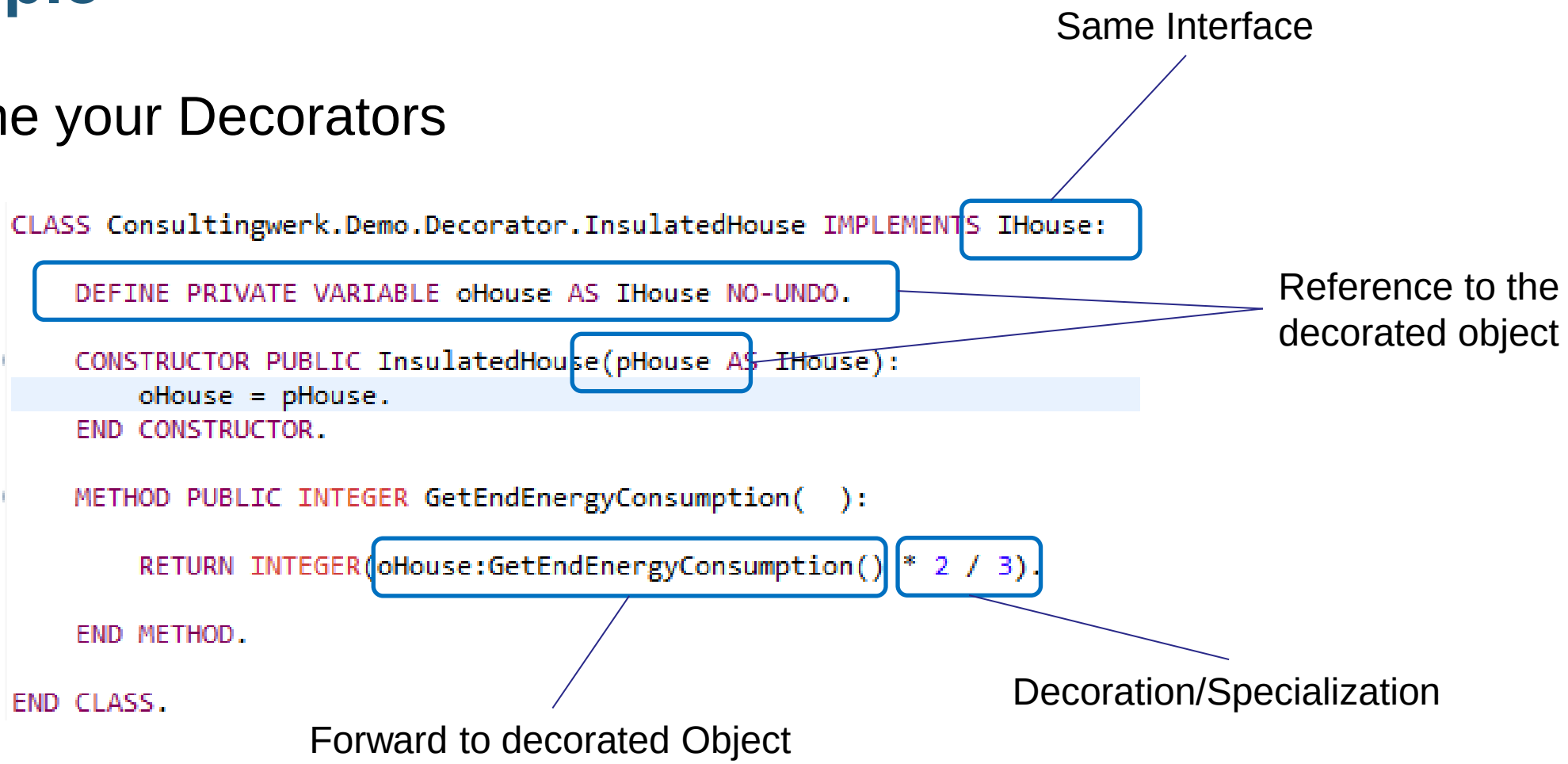
```
INTERFACE Consultingwerk.Demo.Decorator.IHouse:
    METHOD PUBLIC INTEGER GetEndEnergyConsumption().

END INTERFACE.
```

- Define your Basic Class

```
CLASS Consultingwerk.Demo.Decorator.BasicHouse IMPLEMENTS IHouse:

    METHOD PUBLIC INTEGER GetEndEnergyConsumption(  ):

        RETURN 24000.

    END METHOD.

END CLASS.
```

Example

- Define your Decorators




```
CLASS Consultingwerk.Demo.Decorator.InsulatedHouse IMPLEMENTS IHouse:

    DEFINE PRIVATE VARIABLE oHouse AS IHouse NO-UNDO.

    CONSTRUCTOR PUBLIC InsulatedHouse(pHouse AS IHouse):
        oHouse = pHouse.
    END CONSTRUCTOR.

    METHOD PUBLIC INTEGER GetEndEnergyConsumption(  ):

        RETURN INTEGER(oHouse:GetEndEnergyConsumption() * 2 / 3).

    END METHOD.

END CLASS.
```

Example

- Use a Decorator (or multiple) at runtime to change behavior

```
USING Consultingwerk.Demo.Decorator.* FROM PROPATH.

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

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

CURRENT-WINDOW:WIDTH = 320.

//a basic house
oHouse = NEW BasicHouse().
DISPLAY "Basic House Energy Consumption is: " oHouse:GetEndEnergyConsumption() "kWh/year" SKIP WITH WIDTH 320.

//add insulation
oHouse = NEW InsulatedHouse(oHouse).
DISPLAY "Insulated House Energy Consumption is: " oHouse:GetEndEnergyConsumption() "kWh/year" SKIP WITH WIDTH 320.

//replace gas stove with heat pump
oHouse = NEW HeatPumpHouse(oHouse).
DISPLAY "Energy Consumption with Heat Pump and Insulation is: " oHouse:GetEndEnergyConsumption() "kWh/year" SKIP WITH WIDTH 320.

//install some solar panels
oHouse = NEW SolarPoweredHouse(oHouse).
DISPLAY "Energy Consumption after solar installation drops to: " oHouse:GetEndEnergyConsumption() "kWh/year" SKIP WITH WIDTH 320.
```

Progress

```
Basic House Energy Consumption is:          24.000    kWh/year
Insulated House Energy Consumption is:      16.000    kWh/year
Energy Consumption with Heat Pump and Insulation is:     4.000    kWh/year
Energy Consumption after solar installation drops to:    3.000    kWh/year
```

Worse Example

- Use a derived class that is specialized

```
CLASS Consultingwerk.Demo.Decorator.HouseWithHeatPumpAndSolar INHERITS BasicHouse:
    METHOD PUBLIC OVERRIDE INTEGER GetEndEnergyConsumption(  ):

        RETURN INTEGER(SUPER:GetEndEnergyConsumption()
                       * 0.25  // COP
                       * 0.75). // Solar

    END METHOD.

END CLASS.
```

- Each special IHouse implementation would be a class
- You cannot dynamically change behavior

Conclusion

- Decorator pattern allows us to dynamically (ie at runtime) add behavior to objects
  - This can be chained to add more behavior

- Avoid creation of “special” classes at compile time
  - In the Example you can mix and match Decorators to get the behavior as desired.
  - Using Inheritance to specialize you would have created a class per specialization. 3 Variants = 2^3 classes to represent the options instead of 3 decorators only.

Additional info

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

- The *Using the Factory Pattern in OOABL: How, when and why* session is on Thursday / 16:30 in Room 525 . Come see how we improve building of these decorated objects.


lutz.fechner@consultingwerk.de