Consultingwerk
software architecture and development

Implementing and using the Decorator pattern in ABL

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

Consultingwerk Software Services Ltd.

Progress Partner
PREMIER SERVICE DELIVERY
#ProgressProud

- 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
- Inheritance gets ugly
- Decorator
- Adapter

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

Implementation Options

```
class BasicHouse implements IHouse

class SolarHouse inherits BasicHouse

class HeatPumpHouse inherits BasicHouse

class InsulatedHouse inherits BasicHouse

class BatteryBackupHouse inherits BasicHouse

class SolarInsulatedHouse inherits BasicHouse
class SolarHeatPumpInsulatedHouse inherits BasicHouse
class SolarHeatPumpInsulatedBatteryHouse inherits BasicHouse
class SolarHeatPumpBatteryHouse inherits BasicHouse
class SolarBatteryHouse inherits BasicHouse
```

- Challenge is supporting zero, one or more of these optional capabilities

  n capabilities = ***2***^{n} combinations

- A new capability (wind?) means a proliferation of classes

- Potential duplication of implementations

Software Design Patterns

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

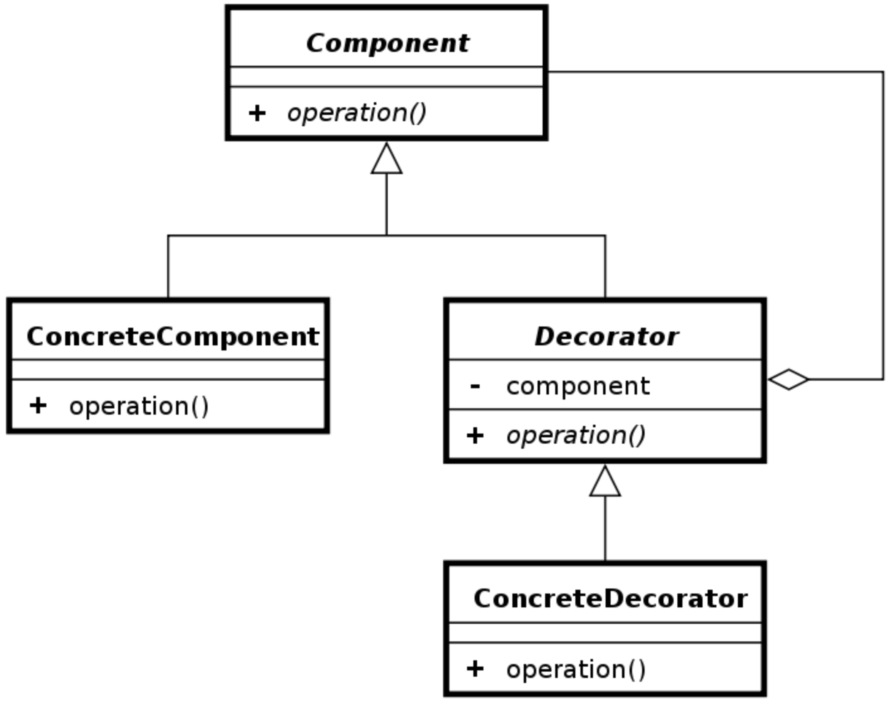
Software Design Patterns

- 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

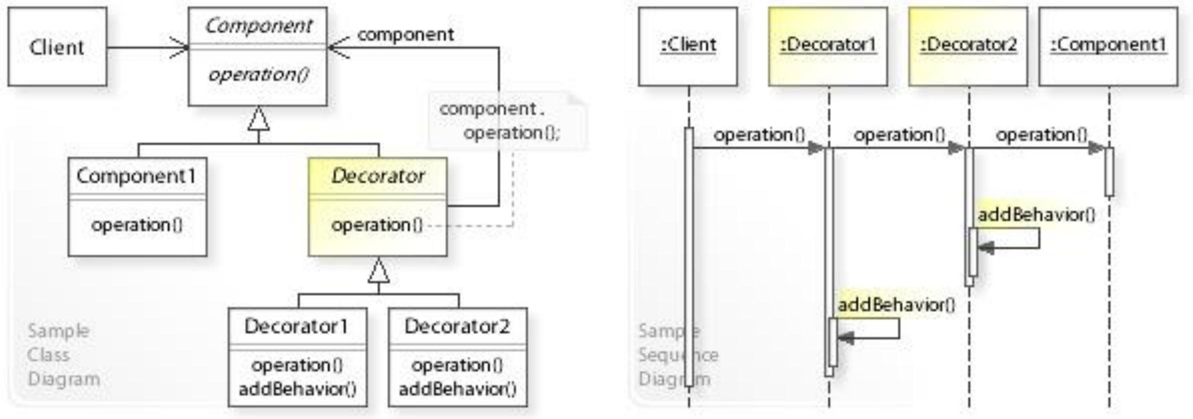
Implementing the Decorator pattern

Building blocks

1. Interface describing common functionality

2. A Decorator class, usually abstract to hold the decorated object

3. Concrete decorator classes for each bit of additional functionality



Abstract HouseDecorator

```
class HouseDecorator
    abstract
    implements IHouse:

    define variable oDecoratedHouse as IHouse no-undo.

    constructor public HouseDecorator(poHouse as IHouse):
        Assert:NotNull(poHouse).

        oDecoratedHouse = poHouse.
    end constructor.

    method public integer GetEndEnergyConsumption():
        return oDecoratedHouse:GetEndEnergyConsumption().
    end method.

end class.
```

Class is defined as ABSTRACT

Implements the IHouse interface

Private variable to hold the instance being decorated. This is also an IHouse

Decorated instance set via constructor

Decorator IHouse members simply call corresponding PUBLIC members on decorated instance

Concrete HouseDecorator

```
class HeatPumpHouse
    inherits HouseDecorator:

    // Coefficient of Performance - How much energy returned for energy put in
    define public property CoefficientOfPerformance as integer no-undo
    get.
    protected set.

    constructor public HeatPumpHouse(pHouse as IHouse):
        super(pHouse).

        this-object:CoefficientOfPerformance = 4.
    end constructor.

    method override public integer GetEndEnergyConsumption(  ):

        return integer(super:GetEndEnergyConsumption() / CoefficientOfPerformance).

    end method.

end class.
```

Inherits from HouseDecorator

Passes the House being decorated to the abstract parent

Modifies / extends standard House behaviour

Building objects

1. Build an object that will be decorated
2. Pass that into a decorator
3. Optionally pass the decorator into another decorator
4. Call methods on the IHouse
   - This is the outermost decorator

```
define variable oMyHouse as IHouse no-undo.

/* Base house (the one to decorate) */
oMyHouse = new BasicHouse().


/* Add Insulation */
oMyHouse = new InsulatedHouse(oMyHouse).


/* And a Heat Pump */
oMyHouse = new HeatPumpHouse(oMyHouse).

/* It's still an IHouse instance */
oMyHouse:GetEndEnergyConsumption().  // 5000w
```

Demo – Decorators,

Accessing other capabilities

```
define variable oMyHouse as IHouse no-undo.

/* Base house (the one to decorate) */
oMyHouse = new BasicHouse().

/* Add Insulation */
oMyHouse = new InsulatedHouse(oMyHouse).

/* And a Heat Pump */
oMyHouse = new HeatPumpHouse(oMyHouse).

/* It's still an IHouse instance */
oMyHouse:GetEndEnergyConsumption().  // 5000w

message cast(oMyHouse, InsulatedHouse):RValue.                          /* !! RUNTIME ERROR !! */

message 'type-of IHouse? ' type-of(oMyHouse, IHouse).                   /* true */
message 'type-of InsulatedHouse? ' type-of(oMyHouse, InsulatedHouse).   /* false */
message 'TypeName ' oMyHouse :GetClass():TypeName.                      /* HeatPumpHouse */
```

Accessing other capabilities

```
define variable oMyHouse as IHouse no-undo.

/* Base house (the one to decorate) */
oMyHouse = new BasicHouse().

/* And a Heat Pump */
oMyHouse = new HeatPumpHouse(oMyHouse).

/* Add Insulation */
oMyHouse = new InsulatedHouse(oMyHouse).

/* It's still an IHouse instance */
oMyHouse:GetEndEnergyConsumption().  // 5000w

message cast(oMyHouse, InsulatedHouse):RValue.                             /* 40 */

message 'type-of IHouse? ' type-of(oMyHouse, IHouse).                     /* true */
message 'type-of InsulatedHouse? ' type-of(oMyHouse, InsulatedHouse).     /* true*/
message 'TypeName ' oMyHouse :GetClass():TypeName.                        /* InsulatedHouse */
```

Using the Adapter Pattern

- Can't rely on `TYPE-OF()` or `Progress.Lang.Class:IsA()` since we are dealing with >1 class
  - `TYPE-OF()` will tell only us the outermost layer of the onion … we need to inspect all of the layers somehow

The adapter pattern is a software design pattern (also known as wrapper, an alternative naming shared with the decorator pattern) that allows the interface of an existing class to be used as another interface

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

Implement OpenEdge.Core.IAdaptable

```
class HouseDecorator
  abstract
  implements IHouse,
             OpenEdge.Core.IAdaptable :

  method public Progress.Lang.Object GetAdapter(pAdaptTo as Progress.Lang.Class):
    /* Does the current decorator implement or inherit from the requested type? */
    if this-object:GetClass():IsA(pAdaptTo) then
      return this-object.

    if valid-object(oDecoratedHouse) then do:
      /* Does the decorated house implement or inherit the requested type? */
      if oDecoratedHouse:GetClass():IsA(pAdaptTo) then
        return oDecoratedHouse.

      /* Is the decorated house itself an Adapter? */
      if type-of(oDecoratedHouse, OpenEdge.Core.IAdaptable) then
        return cast(oDecoratedHouse, OpenEdge.Core.IAdaptable):GetAdapter(pAdaptTo).
    end.

    return ?.
  end method.

end class.
```

Consultingwerk
software architecture and development

Accessing other capabilities via an adapter

```
define variable oAdapter as Progress.Lang.Object no-undo.
define variable oMyHouse as IHouse no-undo.

/* Base house (the one to decorate) */
oMyHouse = new BasicHouse().

/* Add Insulation */
oMyHouse = new InsulatedHouse(oMyHouse).

/* And a Heat Pump */
oMyHouse = new HeatPumpHouse(oMyHouse).

/* It's still an IHouse instance */
oMyHouse:GetEndEnergyConsumption().  // 5000w

message 'type-of IHouse? ' type-of(oMyHouse, IHouse).                  /* type-of IHouse? true */
message 'TypeName ' oMyHouse :GetClass():TypeName.                      /* TypeName HeatPumpHouse */

if type-of(oMyHouse, IAdaptable) then do:
  oAdapter = cast(oMyHouse, IAdaptable):GetAdapter(get-class(InsulatedHouse)).
  if valid-object(oAdapter) then
    message cast(oAdapter, InsulatedHouse):RValue.                      /* works! */
end.
```

Demo

- Run-adapter.p

Conclusion

- Decorator pattern allows us to dynamically (ie at runtime) add behaviour to objects
  - Streamline complex object hierarchies when using multiple interfaces
  - Construction of decorated objects can be verbose (hint: come and see the "Factories" session

- The adapter pattern allows us to ask an object what it is capable of doing
  - … and lets us ignore how that's implemented in the object (inheritance or decorator or …)

Additional info

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

- The *Using the Factory Pattern in OOABL: How, when and why* session is on Tuesday, 14 Nov / 16:40. Come see how we improve building of these decorated objects

peter.judge@consultingwerk.com

PRODUCTIVE
PROFESSIONAL
PROBLEMSOLVING
SMART
EDUCATION
LEADING
TECHSAVVY
RELIABLE
INNOVATIVE
GEEKY
WORLDWIDE
PROACTIVE
REMARKABLE
KNOWLEDGABLE