

OOABL PATTERNS & PRACTICES

PUG.DE Workshop



Mike Fechner

Consultingwerk Ltd

mike.fechner@consultingwerk.de

Peter Judge

Progress Software

pjudge@progress.com

AGENDA – MORNING

10.00

- Intro / Welcome
- OO terms and definitions
- Labs context
- Lab: setup/connect

10.45

- Coffee break

11.00

- Coding to abstraction: interfaces & abstract classes
- Lab: coding to abstractions

12.00

- Lunch

AGENDA - AFTERNOON

13.00

- Value objects
- Enums
- Strongly-typed Events

14.00

- Abstract builder pattern
- Lab: Create a builder
- Fluent interfaces
- Lab: Create a fluent interface

14.30

- Coffee Break

15.00

- Dependency injection
- Lab - inject DAO into BE
- Singletons, object lifespans
- Service Manager
- Lab - implement SvcMgr

~17.00

- Closing

DEFINITIONS

Types	type defines the set of requests to which it can respond. includes classes, interfaces, enums
Strong typing	compile-time enforcement of rules
Member	stuff "inside" a type - methods, properties, variables, etc
Access control	compile-time restriction on member visibility: public, protected, private
Class	type with executable code; implementation of a type
Abstract class	non-instantiable (non-runnable) class that may have executable code
Static	members loaded once per session. think GLOBAL SHARED
Interface	type with public members without implementations
Enum(eration)	strongly-typed name int64-value pairs
Object	running classes, aka instance

COMMON TERMS

Software Design Patterns: general reusable solution to a commonly occurring problem within a given context

- Parameter value
- Fluent Interfaces
- Factory method
- Singleton

SOLID principles

- S**ingle responsibility
- O**pen-closed
- L**iskov substitution
- I**nterface segregation
- D**ependency inversion

http://en.wikipedia.org/wiki/Software_design_pattern

PASTA PATTERNS

SPAGHETTI ... A HUGE MESS ALL TANGLED TOGETHER

LASAGNA ... TOO MANY UNNEEDED LAYERS
ADDING NOTHING

RAVIOLI ... JUST ENOUGH LAYERING AND JUST
RIGHT ENCAPSULATION

SOFTWARE GOALS

- Flexibility
 - Allow implementations to be swapped without changes to the calling/consuming code
 - Make testing (mocking) easier
- Modularity & extensibility
 - Allow independent (teams / companies) to develop frameworks and components
 - Allow later / third-party extension of components without changing initial / core components

EXTENSIBILITY: PROCEDURES & OBJECTS

- Procedures can call classes; classes can call procedures
- Lets you incrementally add OOABL
- Necessary for certain cases
 - Callbacks
 - AppServer event procedures
 - Session start (-p main.p)
- Pass objects to procedures as parameters
- Objects can include includes

LABS CONTEXT:TERMS

- **Business Entity**

It contains/depends on

- Data Access Object
- Authorization Manager
- Logging Manager
- Error Manager

Logical representation of a business function / task

Fetches and save data to and from db from BE

Determines whether user can perform operations

Writes diagnostic information out

Processes errors raised in a standard way

- **Service Interface / gateway**

It contains/depends on

- Authorization Manager

Receives client request and passes to correct BE

Determines whether user can perform operations



Interface

Abstract

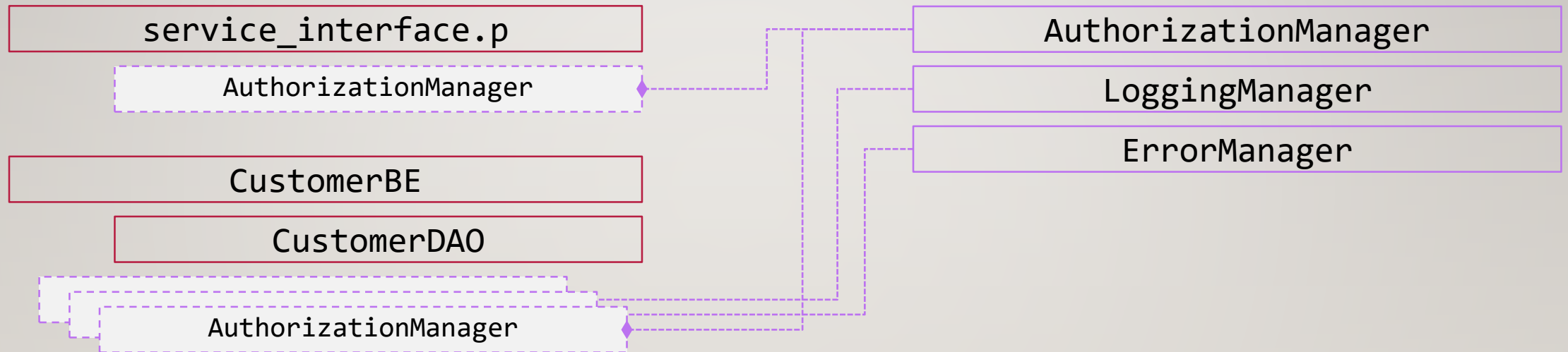
Manager

Service

LABS CONTEXT: STARTING STATE

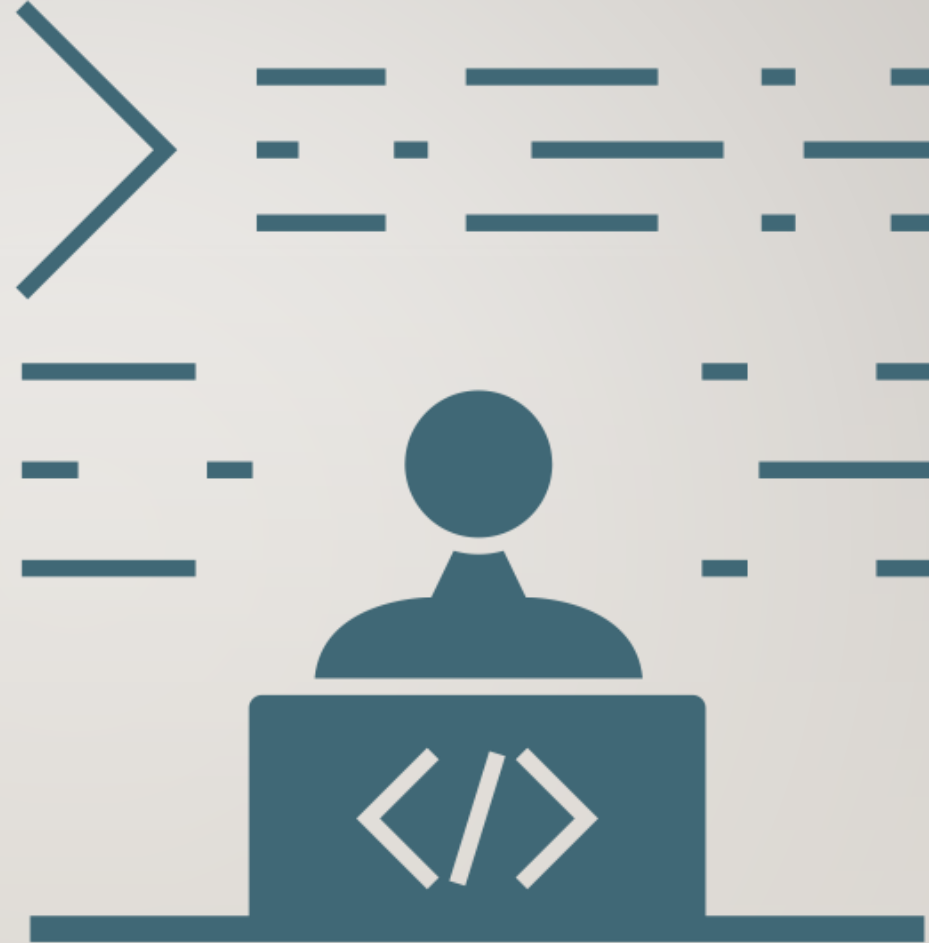
Services.*

Managers.*



LAB: CONNECTION TO YOUR LAB MACHINE

Make sure you can connect to the
lab instances on Amazon using some
form of Remote Desktop



BREAK



CODE TO ABSTRACTION

Interfaces & Abstract classes

OPEN/CLOSED PRINCIPLE

- Design as open for enhancement – while closed for modifications
- Design to a contract (Interface) on the sub-system level, not just a single class
- Domains manage complexity and impact of change
- A change in functionality in one domain does not require changes to other domains
- Through message interaction an event in a certain domain may cause actions in one or multiple domains without impact on the originating domain
- Simplifies testing. Allows mocking of a whole sub-system

OPEN/CLOSED PRINCIPLE

- Business Requirement: Confirming an order needs to commit inventory/stock allocation
- If Order Business Entity would directly call into the Inventory/Stock Business Entity this would create a direct dependency between the two Business Entities
 - Change in the implementation of the Inventory/Stock Business Entity might affect the Order Business Entity
- If the Order Business Entity however, would publish a message using a Message Publisher infrastructure, the Inventory/Stock Domain may – or may not at it's own responsibility perform required action

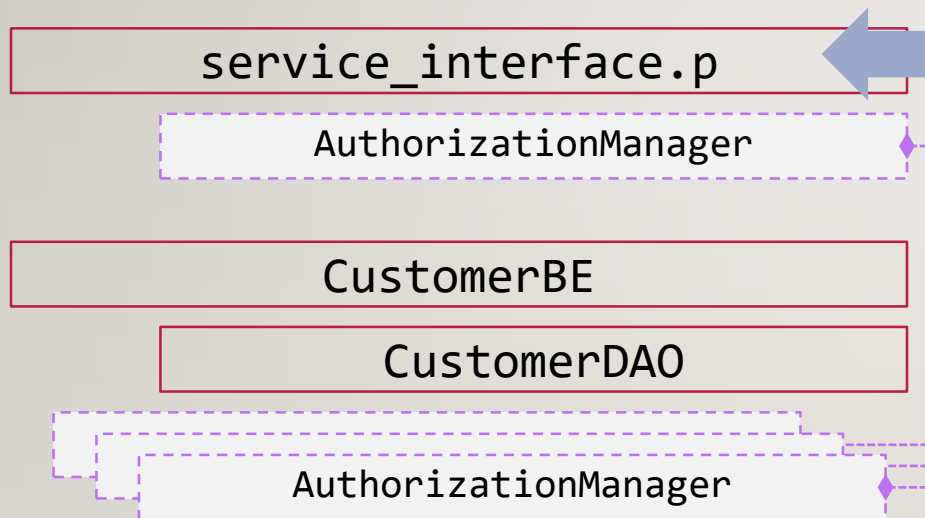
USE 'CONTRACT' TYPES FOR VARIABLE, PARAMETER DEFINITIONS

- Use interfaces and/or abstract classes for defining the programming interface
 - Neither can be instantiated
 - Compiler requires that implementing/concrete classes fulfill a contract
- Interfaces preferred
 - Can use multiple at a time
 - Now have *I-will-not-break* contract with implementers
- Use inheritance for common or shared behaviour
 - Careful of deep hierarchies – reduces flexibility

Interface	Abstract
Manager	Service

ISSUES: SERVICE INTERFACE

Services.*



```

def input param pcServiceName as char.
def input param pcOperation as char.

def in-out param dataset-handle phServiceData.
def in-out param dataset-handle phServiceParams.
def var oCustBE as CustomerBE.
def var oOrderBE as OrderBE.
def var oAuthMgr as AuthorizationManager.

oAuthMgr = new AuthorizationManager().
oAuthMgr:AuthorizeServiceOperation(pcServiceName,
pcOperation).

case pcServiceName:
when 'Customer' then
do:
oCustBE = new CustomerBE().
if pcOperation eq 'fetch' then
oCustBE:Fetch(<args>).
else if pcOperation eq 'save' then
oCustBE:Save(<args>).
end.
when 'Orders' then
oOrderBE = new OrderBE().
/* similar code for operations */
end case.
  
```

Interface

Abstract

Manager

Service

ISSUES: BUSINESS ENTITY

Services.*

service_interface.p

AuthorizationManager

CustomerBE

CustomerDAO

AuthorizationManager

```
class Services.CustomerBE:
  def public prop DataAccess as CustomerDAO
    get. set.
  def public prop LogMgr as LoggingManager
    get. set.
  def public prop ErrorMgr as ErrorManager
    get. set.
  def public prop AuthMgr as AuthorizationManager
    get. set.

  constructor public CustomerBE():
    DataAccess = new CustomerDAO().
    LogMgr = new LoggingManager().
    ErrorMgr = new ErrorManager().
    AuthMgr = new AuthorizationManager().
  end constructor.

  method public void Fetch(<params>):
  end method.
  method public void Save(<params>):
  end method.
end class.
```

Interface

Abstract

Manager

Service

SOLUTION: USE INTERFACES

Services.*

service_interface.p

IAuthorizationManager

IBusinessEntity

BusinessEntity

CustomerBE

IDataAccess

CustomerDAO

IAuthorizationManager

```
interface Services.IBusinessEntity:
  def public prop DataAccess as IDataAccess
    get. set.

  method public void Fetch(
    input-output dataset-handle phData,
    input-output dataset-handle phParams).

  method public void Save (
    input-output dataset-handle phData,
    input-output dataset-handle phParams).

end interface.
```

Interface

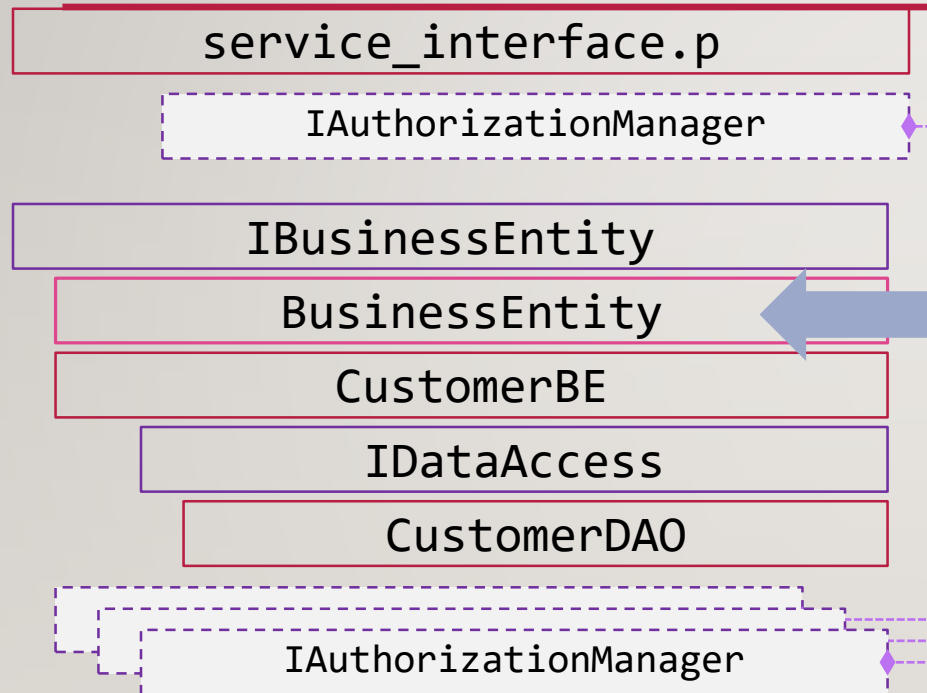
Abstract

Manager

Service

SOLUTION: ABSTRACT SUPER-CLASS

Services.*



```

class Services.BusinessEntity
    abstract implements IBusinessEntity:
    def public prop DataAccess as IDataAccess get. set.
    def public prop LogMgr as ILoggingManager get. set.
    def public prop ErrorMgr as IErrorManager get. set.
    def public prop AuthMgr as IAuthorizationManager
        get. set.

    method public void Fetch(<params>):
        this-object:DataAccess:Fetch(<args>).
    end method.

    method abstract protected void ValidateSave(
        <params>).

    method public void Save (<params>):
        this-object:ValidateSave(<args>).
        this-object:DataAccess:Save(<args>).
    end method.
end class.

```


Interface

Abstract

Manager

Service

SOLUTION: USE SUPER-CLASS

Services.*

service_interface.p

IAuthorizationManager

IBusinessEntity

BusinessEntity

CustomerBE

IDataAccess

CustomerDAO

IAuthorizationManager

```
class Services.CustomerBE
    inherits BaseEntity:

    constructor public CustomerBE():
        DataAccess = new CustomerDAO().
    end constructor.

    method override protected void ValidateSave(
        <params>):

        def var hBuffer as handle.
        hBuffer = phData:get-buffer-handle(1).

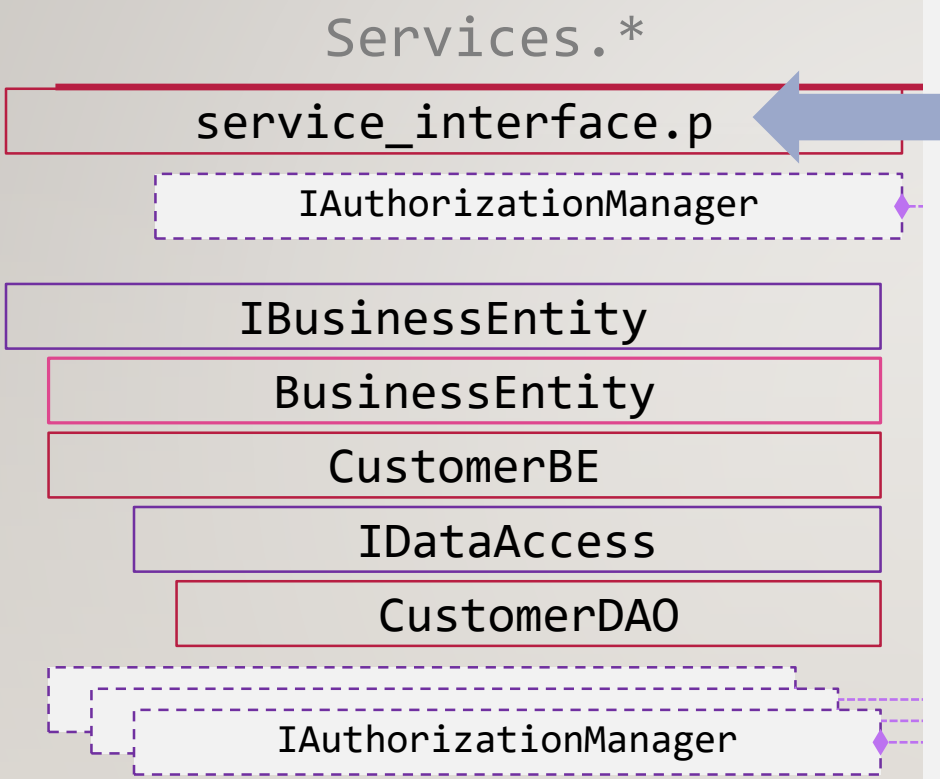
        hBuffer:find-first().

        if hBuffer::CustNum le 0 then
            return error new AppError(
                'CustNum must be positive').
        end method.

end class.
```

Interface	Abstract
Manager	Service

SOLUTION: CALL INTERFACES



```

def input param pcServiceName as char.
def input param pcOperation as char.
def in-out param dataset-handle phServiceData.
def in-out param dataset-handle phServiceParams.
def var oBE as IBusinessEntity.

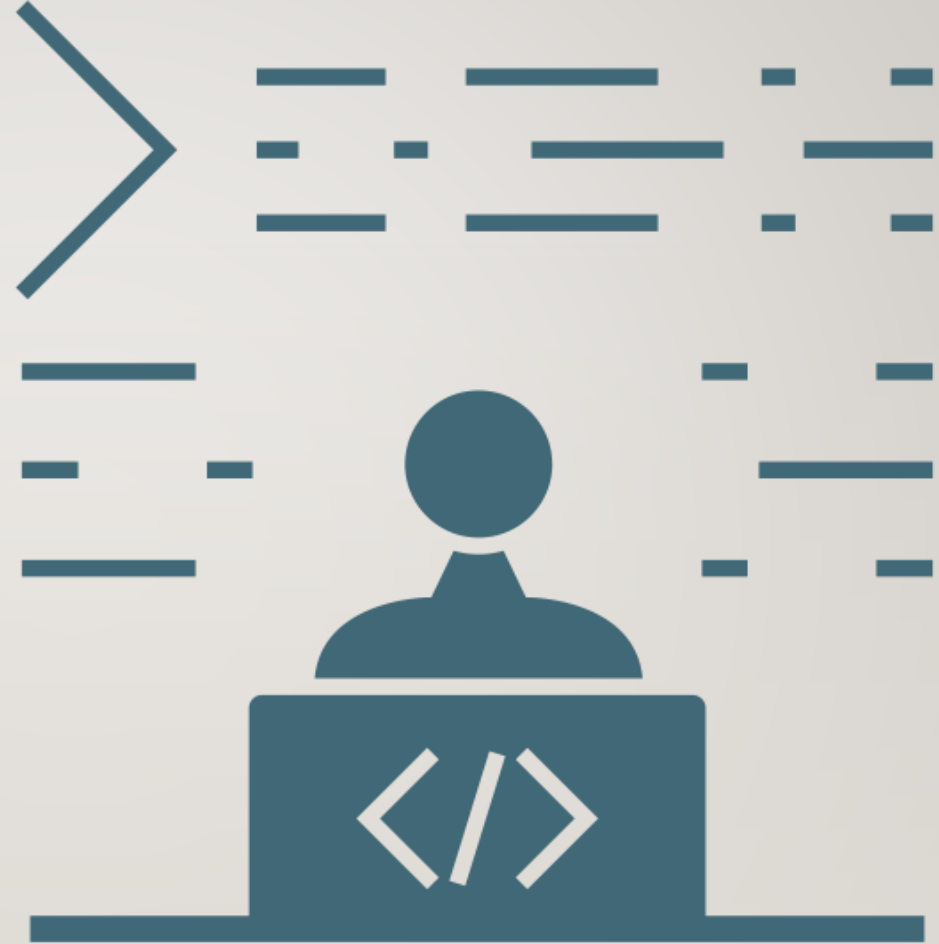
oAuthMgr = new AuthorizationManager().
oAuthMgr:AuthorizeServiceOperation(pcServiceName,
pcOperation).

case pcServiceName:
  when 'Customer' then oBE = new Services.CustomerBE().
  when 'Orders'   then oBE = new Services.OrderBE().
end case.

case pcOperation:
  when 'fetch' then oBE:Fetch(<args>).
  when 'save'  then oBE:Save(<args>).
end case.
  
```

LAB: CODING TO ABSTRACTIONS

Refactor to use interfaces & abstract
classes



AFTER LAB

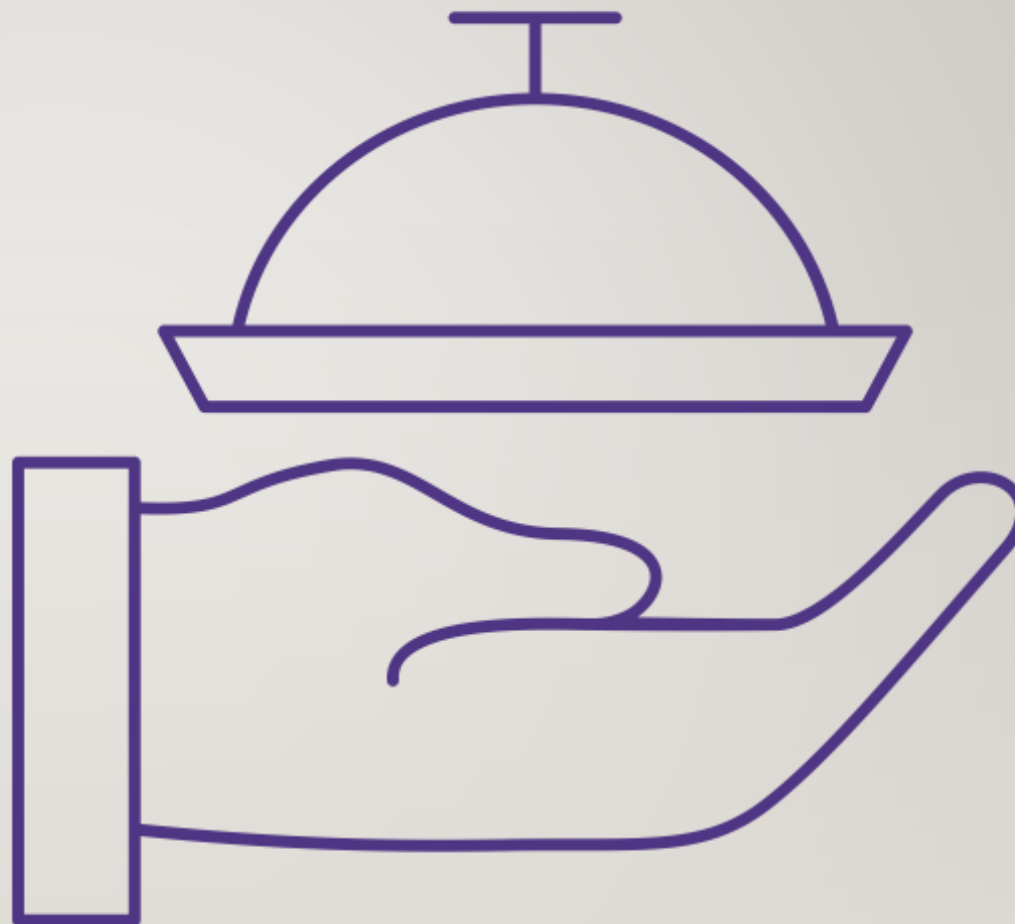
You should have created the following

- The `Services.IBusinessEntity` interface
- The `Services.BusinessEntity` abstract class that implements the interface

And changed the following programs to use the new types

- `Services.CustomerBE` to inherit from the abstract class
- `Services/service_interface.p` to change the variable types

LUNCH



SOME USEFUL CONCEPTS

Value objects

enumerations

strongly-typed events & event args

VALUE OBJECTS



A **value object** is a small object that represents a simple entity whose equality is not based on identity: i.e. two value objects are equal when they have the same value, not necessarily being the same object.

Value objects should be immutable

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

VALUE OBJECTS

```
/* calling */  
oModel:SetValue('Norah').
```

```
/* definition */  
class Example.Data.Model:  
    method public void SetValue (input pValue as character).
```

```
end class.
```


VALUE OBJECTS

```
/* calling */
oModel:SetValue('Norah').
oModel:SetValue('Norah', 'x(20)').

/* definition */
class Example.Data.Model:
  method public void SetValue (input pValue as character).
  method public void SetValue (input pValue as character,
                               input pFormat as character).

end class.
```

VALUE OBJECTS

```
/* calling */
oModel:SetValue('Norah').
oModel:SetValue('Norah', 'x(20)').
oModel:SetValue('Norah', 'x(20)', 'UTF-8').

/* definition */
class Example.Data.Model:
    method public void SetValue (input pValue as character).
    method public void SetValue (input pValue as character,
                                input pFormat as character).
    method public void SetValue (input pValue as character,
                                input pFormat as character,
                                input pEncoding as character).
end class.
```

VALUE OBJECTS

```
/* calling */
oString = new Example.String().
oString:Value = 'Norah'.

oModel:SetValue(oString).

/* definition */
class Example.String:
  define public property Value as character no-undo get. set.

end class.

class Example.Data.Model:
  method public void SetValue (input pValue as Example.String).
end class.
```

VALUE OBJECTS

```
/* calling */
oString = new Example.String().
oString:Value      = 'Norah'.
oString:Format     = 'x(20)'.
oString:Encoding   = 'UTF-8'.
oModel:SetValue(oString).

/* definition */
class Example.String:
  define public property Value      as character no-undo get. set.
  define public property Format     as character no-undo get. set.
  define public property Encoding   as character no-undo get. set.
end class.

class Example.Data.Model:
  method public void SetValue (input pValue as Example.String).
end class.
```


IMMUTABLE VALUE OBJECTS

```
/* calling */
oString = new Example.String('Norah', 'x(20)', 'UTF-8').
oModel:SetValue(oString).

// definition
1. class Example.String:
2.     // read-only properties
3.     define public property Value as character no-undo get. set
4.     define public property Format as character no-undo get. set
5.     define public property Encoding as character no-undo get. set
6.     // values set in constructor(s)
7.     constructor public String(input pValue    as character,
8.                               input pFormat   as character,
9.                               input pEncoding as character):
10.         assign this-object:Value    = pValue
11.             this-object:Format      = pFormat
12.             this-object:Encoding    = pEncoding.
13.     end constructor.
14. end class.

class Example.Data.Model:
    method public void SetValue (input pValue as Example.String).
end class.
```

ENUMS



An **enumerated type** (also called enumeration, `enum[...]`) is a data type consisting of a set of named values called elements, members, enumeral, or enumerators of the type. The enumerator names are usually identifiers that behave as constants in the language

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



WHY USE ENUMS

```
procedure SetOrderStatus(input pOrderNum as integer,  
                        input pStatus as integer):  
    OpenEdge.Core.Assert:IsPositive(pStatus, 'Order status').  
  
    find Order where Order.OrderNum eq pOrderNum exclusive-lock.  
    assign Order.OrderStatus = pStatus.  
    // what happens when the integer isn't a valid status? How do we know ?  
end procedure  
  
run SetOrderStatus (12345, 0).    // this throws an error via the Assert  
run SetOrderStatus (12345, 1).    // what is this status?  
run SetOrderStatus (12345, 2).
```

DEFINING ENUMS

```
// enum type
enum Services.Orders.OrderStatusEnum: //implicitly FINAL so cannot be extended
  // enum member
  define enum    Shipped          = 1  // default start at 0
                  Backordered      // = 2 . Values incremented in def order
                  Ordered
                  Open
                  Cancelled        = -1 // historical set of bad values
                  UnderReview      = -2
                  Default = Ordered.  // = 3
  // enum members are the only members allowed
  // enum members can only be used in enum types
end enum.
```


USING ENUMS

```
procedure SetOrderStatus(input pOrderNum as integer,  
                        input pStatus   as Services.Orders.OrderStatusEnum):  
    //ensures that we have a known, good status  
    OpenEdge.Core.Assert:NotNull(pStatus, 'Order status').  
  
    find Order where Order.OrderNum eq pOrderNum exclusive-lock.  
    assign Order.OrderStatus = pStatus:GetValue().  
    //Alternative way of getting the value  
    assign Order.OrderStatus = integer(pStatus).  
end procedure  
  
run SetOrderStatus (12345, OrderStatusEnum:None).           // COMPILE ERROR  
run SetOrderStatus (12345, OrderStatusEnum:Backordered).  
run SetOrderStatus (12345, OrderStatusEnum:Ordered).
```

STRONGLY-TYPED EVENTS & EVENT ARGS



An extensible Publish / Subscribe model with
compile-time checking



STRONGLY-TYPED EVENTS: DEFINE

- Name
- Signature
- Access level for subscriptions
- Note: classes cannot subscribe to untyped PUB/SUB events

```
class Example.Data.Model:  
  
    define public event StateChanged signature void (  
        input pData as character).  
  
end class.
```

STRONGLY-TYPED EVENTS: LISTEN

- Subscribe & Unsubscribe
- Signature-matched handler public method or internal procedure

```
class OpenEdge.UI.Grid:
  constructor Grid(input poModel as Example.Data.Model):
    poModel:StateChanged:Subscribe(this-object:StateChangedHandler).
  end constructor.
  method public void StateChangedHandler (input pData as character):
    /* when the data has changed in the model, update the UI */
    this-object:Refresh().
  end method.
  destructor Grid():
    poModel:StateChanged:Unsubscribe(this-object:StateChangedHandler).
  end destructor.
end class.
```


STRONGLY-TYPED EVENTS: RAISE

- Publish is **always** private

```
class Example.Data.Model:
  define public event StateChanged signature void (input pData as char).
  method public void SetValue (input pValue as character):
    /* do stuff with data */
    /* tell the world */
    OnStateChanged(input pValue).
  end method.
  /* Protected On<Event> method lets other classes raise the event */
  method protected void OnStateChanged(input pData as character):
    this-object:StateChanged:Publish(pData).
  end method.
end class.
```



EVENT HANDLERS WITH VALUE OBJECTS

- Event handlers have a standard signature*

```
define public event <name> signature void (input pSender AS CLASS Progress.Lang.Object ,  
                                             input pArgs    AS CLASS OpenEdge.Core.EventArgs).
```

- For static events, sender should be the type sending : GET-CLASS()
- Args are data about the event; operations like Cancel too
- A form of value object that follows .NET pattern

<https://blogs.msdn.microsoft.com/kcwalina/2005/11/18/why-do-we-have-eventargs-class/>

* Not required by the ABL but should be a best practice

EVENT ARGS EXAMPLE

```
1. class OpenEdge.Web.DataObject.HandleRequestEventArgs inherits OpenEdge.Core.EventArgs:
2.     /* (mandatory) The request being serviced */
3.     define public property Request as IWebRequest no-undo
4.         get. private set.
5.     /* (optional) An error that results from the handling of this event. */
6.     define public property Error as Progress.Lang.Error no-undo
7.         get. set.
8.     /* Indicates whether the operation should be cancelled */
9.     define public property Cancel as logical no-undo
10.        get. set.
11.     /* (optional) The status code to return for an operation.
12.        Zero = use the event args' Response for the entire response */
13.     define public property ReturnStatusCode as integer no-undo
14.        get. set.
15.     /* Constructor.
16.        @param IWebRequest The request that resulting in the exception */
17.     constructor public HandleRequestEventArgs(input poRequest as IWebRequest):
18.         super().
19.         Assert:NotNull(poRequest, 'Request').
20.         assign this-object:Request = poRequest.
21.     end constructor.
22. end class.
```

Interface

Abstract

Manager

Service

BACK TO OUR SOLUTION USING INTERFACES

Services.*

service_interface.p

IAuthorizationManager

IBusinessEntity

BusinessEntity

CustomerBE

IDataAccess

CustomerDAO

IAuthorizationManager

```
def input param pcServiceName as char.
def input param pcOperation as char.
def in-out param dataset-handle phServiceData.
def in-out param dataset-handle phServiceParams.
def var oBE as IBusinessEntity.

oAuthMgr = new AuthorizationManager().
oAuthMgr:AuthorizeServiceOperation(pcServiceName,
pcOperation).

case pcServiceName:
  when 'Customer' then oBE = new Services.CustomerBE().
  when 'Orders'   then oBE = new Services.OrderBE().
end case.

case pcOperation:
  when 'fetch' then oBE:Fetch(<args>).
  when 'save'  then oBE:Save(<args>).
end case.
```

Interface

Abstract

Manager

Service

IMPROVE WITH DYNAMIC-NEW

Services.*

service_interface.p

IAuthorizationManager

IBusinessEntity

BusinessEntity

CustomerBE

IDataAccess

CustomerDAO

IAuthorizationManager

```
def input param pcServiceName as char.
def input param pcOperation as char.
def in-out param dataset-handle phServiceData.
def in-out param dataset-handle phServiceParams.
def var oBE as IBusinessEntity.

oAuthMgr = new AuthorizationManager().
oAuthMgr:AuthorizeServiceOperation(pcServiceName,
pcOperation).

oBE = dynamic-new
      'Services.' + pcServiceName + 'BE' ().

case pcOperation:
  when 'fetch' then oBE:Fetch(<args>).
  when 'save'  then oBE:Save(<args>).
end case.
```


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?*

BUILDERS & FACTORIES



Abstract factories provide an interface for creating *families* of related or dependent objects without specifying their concrete classes https://en.wikipedia.org/wiki/Abstract_factory_pattern

A **builder** separates the construction of a complex object from its representation, allowing the same construction process to create various representations https://en.wikipedia.org/wiki/Builder_pattern

Factory methods define an interface for creating a *single* object, but let subclasses decide which class to instantiate

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



BUILDER PATTERN

```
class Services.BusinessEntityBuilder abstract:
```

Abstract factory

```
/* Returns a usable BusinessEntity */
```

```
define abstract public property Entity as IBusinessEntity no-undo
```

```
    get.
```

Factory method

```
end class.
```

BUILDER IMPLEMENTATION

```
class Services.DefaultBEBuilder inherits BaseEntityBuilder:
  define private variable mcServiceName as character no-undo.

  /* The getter implementation is the method that does the actual work */
  define override public property Entity as IBusinessEntity no-undo
    get():
      define variable oBE as IBusinessEntity no-undo.

      oBE = dynamic-new 'Services.' + pcServiceName + 'BE' ().

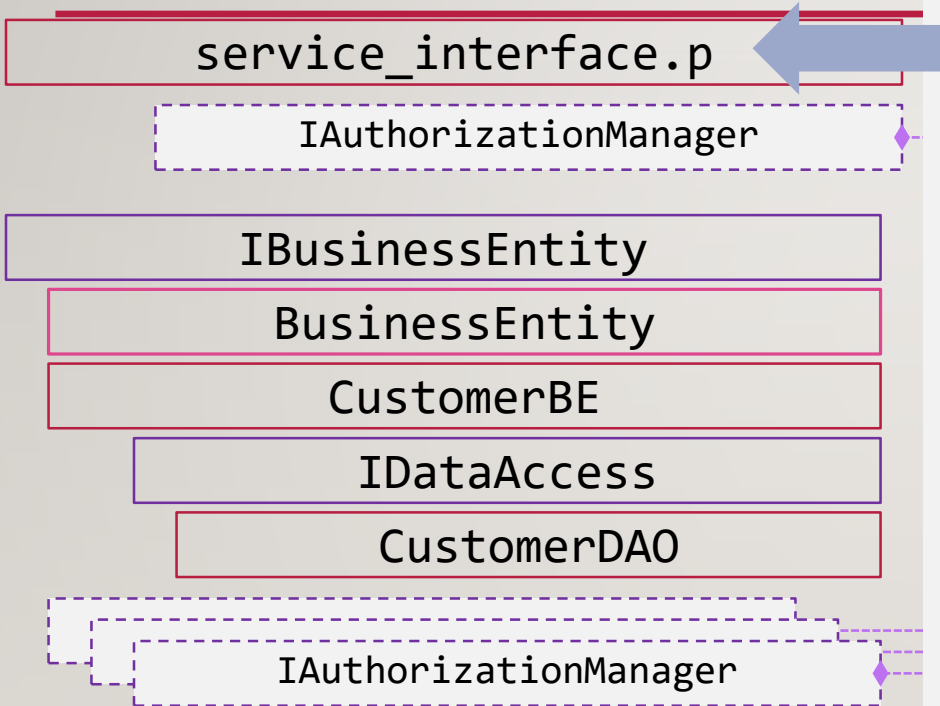
      return oBE.
    end get.

  constructor public DefaultBEBuilder(input pcServiceName as character):
    assign mcServiceName = pcServiceName.
  end constructor.
end class.
```

Interface	Abstract
Manager	Service

USING A BUILDER

Services.*



```

def input param pcServiceName as char.
def input param pcOperation as char.
def in-out param dataset-handle phServiceData.
def in-out param dataset-handle phServiceParams.
def var oBE as IBusinessEntity.

oAuthMgr = new AuthorizationManager().
oAuthMgr:AuthorizeServiceOperation(pcServiceName,
pcOperation).

def var beBuilder as BusinessEntityBuilder.

beBuilder = new Services.DefaultBEBUILDER
              (pcServiceName).

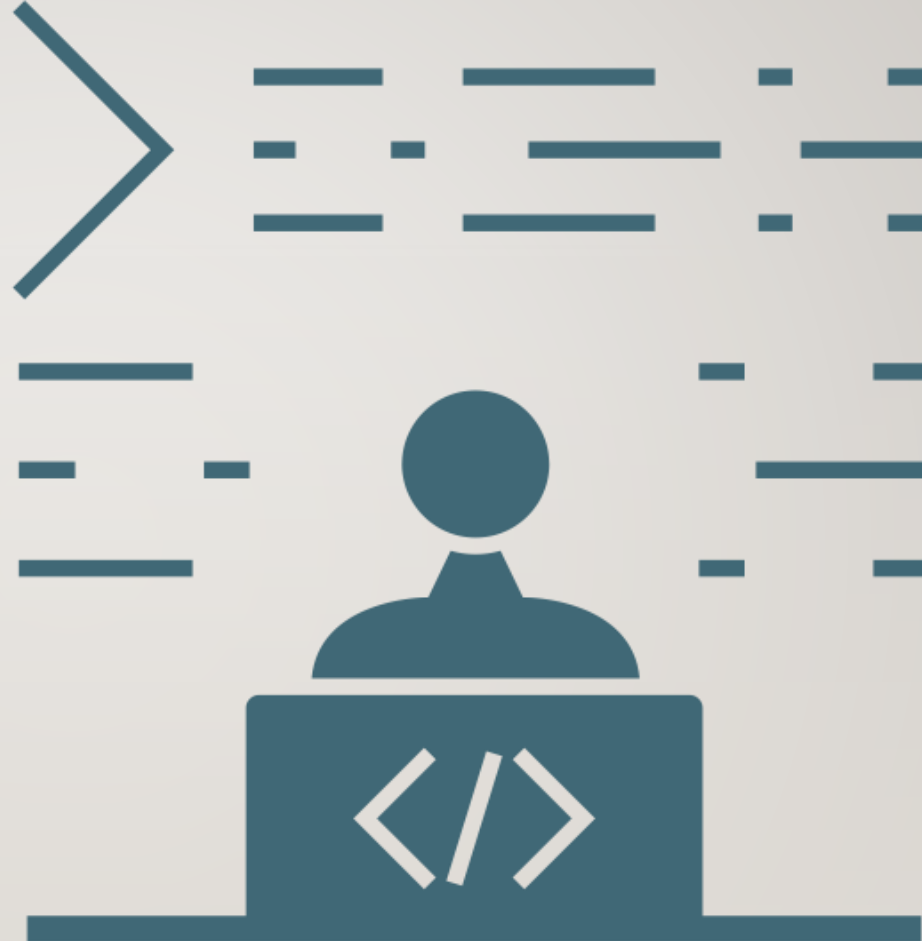
oBE = beBuilder:Entity.

case pcOperation:
  when 'fetch' then oBE:Fetch(<args>).
  when 'save'  then oBE:Save(<args>).
end case.
  
```


LAB : CREATE A BUILDER

Create builder that returns an instance of `IBusinessEntity`

Refactor the service interface code to call the builder to return a BE



WHERE ARE WE NOW?

- We have many abstract representations
 - Services: `IBusinessEntity` , `IDataAccess`
 - Managers: `IErrorManager` , `IAuthorizationManager` , `ILoggingManager`
- And many builders to return runnable implementations
 - Services: `BusinessEntityBuilder` , `DataAccessBuilder`
 - Managers: `ErrorManagerBuilder` , `AuthManagerBuilder`, `LogManagerBuilder`

Interface

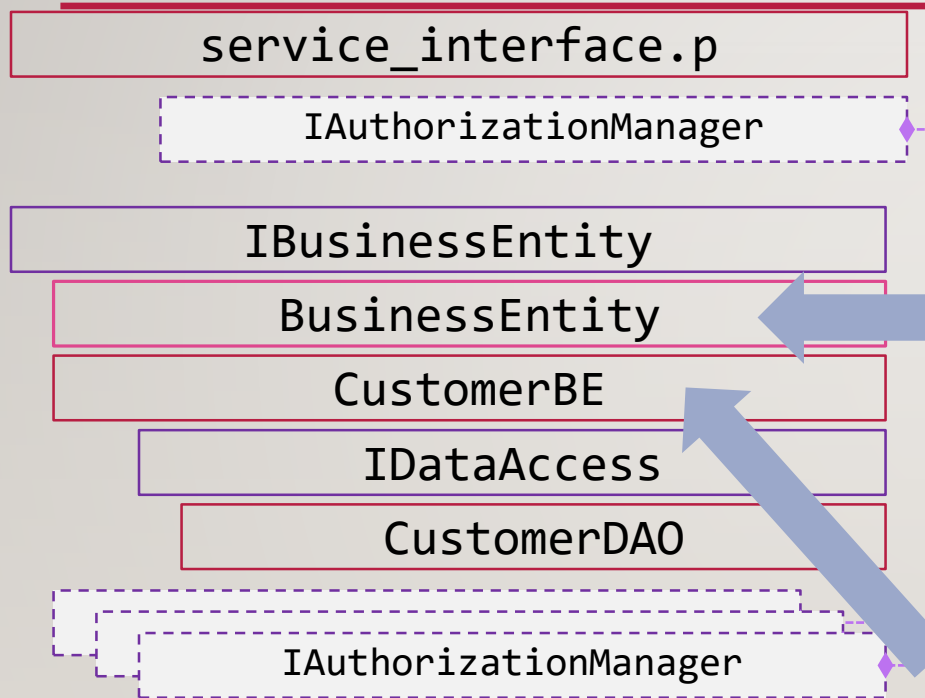
Abstract

Manager

Service

LOOSENING CHILD DEPENDENCIES

Services.*



```

class Services.BusinessEntity
    abstract implements IBusinessEntity:
    def public prop DataAccess as IDataAccess get. set.
    def public prop LogMgr as ILoggingManager get. set.
    def public prop ErrorMgr as IErrorManager get. set.
    def public prop AuthMgr as IAuthorizationManager
        get. set.
  
```

```

    constructor public BusinessEntity():
        /* managers */
        LogMgr = new LoggingManager().
        ErrorMgr = new ErrorManager().
        AuthMgr = new AuthorizationManager().
    end constructor.
end class.
  
```

```

class Services.CustomerBE inherits BusinessEntity:
    constructor public CustomerBE():
        /* services */
        DataAccess = new CustomerDAO().
    end constructor.
end class.
  
```

Interface

Abstract

Manager

Service

EXTRACT API & BUILDERS

- Create new interfaces and/or abstract parent classes
- Create new Builders and Fluent Interfaces

IDataAccess

DataAccessBuilder

DefaultDABuilder

ILoggingManager

LoggingManagerBuilder

DefaultLogMgrBuilder

IAuthorizationManager

AuthManagerBuilder

DefaultAuthMgrBuilder

IErrorManager

ErrorManagerBuilder

DefaultErrMgrBuilder

Interface

Abstract

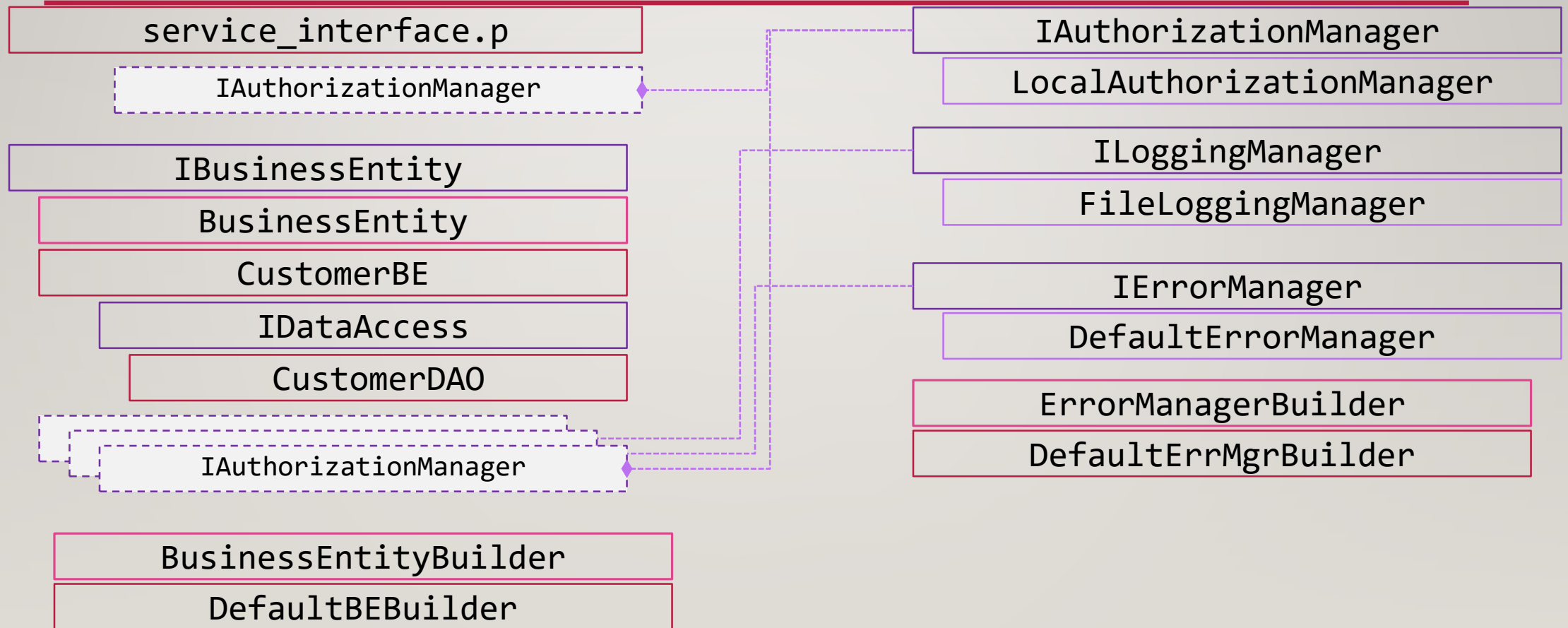
Manager

Service

EXTRACT API & BUILDERS

Services.*

Managers.*



FLUENT INTERFACES



A method for designing object oriented APIs based extensively on method chaining with the goal of making the readability of the source code close to that of ordinary written prose,

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

Based on Fowler and Evans as described at

<https://martinfowler.com/bliki/FluentInterface.html>

FLUENT INTERFACES

- Chain calls together without intermediary variables
- Make OO look like a language - make it more readable
- Often used to build a set of parameters (value objects)

```
def var formData as Progress.Lang.Object no-undo.  
def var req as OpenEdge.Net.HTTP.IHttpRequest.  
  
req = RequestBuilder:Post('http://httpbin.org/post')  
      :ViaProxy('localhost:8888')  
      :WithData(formData, 'multipart/form-data')  
      :AcceptJson()  
      :Request.
```

FLUENT INTERFACES

```
class Services.BusinessEntityBuilder abstract:
  /* Returns a usable BusinessEntity */
  define abstract public property Entity as IBusinessEntity no-undo get.

  /* Starts the chain/fluent interface */
  method static public BusinessEntityBuilder Build(input pcService as character)

  /* 'Language' to modify default behaviour */
  method public BusinessEntityBuilder UseDataAccess (input poDAO as IDataAccess).
  method public BusinessEntityBuilder SupportsLogging (input plSupport as log).
  method public BusinessEntityBuilder UseLogMgr (input poLogMgr as ILoggingManager).
  method public BusinessEntityBuilder SupportsAuthorization(input plSupport as log).
  method public BusinessEntityBuilder UseAuthMgr (
    input poAuthMgr as IAuthorizationManager).
```

Interface

Abstract

Manager

Service

USING A BUILDER

Services.*

service_interface.p

IAuthorizationManager

IBusinessEntity

BusinessEntity

CustomerBE

IDataAccess

CustomerDAO

IAuthorizationManager

```
def input param pcServiceName as char.
def input param pcOperation as char.
def in-out param dataset-handle phServiceData.
def in-out param dataset-handle phServiceParams.
def var oBE as IBusinessEntity.
def var logMgr as IloggingManager.
```

```
oAuthMgr = new AuthorizationManager().
oAuthMgr:AuthorizeServiceOperation(pcServiceName,
pcOperation).
```

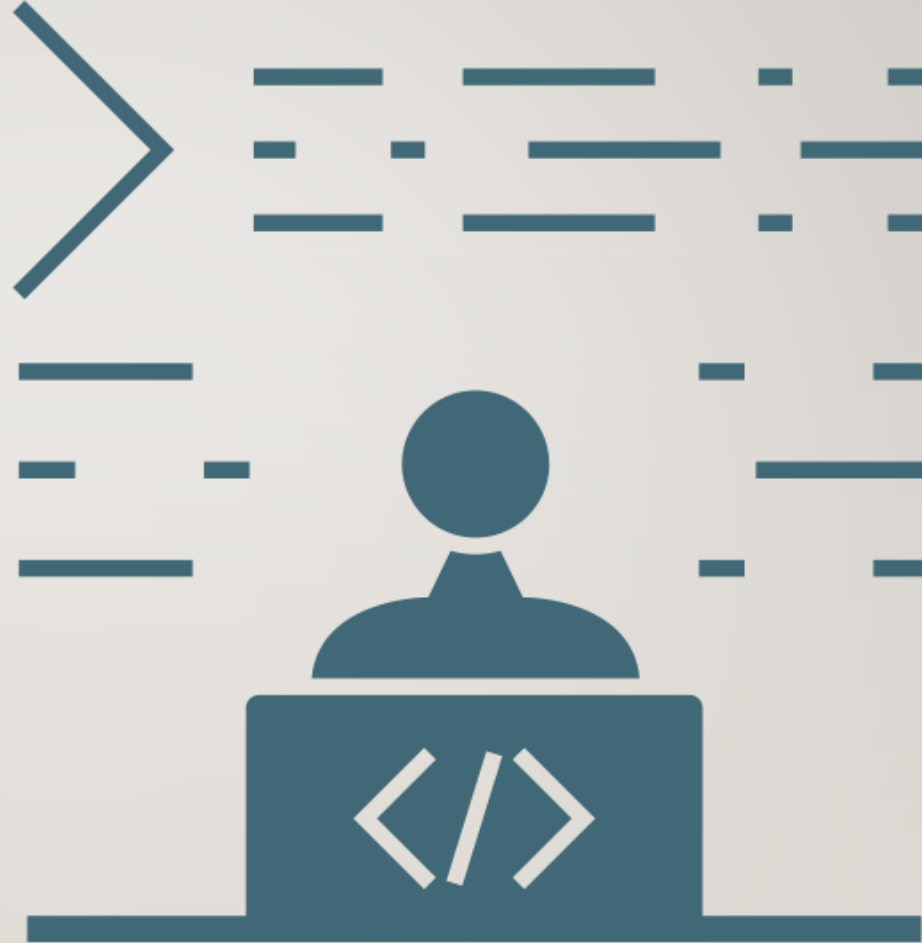
```
logMgr = LogManagerBuilder:Build():Manager.
```

```
oBE = BusinessEntityBuilder
      :Build(pcServiceName)
      :UseLogManager(logMgr)
      :Entity.
```

```
case pcOperation:
  when 'fetch' then oBE:Fetch(<args>).
  when 'save'  then oBE:Save(<args>).
end case.
```

LAB: CREATE A FLUENT INTERFACE

Create a fluent interface for the `BusinessEntityBuilder` class that you have written



BREAK



DEPENDENCY INJECTION



Dependency injection means giving an object its instance variables

<https://www.jamesshore.com/Blog/Dependency-Injection-Demystified.html>



ADDING SUPPORT FOR DATA ACCESS

- Since BE and DA closely linked, BE can create Data Access object

```
class Services.CustomerBE inherits BaseEntity:  
  
    constructor public CustomerBE():  
        this-object:DataAccess = DataAccessBuilder:Build('customer'):DataAccess.  
    end constructor.
```

- Means BE has knowledge of how DA built. This is bad because ...
 1. BE needs to know something about DA that isn't core to BE operation
 2. To use different DA, BE needs changes

INJECT REQUIRED DEPENDENCIES: DATA ACCESS

```
interface Services.IBusinessEntity:  
  def public property DataAccess as IDataAccess get. set  
  method public void Fetch(<params>).  
  method public void Save (<params>).  
end interface.
```

Remove setter from interface

```
class Services.BusinessEntity abstract implements IBusinessEntity:  
  def public property DataAccess as IDataAccess get. private set.  
  constructor public BusinessEntity(poDAO as IDataAccess):  
    DataAccess = poDAO.  
  end constructor.  
end class.
```

Add private setter implementation

```
class Services.CustomerBE inherits BusinessEntity:  
  constructor public CustomerBE(poDAO as IDataAccess):  
    super(poDAO).  
  end constructor.  
end class.
```

ADDING SUPPORT FOR DATA ACCESS

- Our BE now truly only has business (domain) logic in it – it only knows that it has a dependency on the DAO but no further knowledge about it

```
class Services.CustomerBE inherits BaseEntity:
  constructor public CustomerBE(input poDAO as IDataAccess):
    super(poDAO).
  end constructor.

  method override protected void ValidateSave( input dataset-handle phData ):
    define variable hBuffer as handle no-undo.
    hBuffer = phData:get-buffer-handle(1).

    hBuffer:find-first().

    if hBuffer::CustNum le 0 then
      return error new AppError('CustNum must be positive').
    end method.
  end class.
```


BUILDER: INJECT DATA ACCESS INTO BE

```
class Services.DefaultBEBuilder inherits BusinessEntityBuilder:
  define private variable mcServiceName as character no-undo.

  define override public property Entity as IBusinessEntity no-undo
  get():
    define variable oBE as IBusinessEntity no-undo.
    define variable oDAO as IDataAccess no-undo.

    oDAO = DataAccessBuilder:Build(mcServiceName):DataAccess.

    oBE = dynamic-new 'Services.' + mcServiceName + 'BE' (input oDAO).

    return oBE.
  end get.

  constructor public DefaultBEBuilder(input pcServiceName as character):
    assign mcServiceName = pcServiceName.
  end constructor.
end class.
```

BUILDER: ACCEPT EXTERNAL DATA ACCESS

```
class Services.BusinessEntityBuilder abstract:
  def abstract public property Entity as IBusinessEntity no-undo get.

  method static public BusinessEntityBuilder Build(input pcServiceName as
char):
    define variable oBuilder as BusinessEntityBuilder no-undo.
    case pcServiceName:
      /*default */
      otherwise oBuilder = new DefaultBEBuilder(pcServiceName).
    end case.
    return oBuilder.
  end method.

  /* lets us add any DAO to the BE */
  method public BusinessEntityBuilder UseDataAccess(
                                input poDAO as IDataAccess):
    SaveConfig('DAO', poDAO).
    return this-object.
  end method.
end class.
```

BUILDER: USE EXTERNAL DATA ACCESS

```
class Services.DefaultBEBuilder inherits BusinessEntityBuilder:
  define private variable mcServiceName as character no-undo.

  define override public property Entity as IBusinessEntity no-undo
  get():
    define variable oBE as IBusinessEntity no-undo.
    define variable oDAO as IDataAccess no-undo.

    oDAO = GetConfigOption('DAO').
    if not valid-object(oDAO) then
      oDAO = DataAccessBuilder:Build(mcServiceName):DataAccess.

    oBE = dynamic-new 'Services.' + mcServiceName + 'BE' (input oDAO).

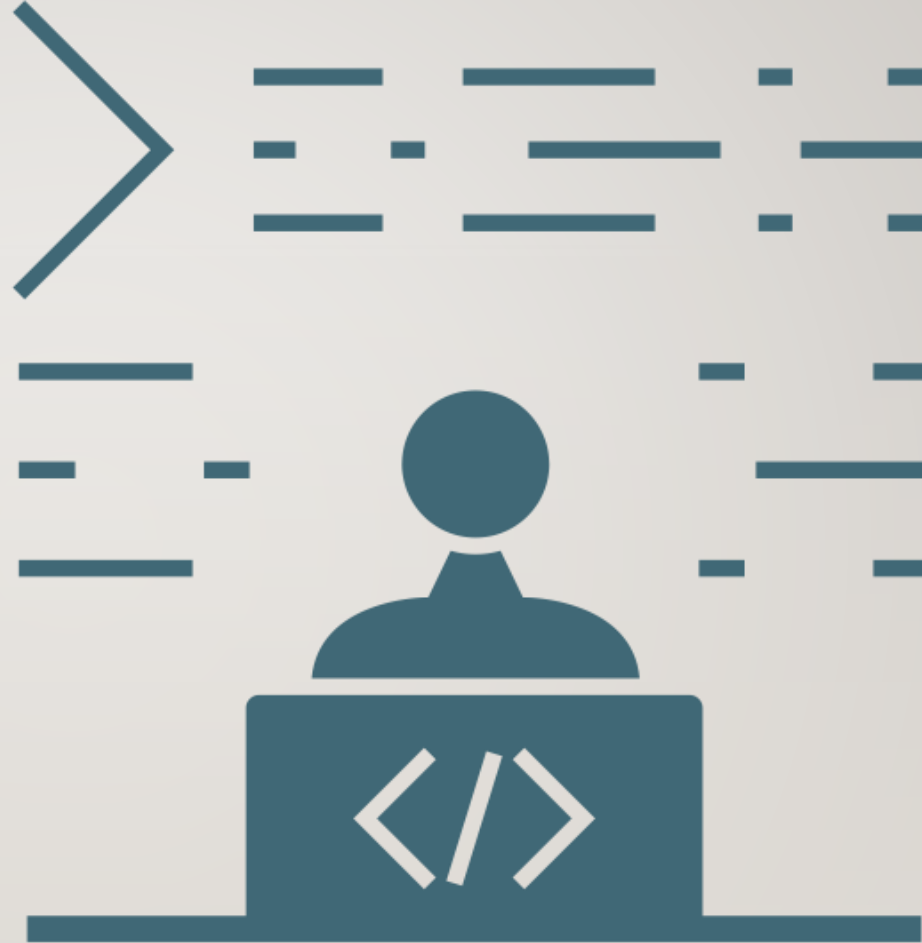
    return oBE.
  end get.

  constructor public DefaultBEBuilder(input pcServiceName as character):
    assign mcServiceName = pcServiceName.
  end constructor.
end class.
```

LAB: INJECT DATA ACCESS

Update the `IBusinessEntity` interface and `BusinessEntity` class to make the `DataAccess` property privately settable

Update the `BusinessEntityBuilder` class you have written to inject a `DataAccess` object



SINGLETONS



The **singleton pattern** is a software design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system.

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



SINGLETONS

TRUE

- Pretty much everyone's first true pattern love
- Use sparingly

FALSE

- Must use STATIC Instance PROPERTY
- The only mechanism for "managers"
- Should never be used
- Should always be used

STATIC-INSTANCE-SINGLETON (IF YOU INSIST)

```
1. class OpenEdge.Example.BusinessLogicManager:
2.     define static public property Instance as BusinessLogicManager no-undo
3.     get():
4.         if not valid-object(BusinessLogicManager:Instance) then
5.             assign BusinessLogicManager:Instance = new BusinessLogicManager().
6.             return BusinessLogicManager:Instance.
7.     end get.
8.     private set.
9.     // make the default constructor PRIVATE to prevent anyone else NEWing
10.    constructor private BusinessLogicManager():
11.    end constructor.
12.    // normal instance-level method to do work
13.    method public void DoStuff(input pArgument as ValueObject):
14.    end method.
15. end class.
16. // Calling mechanism
17. // static property access (could be stored in a variable)
18. //         normal object instance operation
19. BusinessLogicManager:Instance:DoStuff(valueObject).
```

WHY NOT?

- Factory + implementation = singleton in a single class
 - Does not allow replacements of any one of those pieces
 - Does not allow mocking (true for static members in general)
- Object contains application (or component) behaviour and lifecycle
 - Not a good separation of concerns

SERVICE MANAGER



The Service Manager is the infrastructure component that manages the relationship between the abstraction (an OO type name) and the implementing instance (a concrete, instantiable OO type name).

Edu, M., Fechner, M., Prinsloo S. L., Judge, P., & Smith, R. (2016). *Service Manager Specification OpenEdge Application Architecture Specification (OEAA) version 1.0*. Progress Software.

WHAT IS THE COMMON COMPONENT SPECIFICATION (CCS) PROJECT?

- A project for developing standard business application component specifications for business applications, driven by OpenEdge experts and evangelists from the entire Progress Community
- Defines a common understanding and language of an business application architecture
- Enables standards-based framework components that can easily interoperate
- Enables creation of standard tools for those components

CCS OUTPUTS

- Versioned specifications in document form
- Interface definitions – OOABL source code
 - Also includes many interfaces, many enums and one class
 - In 11.7.2 all published interfaces included in OE install

The behavior described in the document and interface(s) MUST be followed in order to claim compatibility with a version of a specification

- May include some sample code or test cases
 - No complete reference implementation as part of the project

<https://github.com/progress/CCS>

SERVICE MANAGER RESPONSIBILITIES

- The Service Manager provides two categories of functionality

1. Service name resolution

```
/* Returns a usable instance of the requested service.  
   @param P.L.Class The service name requested  
   @return P.L.Object A usable instance  
   @throws P.L.AppError Thrown when no implementation can be found */  
method public Progress.Lang.Object getService(  
    input poService as class Progress.Lang.Class).
```

2. Lifecycle management

```
/* Destroys and flushes from any cache(s) objects scoped to the argument scope.  
   @param ILifecycleScope A requested scope for which to stop services. */  
method public void stopServices(  
    input poScope as Ccs.ServiceManager.ILifecycleScope).
```


OBJECT LIFECYCLES (AKA "SCOPES")

As long as an object respects a contract – it implements an interface or inherits from an abstract class - does it matter who created it and how?

Or how long it lives for after an object is done using it?

- How long should an object live?

It could be for the life of the ...

- Session  singleton
- User's login session
- Request
- As long as it is used ("transient")

- How many instances should there be?

COMPLETE CCS.COMMON.ISERVICEMANAGER

```
interface Ccs.Common.IServiceManager inherits Ccs.Common.IManager:
    /* Returns a usable instance of the requested service.
       @param P.L.Class The service name requested
       @return P.L.Object A usable instance */
    method public Progress.Lang.Object getService(input poService as class Progress.Lang.Class).

    /* Returns a usable instance of the requested service.
       @param P.L.Class The service name requested
       @param ILifecycleScope A requested scope. The implementation may choose to ignore this value.
       @return P.L.Object A usable instance */
    method public Progress.Lang.Object getService(input poService as class Progress.Lang.Class,
                                                input poScope as Ccs.ServiceManager.ILifecycleScope).

    /* Returns a usable instance of the requested service.
       @param P.L.Class The service name requested
       @param character An alias for the service. The implementation may choose to ignore this value.
       @return P.L.Object A usable instance */
    method public Progress.Lang.Object getService(input poService as class Progress.Lang.Class,
                                                input pcAlias as character).

    /* Destroys and flushes from any cache(s) objects scoped to the argument scope.
       @param ILifecycleScope A requested scope for which to stop services. */
    method public void stopServices(input poScope as Ccs.ServiceManager.ILifecycleScope).
end interface.
```

USING A SERVICE MANAGER

- How does it know what implementation to return for a requested interface (type)?
 - Use a database- or temp-tables as some form of registry
 - Pattern matching
- Load mappings from
 - JSON or XML
 - Code
- How do you get a reference to it?

CCS.COMMON.APPLICATION CLASS

- The only class CCS will ever publish
- Naming it *Framework* seemed wrong
- Provides access to the Startup Manager
- Provides access to the Service Manager (for convenience)
- Yes – it is like a GLOBAL SHARED variable

... but there didn't seem to be a better way

CCS.COMMON.APPLICATION

```
CLASS Ccs.Common.Application FINAL:
  // Provides access to the injected IStartupManager.
  DEFINE STATIC PUBLIC PROPERTY StartupManager AS Ccs.Common.IStartupManager NO-UNDO GET. SET.

  // Provides access to the injected IServiceManager.
  DEFINE STATIC PUBLIC PROPERTY ServiceManager AS Ccs.Common.IServiceManager NO-UNDO GET. SET.

  // Provides access to the injected ISessionManager.
  DEFINE STATIC PUBLIC PROPERTY SessionManager AS Ccs.Common.ISessionManager NO-UNDO GET. SET.

  // Version of the Common Component Specification implementation.
  DEFINE STATIC PUBLIC PROPERTY Version AS CHARACTER NO-UNDO INITIAL '1.0.0':u
  GET.
  // Prevent creation of instances.
  CONSTRUCTOR PRIVATE Application ():
    SUPER ().
  END CONSTRUCTOR.
END CLASS.
```

USING CCS.COMMON.APPLICATION

```
/* Invokes/instantiates a webhandler */
method private IWebHandler InvokeHandler(input pHandlerName as character):
    define variable webHandler as IWebHandler no-undo.

    assign webHandler = cast(Ccs.Common.Application:ServiceManager
                             :getService(get-class(Progress.Web.IWebHandler),
                                         pHandlerName
                                         ),
                             IWebHandler) no-error.

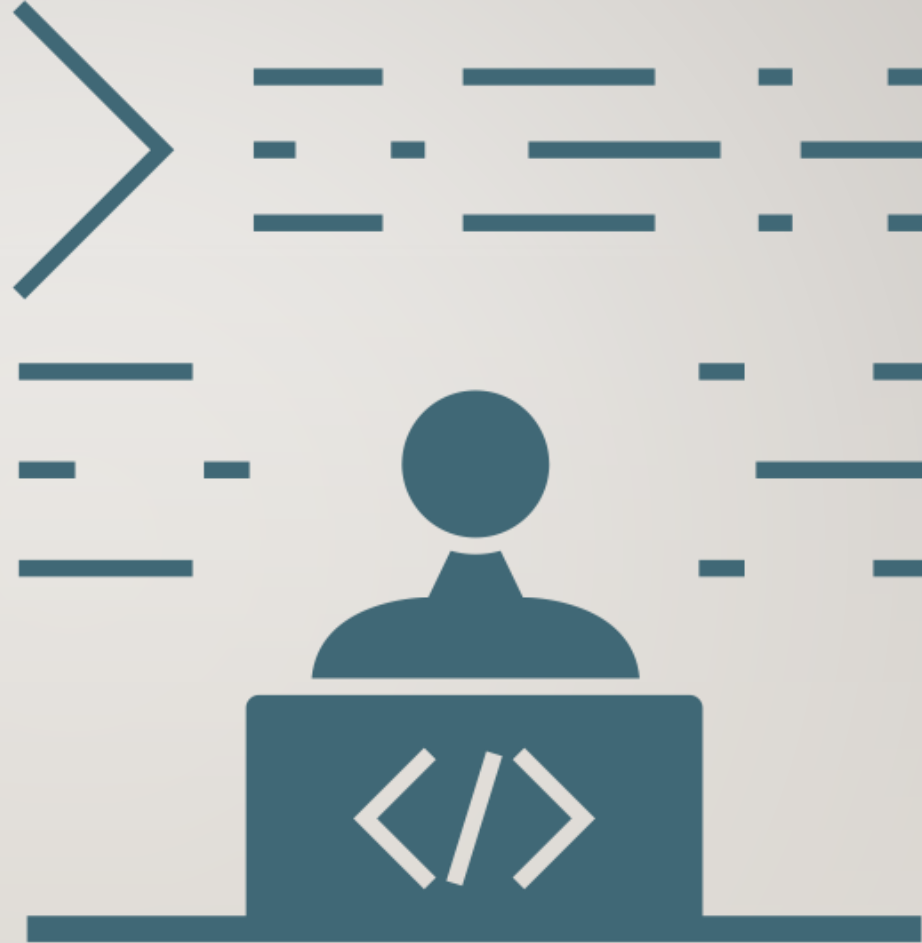
    if not valid-object(webHandler) then
    do:
        webHandler = dynamic-new pHandlerName ().
        if type-of(webHandler, ISupportInitialize) then
            cast(webHandler, ISupportInitialize):Initialize().
        end.
    return webHandler.
end method.
```

LAB: IMPLEMENT A SERVICE MANAGER

Use the CCS IServiceManager interface

Implementation can call the builders already created

Update the service interface calling code to use this implementation



WHAT DID WE LEARN?

- OOABL terminology
- Using Interfaces to provide flexibility
- How to create and use builders
- Managing object lifespans / lifecycles with a Service Manager

USEFUL LINKS / SITES

https://github.com/PeterJudge-PSC/Composing_Complex_Apps

<https://github.com/progress/CCS>

https://github.com/consultingwerk/CCS_Samples



THINGS FOR ANOTHER TIME

- Domain-Driven Design
- Working with relational data
- Generic programming using reflection
- Garbage collection
- Facades / Decorators & Adapters
- "GUI" for .NET



Peter Judge
Mike Fechner

pjudge@progress.com
mike.fechner@consultingwerk.de

