OOABL PATTERNS & PRACTICES

PUG Challenge Americas Workshop

Mike Fechner **Consultingwerk,** mike.fechner@consultingwerk.de

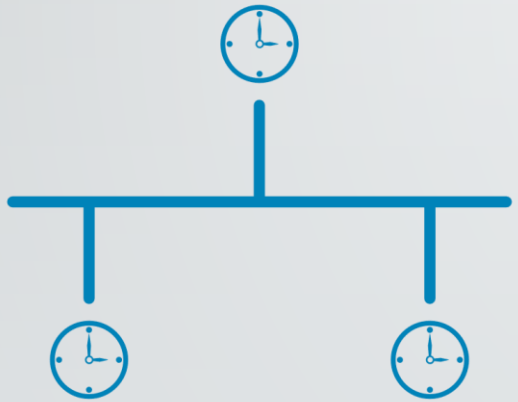
Peter Judge **Progress Software,** pjudge@progress.com

https://github.com/4gl-fanatics

ABSTRACT

- Writing class files is easy. Writing good class files requires a new school of thought – especially for procedural developers. In this session we will discuss about some best practices for the OOABL. We'll be talking about best usage of Enums, Interfaces, Parameter classes and ways to manage dependencies the CCS way. The presenters will be showing OO patterns every developer should know and know when and when not to use them. We'll also be talking about the role of relational concepts like temp-tables and ProDatasets in a class based world.

AGENDA



- OO terms and definitions
- General OOABL coding
  - Procedures
  - Garbage collector
  - Defining variables vs. passing references
  - Using includes
- Lab context: what are we building
- Enumerations
- Value objects
- Static members
- Contract types: interfaces & abstract classes
- Dependency injection
- Service Manager

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, events 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

ACCESS LEVELS

- Defines the visibility of members
- is enforced by the compiler

Best practices

- it's easier to increase access once released than to descrease it
- PROTECTED is a decent default

PRIVATE MEMBERS ARE CLASS-PRIVATE

```
1.  class OpenEdge.Core.String serializable:
2.      /* Holds the actual value, in UTF-8. UTF-8 can hold pretty much
3.         all characters, so let's be smart and use that.
4.         SERIALIZABLE since this is the actual value 'holder' */
5.      define PRIVATE serializable variable mUTF8Value as longchar no-undo.
6.
7.      method override public logical Equals(input p0 as Object):
8.          if type-of(p0, OpenEdge.Core.String) then
9.              return (mUTF8Value eq cast(p0, OpenEdge.Core.String):mUTF8Value).
10.
11.         return false.
12.     end method.
13. end class.

14. // Calling code
15. define variable str1 as OpenEdge.Core.String.
16. define variable str2 as OpenEdge.Core.String.
17.
18. str1 = new OpenEdge.Core.String('a string').
19. str2 = new OpenEdge.Core.String('a string').
20.
21. message  str1:Equals(str2).  // TRUE
22.
23. message (str1:mUTF8Value eq str2:mUTF8Value).// FAILS TO COMPILE
```

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

GENERAL OOABL PROGRAMMING

Procedures Garbage collector

Defining variables vs. passing references Using includes

PROCEDURES & OBJECTS

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

GARBAGE COLLECTOR

- Automatically deletes an instance if there are no references to it being held.
  - Same effect as DELETE OBJECT – runs any destructor
- References are held by
  - Variables, Properties, Temp-table fields
  - Event subscriptions
  - Progress.Lang.Object's NEXT-SIBLING and PREV-SIBLING excluded
  - SESSION:FIRST-OBJECT and FIRST-FORM chains excluded
- References are let go by
  - Variables going out of scope
  - ASSIGN <variable | property > = <some value, including ?>.
  - DELETE OBJECT
  - DELETE temp-table record
- `LOG-MANAGER:LOG-ENTRY-TYPE = 'DynObjects.Class'` shows manual and auto-deletion

GC GOTCHAS

- Circular references
  - Event subscriptions
  - Parent-child with 2-way references
  - Weak references can help
- References held in persistent procs
- References held by static members
- Logging cannot tell where a reference is held

```
1.  class HeaderRecord:
2.    define public property Children as DetailRecord extent no-undo
3.      get. set.
4.
5.    method public void AddDetail(pDetail as DetailRecord):
6.      assign extent(Children)           = extent(Children) + 1
7.             Children[extent(Children)] = pDetail.
8.    end method.
9.  end class.

10. class DetailRecord:
11.   define public property Parent as HeaderRecord no-undo get.
12.
13.   constructor DetailRecord(pParent as HeaderRecord):
14.     this-object:Parent = pParent.
15.   end constructor.
16. end class.

17. // caller
18. def var hdr as HeaderRecord.
19. def var detail as DetailRecord.
20.
21. hdr = new HeaderRecord().
22. detail = new DetailRecord(hdr).
23.
24. hdr:AddDetail(detail).
25. // now we have a circular reference and GC will never clean up.
26. // DELETE OBJECT will cause all of the DetailRecords to be cleaned up
27. delete object hdr.
```

GC OF HANDLES

```
1.  // Available since 11.6.3 in $DLC/[gui|src|tty]/OpenEdge.Core.pl
2.  class OpenEdge.Core.WidgetHandle:
3.    define public property Value as handle no-undo
4.      get.
5.      private set.
6.
7.    // Indicates whether the handle will be destroyed/cleared when this object is destroyed.
8.    define public property AutoDestroy as logical no-undo get. set.
9.
10.   destructor public WidgetHandle():
11.     if AutoDestroy and valid-handle(this-object:Value) then
12.       delete object this-object:Value.
13.   end destructor.
14.
15.   constructor public WidgetHandle(input phValue as handle,
16.                                   input plAutoDestroy as logical):
17.     assign this-object:AutoDestroy = plAutoDestroy
18.            this-object:Value       = phValue.
19.   end constructor.
20.
21.   // Does stuff with the handle
22. end class.
```

CLASS-BASED VARIABLES VS. PASSING PARAMETERS

- Variable scope
- Who creates the instances?

INCLUDES AND CLASSES

- Objects can include includes
- Allow you to emulate generics (imperfectly)
  - Avoid CAST
- Still good for boilerplate code / generation

```
{get-service.i}
1.   CAST (Consultingwerk.Framework.FrameworkSettings:ServiceContainer
2.              :GetService(get-class("{1}":U)),
3.                     {1})
// caller.p
1.   define variable svc as ISomething.
2.   // simple, easy to read
3.   svc = {get-service.i ISomething} .
4.   // as opposed to
5.   svc =
6.    CAST(Consultingwerk.Framework.FrameworkSettings:ServiceContainer
7.              :GetService(get-class(ISomething)),
8.         ISomething).
```

LABS: WHAT ARE WE BUILDING

CODE EXAMPLE

- https://github.com/4gl-fanatics

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. PRIVATE set.
4.    define public property Format   as character no-undo get. PRIVATE set.
5.    define public property Encoding as character no-undo get. PRIVATE 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 Example.Orders.OrderStatusEnum:  //implicitly FINAL and 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.    // A single value can have many names
    // enum members are the only members allowed
    // enum members can only be used in enum types
end enum.
```

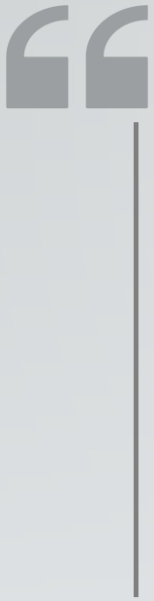
USING ENUMS

```
procedure SetOrderStatus(input pOrderNum as integer,
                         input pStatus   as Example.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).
```

STATIC MEMBERS



A static member is scoped to the defining class type and makes it available for the duration of the session without the need to instantiate a member of that class

USING STATIC MEMBERS

- Easy to use: no NEW required
- Properties, variables, methods, events, temp-tables/datasets, buffers can all be STATIC
- Once loaded, always available for the duration of the session
- Members can hold object references (instances) which may be explicitly destroyed
- Are never part of an interface
- Can never be inherited
- Prevent the class from being part of the reusable object cache

WHAT SHOULD THEY BE USED FOR?

- ✓ Factories and builder methods
- ✓ Session-wide caches
- ✓ To replace GLOBAL SHARED variables
- ✓ Session-wide “anywhere” events

LAB: CONNECTION TO YOUR LAB MACHINE

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

CONNECTION DETAILS

Wifi details

SSID	`Hilton Honors Meeting`
Password	**`PUGAMER19`**

Connecting to the VM

`\Administrator`

`PROworkshop2019`

AMAZON INSTANCES

Name	IP
	54.234.93.67
	3.93.188.25
	54.208.102.46
	34.204.10.173
	3.83.234.140
	18.208.214.3
	54.234.39.30
	34.228.55.149
	54.145.23.7
	54.146.231.235
	52.90.100.91
	54.174.212.81
	34.203.14.154
	54.158.30.235
	54.197.200.208

CODING TO A CONTRACT

Interfaces

Abstract classes

SOFTWARE GOALS

- Loosen dependencies between objects
  - Easier to change/replace/extend behaviour
  - Easier to test (swap out real objects for doppelgängers)
- Extensibility
  - Need capability to add and extend object behaviour
  - May not have ability to change base behaviour (no/encrypted source code)
- Lower the impact of changes

USE CONTRACT TYPES FOR DEFINITIONS, PARAMETERS

- Use them to define the programming interface
  - Compiler requires that implementing/concrete classes fulfill a contract
  - Neither can be instantiated aka NEW'd : application requires a component / code to provide a runnable class that satisfies the contract
- Interfaces preferred
  - Can use multiple at a time
  - Now have *I-won't-break* contract with implementers
- Use inheritance for common or shared behaviour
  - Careful of deep hierarchies – reduces flexibility

CONTRACT TYPES

INTERFACES

- Public contract
- Effectively a header – can never have any implementation
- Can inherit from interfaces and be extended by other interfaces
- Members must be implemented by an implementer

ABSTRACT CLASSES

- Public or protected contract
- May have abstract members that must be implemented in a descendant
- May have concrete / implemented members (ie with behaviour etc)
- Can implement interface(s)
- Can inherit from and be extended by abstract or concrete classes

LAB: INTERFACES AND ABSTRACT CLASSES

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).
```

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.
```

IMPLEMENTING A SERVICE MANAGER

1. How does it know what implementation to return for a requested service (type)?
   - Use a database- or temp-tables as some form of registry?
   - Pattern matching?
2. How does it know how to run (instantiate) an implementation?
   - What if there's no default constructor?
   - What if there are required arguments?
3. How long does object live?
4. How do you get a reference to it?
   - A service manager of service managers?

IMPLEMENTING A SERVICE MANAGER

MAPPING

1. How does it know what implementation to return for a requested service (type)?
   - Use a database- or temp-tables as some form of registry?
   - Pattern matching?

PROVIDERS

2. How does it know how to run (instantiate) an implementation?
   - What if there's no default constructor?
   - What if there are required arguments?

SCOPE

3. How long does object live?

DISCOVERY

4. How do you get a reference to it?
   - A service manager of service managers?

MAPPING: HOW DO WE KNOW WHICH IMPLEMENTATION MATCHES A SERVICE (TYPE)?

- Naming conventions
- Explicit
  - Flat file
  - Database
  - Code

Service (INTERFACE)	Alias	Implementation (CLASS)
`4GlFanatics.UI.IInputFormHelper`	`GUI`	`4GlFanatics.UI.Implementation.GuiInputFormHelper`
`4GlFanatics.UI.IInputFormHelper`	`TTY`	`4GlFanatics.UI.Implementation.TtyInputFormHelper`

PROVIDERS: HOW DO WE INJECT DEPENDENCIES INTO AN OBJECT?

- Injection types: **`CONSTRUCTOR`** , `METHOD` , `PROPERTY`
- General purpose
  - Use the default constructor
  - Picks a constructor via reflection
  - Write once, use often
- Tailored to an implementation type
  - Knows what the dependencies are
  - Knows , and how to inject them: constructor, method, property
  - Less reuse

PROVIDERS: SIMPLE EXAMPLE (NO DEPENDENCIES)

```
1.  class 4GlFanatics.ServiceManager.Provider.DefaultConstructorProvider implements IProvider:
2.      /* Instantiates (NEWs) a class. */
3.      method public Progress.Lang.Object invokeService(pImplType as Progress.Lang.Class,
4.                                                        pContext as Progress.Lang.Object):
5.          define variable svc as Progress.Lang.Object no-undo.
6.          Assert:NotNull(pImplType, 'Implementation type').
7.          Assert:NotAbstract(pImplType).
8.          Assert:NotInterface(pImplType).
9.
10.         svc = pImplType:New().
11.
12.         if type-of(svc, IService) then
13.             cast(svc, IService):initialize().
14.
15.         return svc.
16.     end method.
17. end class.
```

SCOPE: MANAGING THE IMPLEMENTATION'S LIFESPAN

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

  singleton
- How many instances should there be?

DISCOVERY: 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

```
// session_start.p
using 4GlFanatics.ServiceManager.*.

// start & init the IServiceManager implementation
Ccs.Common.Application:ServiceManager = new ConfigFileSvcMgr().
Ccs.Common.Application:ServiceManager:initialize().
```

```
using 4GlFanatics.UI.*.
define variable svc as Progress.Lang.Object no-undo.
define variable ifh as IInputFormHelper no-undo.

svc = Ccs.Common.Application:ServiceManager
                            :getService(get-class(IInputFormHelper)).
ifh = cast(svc, IInputFormHelper).
// use the Input Form Helper
```

LAB: IMPLEMENT A SERVICE MANAGER

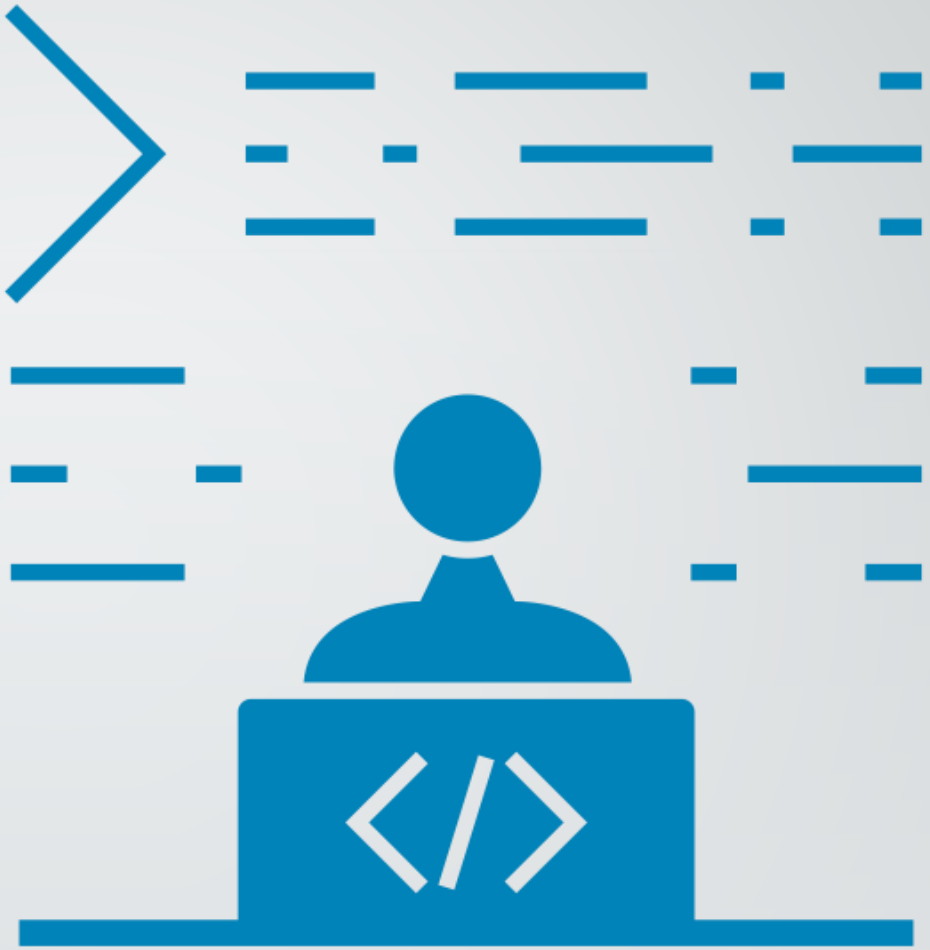
Use the CCS IServiceManager interface

Mapping: file-based or naming-convention is good

Providers: make swappable (reusable)

Discovery: use the CCS Application class

Scope: add caching



WHAT DID WE LEARN?

- OOABL terminology
- What is garbage collection
- Using (immutable) value objects
- Enumerations
- Using interfaces & abstracts to create plug-n-play components

USEFUL LINKS / SITES

https://github.com/4gl-fanatics

https://github.com/progress/CCS

Peter Judge pjudge@progress.com
Mike Fechner mike.fechner@consultingwerk.de