Advanced OOABL Design Patterns

Working with data

Peter Judge / Roland de Pijper
pjudge@progress.com / rpy@progress.com

Agenda

Why patterns?

Interface-first design

Active record

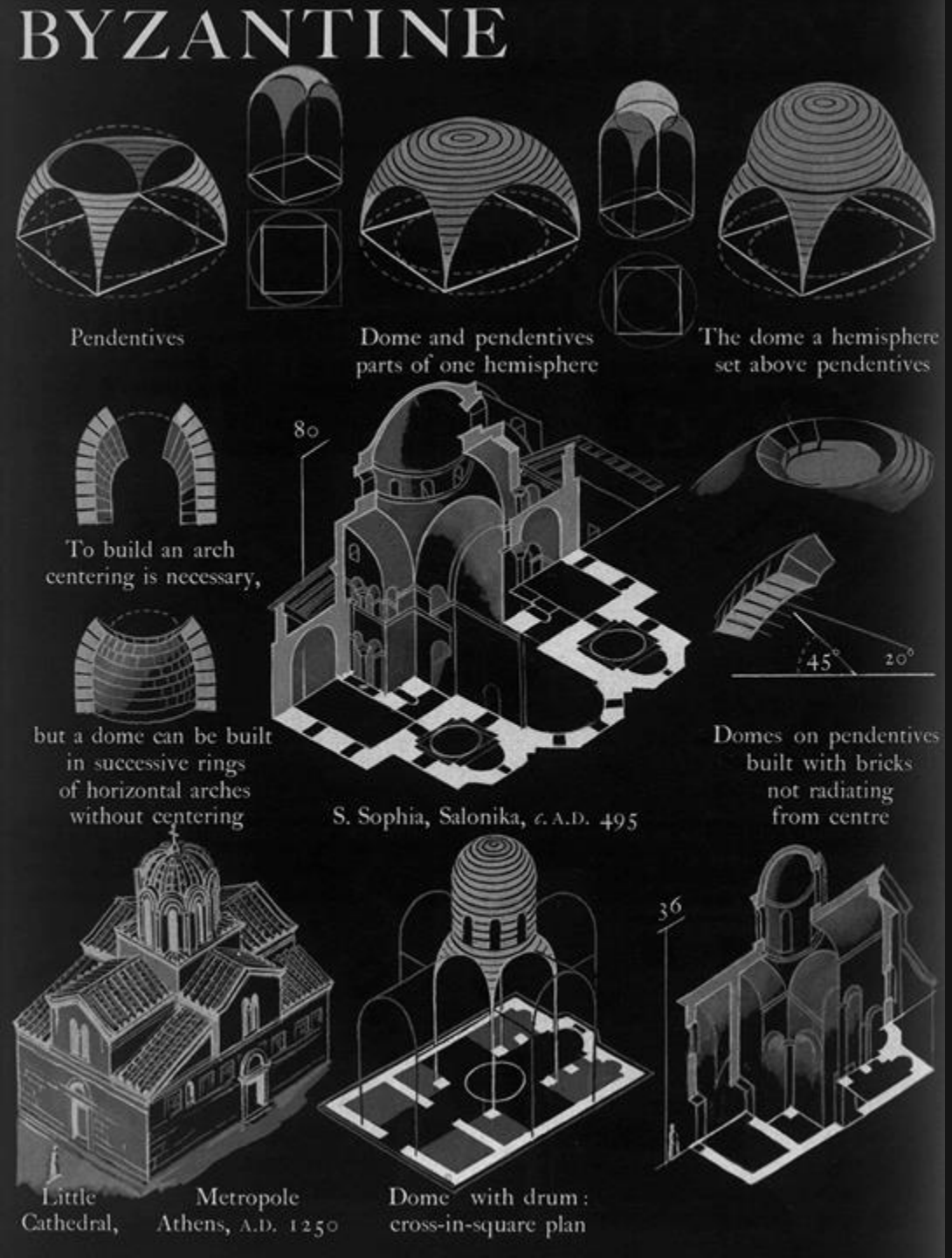
Data Mapper

Collections and Iterator

Decorator / Façade

Adapter

What are patterns and why – a recap



> Patterns expose knowledge about software construction that has been gained by many experts over many years. All work on patterns should therefore focus on making this precious resource widely available. Every software developer should be able to use patterns effectively when building software systems. When this is achieved, we will be able to celebrate the human intelligence that patterns reflect, both in each individual pattern and in all patterns in their entirety.

http://www.bradapp.com/docs/patterns-intro.html

Reminder: driven to abstraction

Use 'contract' types in 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

Software goals

- Flexibility
  - Implementations can be swapped without changes to the calling/consuming code
    - Many implementations of an interface possible
  - Testing (mocking) becomes easier

- Modularity & extensibility
  - Skeletons/frameworks and their pluggable modules can be developed in isolation
  - Implementations can be added later and/or developed elsewheree
  - Requires a mechanism for getting a implementations

Active Record

> An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data

https://martinfowler.com/eaaCatalog/activeRecord.html

Active Record pattern

```
class data.s2k.DepartmentRecord:
  // DATABASE FIELD Department.DeptCode
  define public property DeptCode as character no-undo
    get. set.

  // DATABASE FIELD Department.DeptName
  define public property DeptName as character no-undo
    get. set.

  // CALCULATED FIELD: The average employee tenure/months
  define public property AvgEmpTenure as decimal no-undo
    get. set.

end class.
```

Active Record

- Simple value holder
- Properties give strongly-typed access to fields
  - Allow different access levels for Read (get) and Write (set)
- Separates where the data lives from where it's used ….
  …. like a temp-table
  - Only more compiler help
  - Better scoping
- You will end up with many objects/instances

```
-reusableObjects <very-large-number>
```

Consuming the Active Record

```
1.  // read the data from the DB
2.  find first Department no-lock.
3.  assign dept          = new data.s2k.DepartmentRecord()
4.         dept:DeptCode = Department.DeptCode
5.         dept:DeptName = Department.DeptName.
6.  // calculate average tenure
7.  for each Employee where Employee.DeptCode eq Department.DeptCode no-lock:
8.    assign numEmps = numEmps + 1
9.           totAge  = totAge + interval(today, Employee.Startdate, 'months').
10. end.
11. assign dept:AvgEmpTenure = (totAge / numEmps).
12.
13. // application/business logic does Stuff
14. message dept:DeptCode       // 100
15.         dept:AvgEmpTenure   // 236.29
16. assign dept:DeptName = 'Department of One Hundred'.
17.
18. // write any changes to the DB
19. find first Department where Department.DeptCode eq dept:DeptCode exclusive-lock.
20. assign Department.DeptCode = dept:DeptCode
21.        Department.DeptName = dept:DeptName
```

Create the active record instance

This is where our domain / business logic happens

Separation of concerns

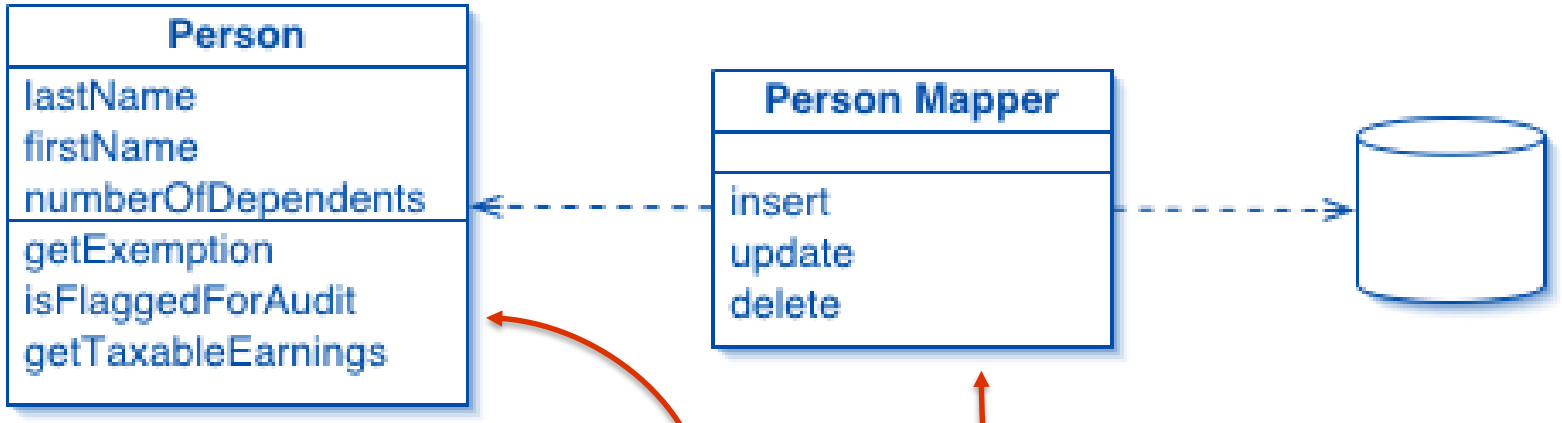
```
1.  // read the data from the DB
2.  find first Department no-lock.
3.  assign dept          = new data.s2k.DepartmentRecord()
4.         dept:DeptCode = Department.DeptCode
5.         dept:DeptName = Department.DeptName.
6.  // calculate average tenure
7.  for each Employee where Employee.DeptCode eq Department.DeptCode no-lock:
8.    assign numEmps = numEmps + 1
9.           totAge  = totAge + interval(today, Employee.Startdate, 'months').
10. end.
11. assign dept:AvgEmpTenure = (totAge / numEmps).
12.
13. // application/business logic does Stuff
14. message dept:DeptCode       // 100
15.         dept:AvgEmpTenure   // 236.29
16. assign dept:DeptName = 'Department of One Hundred'.
17.
18. // write any changes to the DB
19. find first Department where Department.DeptCode eq dept:DeptCode exclusive-lock.
20. assign Department.DeptCode = dept:DeptCode
21.        Department.DeptName = dept:DeptName
```

Data Mapper

> A layer of Mappers that moves data between objects and a database while keeping them independent of each other and the mapper itself.

https://martinfowler.com/eaaCatalog/dataMapper.html

Active Record + Data Mapper in OERA



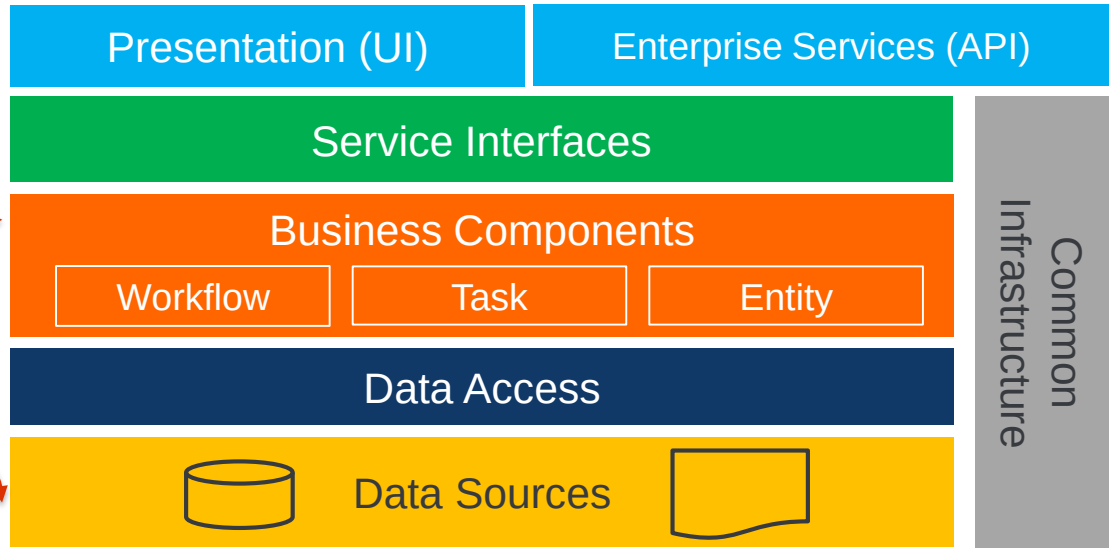
ActiveRecord

logical data /

business logic / entities

DataMapper

knows where data is and
how to read / write it



Data Mapper : Department

```
1.  class data.s2k.DepartmentMapper:
2.    //Read from the persistent store/database
3.    method public class DepartmentRecord Get(input pWhere as character):
4.      buffer Department:find-first(pWhere, no-lock).
5.      assign dept          = new data.s2k.DepartmentRecord()
6.             dept:DeptCode = Department.DeptCode
7.             dept:DeptName = Department.DeptName.
8.      // Calculate field value
9.      for each Employee where Employee.DeptCode eq Department.DeptCode no-lock:
10.       assign numEmps = numEmps + 1
11.              totAge  = totAge + interval(today, Employee.Startdate, 'months').
12.     end.
13.     assign dept:AvgEmpTenure = (totAge / numEmps).
14.     return dept.
15.   end method.
16.
17.   //Create a new Department record . Other CRUDs are left out for space reasons
18.   method public void Create(input pDept as class DepartmentRecord):
19.     create Department.
20.     assign Department.DeptCode = pDept:DeptCode
21.            Department.DeptName = pDept:DeptName.
22.   end method.
```

The Mapper creates and populates the ActiveRecord

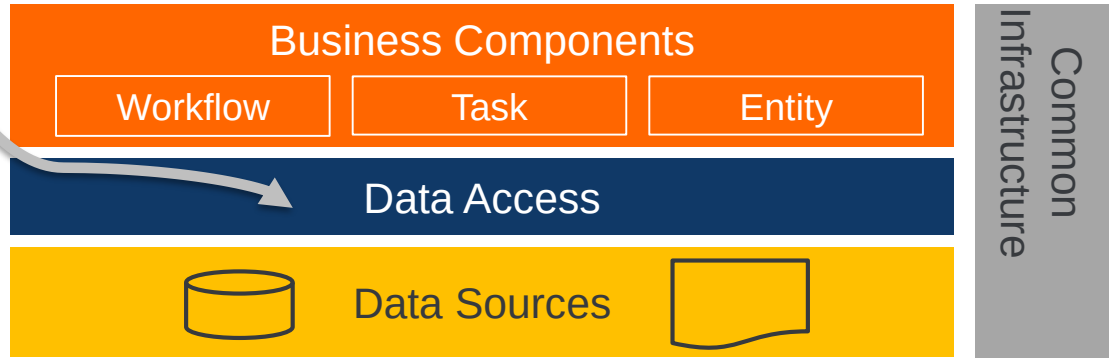
Assign calculate field values

Consuming the mapper

```
def var dept   as class data.s2k.DepartmentRecord.
def var mapper as class data.s2k.DepartmentMapper.


// read the data from the DB
assign mapper  = new data.s2k.DepartmentMapper()
       dept    = mapper:Get('DeptCode eq "100"').

// application/business logic does Stuff
message dept:DeptCode       // 100
        dept:AvgEmpTenure   // 236.29

assign dept:DeptName = 'Department of One Hundred'.

// write any changes to the DB
mapper:Update(dept).
```

A Data Access layer provides ActiveRecord instances to business logic

- Knows which mapper(s) to use



Improving our mapper

- ✓ Extract interfaces from the Mapper and Record types
  - Allows us to keep existing implementations
  - Allows us to write general framework code around mappers
  - Allows us to write custom, complex mappers
- ✓ Add a buffer-based implementation
  - A single Mapper with parameters can be used by many/most/all mappers
  - Can map to TEMP-TABLES or DB buffer
- ✓ Implement default property-to-field mappings through reflection

`DYNAMIC-PROPERTY`	OE11.0.0+
`Progress.Reflect.*`	OE11.4.0+

Sample IMapper

```
interface data.shared.IMapper:

  /* what type of IRecord do we create? Eg. DepartmentRecord */
  define public property RecordType as class Progress.Lang.Class no-undo get.

  /* Read from the persistent store/database */
  method public class data.shared.IRecord    Get(input pWhere as character).

  /* Creates new records in the persistent store/database */
  method public void Create(input pData as class IRecord).

  /* Deletes records from the persistent store/database */
  method public void Delete(input pData as class IRecord).

  /* Updates records in the persistent store/database */
  method public void Update(input pData as class IRecord).

end interface.
```

BufferMapper :: IMapper


```
class data.shared.BufferMapper implements IMapper:
  define public property RecordType as class Progress.Lang.Class no-undo  get. protected set.
  // the underlying buffer
  define protected property DataBuffer as handle no-undo get. private set.
  /* Constructor */
  constructor public BufferMapper(input pBuffer as handle,
                                  input pRecordType as class Progress.Lang.Class):
    this-object:DataBuffer = pBuffer.
    this-object:RecordType = pRecordType.
  end constructor.
  // Remove a record
  method public void Delete(input pData as class IRecord):
    do transaction:
      // simplified FIND. Doesn't deal with multiple key fields and non-char values
      DataBuffer:find-first(substitute('where &1 = "&2"',
                                        DataBuffer:keys, dynamic-property(pData, DataBuffer:keys)),
                 exclusive-lock).
      DataBuffer:buffer-delete().
      finally:
        DataBuffer:buffer-release().
      end finally.
    end.    // trans
  end method.
```

This is the data source (DB or TT)

BufferMapper :: IMapper


```
25.    //Read from the persistent store/database
26.    method public class IRecord Get(input pWhere as character):
27.      DataBuffer:find-first(pWhere, no-lock).
28.      data = cast(RecordType:New(), IRecord).
29.      props = RecordType:GetProperties((Flags:Public or Flags:Instance)).
30.      cnt = extent(props).
31.      do loop = 1 to cnt:
32.        if not props[loop]:SetterAccessMode eq AccessMode:Public then
33.          next.
34.        // assumes no arrays, names are identical
35.        fld = DataBuffer:buffer-field(props[loop]:Name) no-error.
36.        dynamic-property(data, props[loop]:Name) =
37.                                DataBuffer:buffer-field(props[loop]:Name):buffer-value.
38.      end.
39.      return data.
40.   end method.
41.  end class.
```

Maps a field to and from a property

Department DAO

```
1.  class data.s2k.DepartmentDAO:
2.      define public property DepartmentMapper as IMapper no-undo get. private set.
3.      // Injectables
4.      constructor public DepartmentDAO(input pDeptMapper as IMapper):
5.          this-object:DepartmentMapper = pDeptMapper.
6.      end constructor.
7.
8.      // Defaults: probably not a good idea for real life
9.      constructor public DepartmentDAO():
10.         this-object(new BufferMapper(buffer Department:handle, get-class(DepartmentRecord))).
11.     end constructor.
12.     // Example read
13.     method public DepartmentRecord Get(input pWhere as character):
14.         recDept = cast(DepartmentMapper:Get(pWhere), DepartmentRecord).
15.
16.         return recDept.
17.     end method.
18.     // Example update method
19.     method public void Update(input pData as class DepartmentRecord):
20.         DepartmentMapper:Update(pData).
21.     end method.
22. end class.
```

We can change where the data is stored at runtime

Consuming records

```
def var dept       as class data.s2k.DepartmentRecord.
def var deptDAO    as class data.s2k.DepartmentDAO.


assign deptDAO = new data.s2k.DepartmentDAO()
       dept    = deptDAO:Get('where DeptCode eq "100"')

// application/business logic does Stuff
message dept:DeptCode       // 100
        dept:AvgEmpTenure   // 236.29
        .
assign dept:DeptName = 'Department of One Hundred'.

deptDAO:Update(dept).
```

Populate data from mapper

Collections & Iterators

> [A] grouping of some variable number of data items (possibly zero) that have some shared significance to the problem being solved and need to be operated upon together in some controlled fashion.
>
> https://en.wikipedia.org/wiki/Collection_(abstract_data_type)

Collections

A set or group of data

- [OOABL] Objects
- [RDBMS] Records / rows

They have some relationship ("shared significance")

- [OOABL] Common ancestor / interface
- [RDBMS] Same (temp-) table

A cursor is used to navigate/traverse over the collection

- [OOABL] Iteration or Enumerator objects
- [RDBMS] Queries

Returning collections from the mapper

```
class data.s2k.DepartmentMapper:
  //Read from the persistent store/database
  method public class DepartmentRecord Get(input pWhere as character).

  //Read from the persistent store/database
  method public class DepartmentRecord extent                GetAll(input pWhere as
  method public class DepartmentRecordCollection             GetAll(input pWhere as character):
  method public class DepartmentRecordMap                    GetAll(input pWhere as character):
```

Simple, no extra stuff

May be ordered

No consistent iteration

No quick find

```
  //Create a new Department record . Other CRUDs are left out for space reasons
  method public void Create(input pDept as class DepartmentRecord):
    create Department.
    assign Department.DeptCode = pDept:DeptCode
           Department.DeptName = pDept:DeptName.
  end method.
```

Single object stored

May be ordered or unique

Have to inspect to find key val (no quick find)

No order

Is unique

Lookups on a key

Collections

Parent package: OpenEdge.Core

OpenEdge.Core.Collections

Interfaces

OpenEdge.Core.Collections.ICollection	A collection represents a group of objects, known as its elements.
OpenEdge.Core.Collections.IIterable	
OpenEdge.Core.Collections.IIterator	traverses a collection forward
OpenEdge.Core.Collections.IList	
OpenEdge.Core.Collections.IListIterator	An iterator for lists that can traverse the list in both directions
OpenEdge.Core.Collections.IMap	
OpenEdge.Core.Collections.IMapEntry	A map entry (key-value pair). The IMap:EntrySet returns a set-view of the map, whose elements are of this class.
OpenEdge.Core.Collections.ISet	A collection that contains no duplicate elements.
OpenEdge.Core.Collections.IStringCollection	Interface defining a typed String Collection
OpenEdge.Core.Collections.IStringKeyedMap	A typed String/String Map
OpenEdge.Core.Collections.IStringStringMap	A typed String/String Map

Group of objects

General cursor

Keys / values grp

Classes

AbstractTTCollection

Abstract class that holds the collection and its items

- Available in $DLC/src|tty|gui/OpenEdge.Core.pl
- API doc at https://documentation.progress.com/output/oehttpclient/117/

Populating a collection in mapper

```
1.  method public class DepartmentRecordCollection GetAll(input pWhere as character):
2.    query qryDept:query-prepare('preselect each Department where ' + pWhere + ' no-lock').
3.    query qryDept:query-open().
4.    data = new DepartmentRecordCollection().
5.    query qryDept:get-first().
6.    do while not query qryDept:query-off-end:
7.      dept          = new data.s2k.DepartmentRecord().
8.      dept:DeptCode = Department.DeptCode.
9.      dept:DeptName = Department.DeptName.
10.     data:Add(dept).
11.     // Calculate field value
12.     for each Employee where Employee.DeptCode eq Department.DeptCode no-lock:
13.       assign numEmps = numEmps + 1
14.              totAge  = totAge  + interval(today, Employee.Startdate, 'months').
15.     end.
16.     assign dept:AvgEmpTenure = (totAge / numEmps).
17.     query qryDept:get-next().
18.   end.
19.   return data.
20. end method.
```

Create a new collection object

Simply add the ActiveRecord objects; no need to count here

Return to the caller

Collections enable ActiveRecord children

```
class data.s2k.DepartmentRecord:
  // DATABASE FIELD Department.DeptCode
  define public property DeptCode as character no-undo
    get. set.

  // DATABASE FIELD Department.DeptName
  define public property DeptName as character no-undo
    get. set.

  // CALCULATED FIELD: The average employee tenure/months
  define public property AvgEmpTenure as decimal no-undo
    get. set.

 // CHILD RECORDS
  define public property Employees as EmployeeRecordCollection no-undo
    get. set.

end class.
```

Child records

Updated IMapper

```
1.  interface data.shared.IMapper:
2.
3.    /* what type of IRecord do we create? Eg. DepartmentRecord */
4.    define public property RecordType as class Progress.Lang.Class no-undo get.
5.
6.    /* Read from the persistent store/database */
7.    method public class data.shared.IRecord    Get(input pWhere as character).
8.    method public class ICollection         GetAll(input pWhere as character).
9.
10.   /* Creates new records in the persistent store/database */
11.   method public void Create(input pData as class IRecord).
12.   method public void Create(input pData as class ICollection).
13.
14.   /* Deletes records from the persistent store/database */
15.   method public void Delete(input pData as class IRecord).
16.   method public void Delete(input pData as class ICollection).
17.
18.   /* Updates records in the persistent store/database */
19.   method public void Update(input pData as class IRecord).
20.   method public void Update(input pData as class ICollection).
21.
22. end interface.
```

Retrieve many records

Update many records

Department DAO

```
1.  class data.s2k.DepartmentDAO:
2.      define public property DepartmentMapper as IMapper no-undo get. private set.
3.      define public property EmployeeMapper   as IMapper no-undo get. private set.
4.      // Injectables
5.      constructor public DepartmentDAO(input pDeptMapper as IMapper, input pEmpMapper as IMapper):
6.          this-object:DepartmentMapper = pDeptMapper.
7.          this-object:EmployeeMapper   = pEmpMapper.
8.      end constructor.
9.      // Defaults: probably not a good idea for real life
10.     constructor public DepartmentDAO():
11.         this-object(new BufferMapper(buffer Department:handle, get-class(DepartmentRecord)),
12.                     new BufferMapper(buffer Employee:handle,   get-class(EmployeeRecord)) ).
13.     end constructor.
14.     // Example read
15.     method public DepartmentRecord Get(input pWhere as character):
16.         recDept           = DepartmentMapper:Get(pWhere).
17.         recDept:Employees = EmployeeMapper:GetAll(pWhere).
18.         return recDept.
19.     end method.
20.     // Example update method
21.     method public void Update(input pData as class DepartmentRecord):
22.         DepartmentMapper:Update(pData).
23.     end method.
24. end class.
```

Child record mapper

Get the parent and child data

Consuming records … still looks the same

```
def var dept       as class data.s2k.DepartmentRecord.
def var deptDAO    as class data.s2k.DepartmentDAO.


assign deptDAO = new data.s2k.DepartmentDAO()
       dept    = deptDAO:Get('where DeptCode eq "100"')

// application/business logic does Stuff
message dept:DeptCode       // 100
        dept:AvgEmpTenure   // 236.29
        dept:Employees:Size // 7
assign dept:DeptName = 'Department of One Hundred'.

// write any changes to the DB
mapper:Update(dept).
```

But now we have Employees

Making record properties read-only

```
class data.s2k.DepartmentRecord:
  // DATABASE FIELD Department.DeptCode
  define public property DeptCode as character no-undo
    get. set.

  // DATABASE FIELD Department.DeptName
  define public property DeptName as character no-undo
    get. set.

  // CALCULATED FIELD: The average employee tenure/months
  define public property AvgEmpTenure as decimal no-undo
    get.

 // CHILD RECORDS
  define public property Employees as EmpoyeeRecordCollection no-undo
    get. set.

end class.
```

Calculated/derived field

Read-only property, from a collection


```
class data.s2k.DepartmentRecord:
  // DATABASE FIELD Department.DeptCode
  define public property DeptCode as character no-undo
    get. set.

  // DATABASE FIELD Department.DeptName
  define public property DeptName as character no-undo
    get. set.

  // CHILD RECORDS
  define public property Employees as EmpoyeeRecordCollection no-undo
    get. set.

  // CALCULATED FIELD: The average employee tenure/months
  define public property AvgEmpTenure as decimal no-undo
    get():
      // sadly we can't define the property as DECIMALS 2
      return CalculateTenure('months').
    end get.
end class.
```

Read-only property, from a collection

2/2

```
method protected decimal CalculateTenure(input pUnit as character):
  define variable totTenure as integer no-undo.
  // define the variable we return as DECIMALS 2
  define variable avgTenure as decimal decimals 2 no-undo.
  define variable iterator as IIterator no-undo.

  if this-object:Employees:Size eq 0 then
    avgTenure = 0.00.
  else
  do:
      iterator = this-object:Employees:Iterator().
      do while iterator:HasNext():
        totTenure = totTenure
                  + interval(today, cast(iterator:Next(), EmployeeRecord):StartDate, pUnit).
      end.
      avgTenure = (totTenure / this-object:Employees:Size).
  end.
  return avgTenure.
end method.
```

Decorators

> **Decorator** is used to add more gunpowder to your objects (note the term objects -- you typically decorate objects dynamically at runtime). You do not hide/impair the existing interfaces of the object but simply extend it at runtime.

https://stackoverflow.com/a/3489187/18177

Adding more to a mapper

Record Transaction Scope	Defines transaction scope for an update operation
Collection Transaction Scope	
Authorized Buffer Operation	Adds authorization for a data operation
Logging Mapper	Logs events

```
interface ISupportAuthorization
  define public property AuthMgr as IAuthorizationManager get. set.

interface OpenEdge.Logging.ISupportLogging
  define public property Logger as OpenEdge.Logging.ILogWriter get. set.
```

Adding functionality using inheritance

```
class Department<auth|log|auth-log|…>Mapper inherits Mapper
    1.  implements <none>
    2.  implements ISupportAuthorization
    3.  implements                       ISupportLogging
    4.  implements ISupportAuthorization, ISupportLogging

class DepartmentMapper inherits <Mapper|CollectionTransaction>
    1.  inherits <none>
    2.  inherits Mapper
    3.  inherits        CollectionTransation
    4.  inherits Mapper, CollectionTransation
```

Adding functionality using inheritance

```
class Depart
    1. imple
    2. imple
    3. imple
    4. imple

class Depart
    1. inher
    2. inher
    3. inher
    4. inher
```

Adding support for optional dependencies

Challenge is supporting zero, one or more of these optional dependencies

1. EITHER Implement interface in Mapper superclass

   All Mappers get this behaviour

```
class data.shared.Mapper abstract implements IMapper, ISupportLogging:
```

2. OR Implement interface in individual Mapper

   Only this Mapper gets this behaviour

```
class data.s2k.DepartmentMapper inherits Mapper implements ISupportLogging:
```

3. OR Implement interface in a Decorator

   Only certain Mappers get this behaviour

Our MapperDecorator class

```
class data.shared.MapperDecorator abstract implements IMapper:
    // the IMapper being decorated
    define protected property DecoratedMapper as IMapper no-undo get. private set.
    // We MUST get at least the decorated object via Ctor
    constructor public MapperDecorator(input pMapper as IMapper):
      assign DecoratedMapper = pMapper.
    end constructor.
    // We must implement all of the IMapper interface
    define public property RecordType as class Progress.Lang.Class no-undo
        get():
            return DecoratedMapper:RecordType.
        end get.
    method public void Create(input pData as IRecord):
        DecoratedMapper:Create(pData).
    end method.
    method public void Create(input pData as ICollection):
        DecoratedMapper:Create(pData).
    end method.
end class.
```

Authorized Buffer Operation

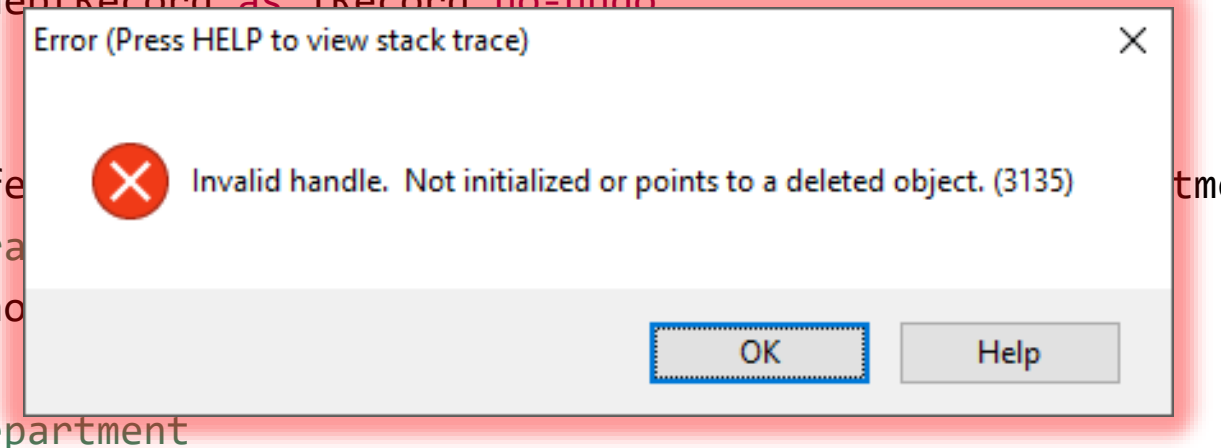
```
class data.shared.AuthorisedBufferOperation
                inherits   MapperDecorator              // behaviour from IMapper
                implements ISupportAuthorization:       // Adds Authorization

    define public property AuthManager as IAuthorizationManager no-undo get. set.

    constructor public AuthorisedBufferOperation (input pMapper as IMapper):
        super (input pMapper).
    end constructor.

    method override public void Delete( input pData as IRecord ):
        if AuthManager:AuthorizeOperation('delete+' + pData:GetClass():TypeName) then
            super:Delete(input pData).
    end method.

end class.
```

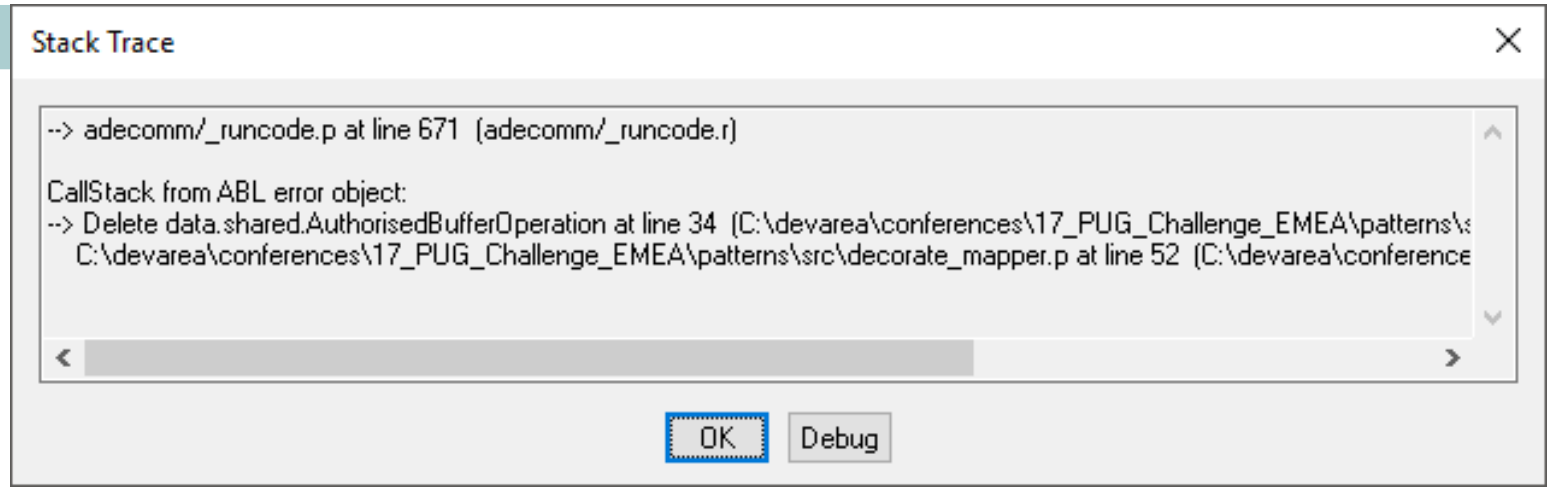
Building a decorated Mapper

```
define variable mapper as IMapper no-undo.
define variable deptRecord as IRecord no-undo.

// base IMapper
mapper = new BufferMapper(buffer Department:handle, get-class(DepartmentRecord)).
// add auth decorator for deletes
mapper = new AuthorisedBufferOperation(mapper).

// Delete this department
deptRecord = mapper:Get('DeptCode eq "100"').
mapper:Delete(deptRecord).
```

Building a decorated Mapper

```
1.  define variable mapper as IMapper no-undo.
2.  define variable deptRecord as IRecord no-undo.
3.
4.  // base IMapper
5.  mapper = new Buffe                                  tmentRecord)).
6.  // add auth decora
7.  mapper = new Autho
8.
9.  // Delete this department
10. deptRecord = mapper:Get('DeptCode eq "100"').
11. mapper:Delete(deptRecord).
```





Adapters

> **Adapter** adapts a given class/object to a new interface. In the case of the former, multiple inheritance is typically employed. In the latter case, the object is wrapped by a conforming adapter object and passed around. The problem we are solving here is that of non-compatible interfaces.

https://stackoverflow.com/a/3489187/18177

Building a decorated Mapper

```
1. define variable mapper as IMapper no-undo.
2. define variable deptRecord as IRecord no-undo.
3.
4. // base IMapper
5. mapper = new BufferMapper(buffer Department:handle, get-class(DepartmentRecord)).
6.
7. // add auth on deletes
8. mapper = new AuthorisedBufferOperation(mapper).
9. // Set the AuthManager in the mapper
10. cast(mapper, ISupportAuthorization):AuthManager = new common.shared.AuthorizationManager().
11.
12. // Delete this department
13. deptRecord  = mapper:Get('DeptCode eq "100"').
14. mapper:Delete(deptRecord).
```

What about multiple decorations?

```
function BuildMapper returns IMapper():
  return new CollectionTransactionScope(
             new LoggingMapper(
                 new AuthorisedBufferOperation(
                     new BufferMapper(buffer Department:handle, get-class(DepartmentRecord)).
end function.

define variable mapper as IMapper no-undo.
define variable deptRecord as IRecord no-undo.

mapper = BuildMapper().
// Set the AuthManager in the mapper
cast(mapper, ISupportAuthorization):AuthManager = new common.shared.AuthorizationManager().

// Delete this department
deptRecord  = mapper:Get('DeptCode eq "100"').
mapper:Delete(deptRecord).
```

Adapters

```
/*------------------------------------------------------------------
    File        : IAdaptable
    Purpose     : General interface for allowing classes to provide adapters
                  via the Adapter design pattern https://en.wikipedia.org/wiki/Adapter_pattern
    Author(s)   : pjudge
    Created     : 2016-10-12
    Notes       :
  ------------------------------------------------------------------*/
interface OpenEdge.Core.IAdaptable:

  /* Returns an adapter for this message

     @param P.L.Class The type we want to adapt to
     @return P.L.Object The adapter. SHOULD be of the type specified by the input argument */
  method public Progress.Lang.Object GetAdapter(input poAdaptTo as class Progress.Lang.Class).

end interface.
```

Make the MapperDecorator Adaptable

```
class data.shared.MapperDecorator abstract implements IMapper, IAdaptable:
    // the IMapper being decorated
    define protected property DecoratedMapper as IMapper no-undo get. private set.

    /* Can this decorator op decorated object adapt in the way required? */
    method public Progress.Lang.Object GetAdapter(input pAdaptTo as class Progress.Lang.Class):
        if this-object:GetClass():IsA(pAdaptTo) then
            return this-object.
        if valid-object(DecoratedMapper) and type-of(DecoratedMapper, IAdaptable) then
            return cast(DecoratedMapper, IAdaptable):GetAdapter(pAdaptTo).
        return ?.
    end method.

end class.
```

Building a decorated Mapper

```
function BuildMapper returns IMapper():
 return new CollectionTransactionScope(
             new LoggingMapper(
                 new AuthorisedBufferOperation(
                     new BufferMapper(buffer Department:handle, get-class(DepartmentRecord)).
end function.

define variable mapper as IMapper no-undo.
define variable deptRecord as IRecord no-undo.
define variable supportsAuth as ISupportAuthorization no-undo.

mapper = BuildMapper().
// Set the AuthManager in the mapper
supportsAuth = mapper:GetAdapter(get-class(ISupportAuthorization)).
if valid-object(supportsAuth) then
    assign supportsAuth:AuthManager = new common.shared.AuthorizationManager().

// Delete this department
deptRecord  = mapper:Get('DeptCode eq "100"').
mapper:Delete(deptRecord).
```

Finding an adapter

```
15. supportsAuth = mapper:GetAdapter(get-class(ISupportAuthorization)).
```

```
1. class data.shared.CollectionTransactionScope inherits MapperDecorator:
2.   // IsA(ISupportAuthorization) = FALSE
3.   method public Progress.Lang.Object GetAdapter(input pAdaptTo as class Progress.Lang.Class):
```

```
1. class data.shared.LoggingMapper inherits MapperDecorator  implements ISupportLogging:
2.   // IsA(ISupportAuthorization) = FALSE
3.   method public Progress.Lang.Object GetAdapter(input pAdaptTo as class Progress.Lang.Class):
```

```
1. class data.shared.AuthorisedBufferOperation inherits MapperDecorator
2.                                             implements ISupportAuthorization:
3.   // IsA(ISupportAuthorization ) = TRUE
4.   method public Progress.Lang.Object GetAdapter(input pAdaptTo as class Progress.Lang.Class):
```

```
1. class data.shared.BufferMapper:
2.   // IsA(ISupportAuthorization) = FALSE
```

In conclusion

- OO(ABL) gives you mechanisms to get help from the compiler
  - Can add flexibility in constructing optional dependencies
  - Build software infrastructure / framework / skeleton first and add implementations later
- Dynamic data constructs (handles) let you build once for many uses
  - OOABL wrappers keep errors as compile-time
- Design patterns give you a common set of knowledge for building applications
  - Give you a set of blueprints to combine with ABL

Sample code from this session available at https://github.com/PeterJudge-PSC/ooabl_patterns

Progress®