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.
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 elsewhere
  - 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

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

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

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

11. 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()
   dept:DeptCode = Department.DeptCode
   dept:DeptName = Department.DeptName.
4. // calculate average tenure
5. for each Employee where Employee.DeptCode eq Department.DeptCode no-lock:
6.   assign numEmps = numEmps + 1
7.     totAge = totAge + interval(today, Employee.Startdate, 'months').
8. end.
9. assign dept:AvgEmpTenure = (totAge / numEmps).
10. // application/business logic does Stuff
11. message dept:DeptCode // 100
12.     dept:AvgEmpTenure // 236.29
13. assign dept:DeptName = 'Department of One Hundred'.
14. // write any changes to the DB
15. find first Department where Department.DeptCode eq dept:DeptCode exclusive-lock.
16. assign Department.DeptCode = dept:DeptCode
17.     Department.DeptName = dept:DeptName
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. // application/business logic does Stuff
13. message dept:DeptCode // 100
14.   dept:AvgEmpTenure // 236.29
15. assign dept:DeptName = 'Department of One Hundred'.
16. // write any changes to the DB
17. find first Department where Department.DeptCode eq dept:DeptCode exclusive-lock.
18. assign Department.DeptCode = dept:DeptCode
19.   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 dept.
16. //Create a new Department record. Other CRUDs are left out for space reasons
17. method public void Create(input pDept as class DepartmentRecord):
18.    create Department.
20.       Department.DeptName = pDept:DeptName.
21. end method.

The Mapper creates and populates the ActiveRecord
Assign calculate field values
Consuming the mapper

1. def var dept as class data.s2k.DepartmentRecord.
2. def var mapper as class data.s2k.DepartmentMapper.

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

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

// write any changes to the DB
16. 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
  Progress.Reflect.*

  OE11.0.0+
  OE11.4.0+
Sample 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.
9. /* Creates new records in the persistent store/database */
10. method public void Create(input pData as class IRecord).
11.
12. /* Deletes records from the persistent store/database */
13. method public void Delete(input pData as class IRecord).
14.
15. /* Updates records in the persistent store/database */
16. method public void Update(input pData as class IRecord).
17.
18. end interface.
BufferMapper :: IMapper

1. class data.shared.BufferMapper implements IMapper:
2. define public property RecordType as class Progress.Lang.Class no-undo get. protected set.
3. // the underlying buffer
4. define protected property DataBuffer as handle no-undo get. private set. This is the data source (DB or TT)
5. /* Constructor */
6. constructor public BufferMapper(input pBuffer as handle,
7.    input pRecordType as class Progress.Lang.Class):
8.    this-object:DataBuffer = pBuffer.
9.    this-object:RecordType = pRecordType.
10. end constructor.
11. // Remove a record
12. method public void Delete(input pData as class IRecord):
13.      do transaction:
14.         // simplified FIND. Doesn't deal with multiple key fields and non-char values
15.         DataBuffer:find-first(substitute('where &1 = "&2"',
16.            DataBuffer:keys, dynamic-property(pData, DataBuffer:keys)),
17.            exclusive-lock).
18.         DataBuffer:buffer-delete().
19.      finally:
20.         DataBuffer:buffer-release().
21.      end finally.
22.      end. // trans
23. end method.
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. Maps a field to and from a property
36.         dynamic-property(data, props[loop]:Name) =
37.         
38.         
39.         end.
40.     return data.
41. end method.
42. end class.
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):
21. end method.
22. end class.

We can change where the data is stored at runtime
Consuming records

1. def var dept as class data.s2k.DepartmentRecord.
2. def var deptDAO as class data.s2k.DepartmentDAO.
3. 
4. assign deptDAO = new data.s2k.DepartmentDAO()
5.   dept = deptDAO:Get('where DeptCode eq "100"')
6. 
7. // application/business logic does Stuff
8. message dept:DeptCode // 100
9.   dept:AvgEmpTenure // 236.29
10. .
11. assign dept:DeptName = 'Department of One Hundred'.
12. 
13. deptDAO:Update(dept).
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 character).
    method public class DepartmentRecordCollection GetAll(input pWhere as character).
    method public class DepartmentRecordMap GetAll(input pWhere as character):

    //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.

Simple, no extra stuff
May be ordered
No consistent iteration
No quick find

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

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

## Interfaces

- OpenEdge.Core.Collections.ICollection
- OpenEdge.Core.Collections.IEnumerable
- OpenEdge.Core.Collections.IList
- OpenEdge.Core.Collections.IListIterator
- OpenEdge.Core.Collections.IMap
- OpenEdge.Core.Collections.IMapEntry
- OpenEdge.Core.Collections.ISet
- OpenEdge.Core.Collections.IStringCollection
- OpenEdge.Core.Collections.IStringKeyedMap
- OpenEdge.Core.Collections.IStringStringMap

## Classes

- AbstractTTCollection

---

- **Group of objects**: A collection represents a group of objects, known as its elements.
- **General cursor**: Traverses a collection forward.
- **Keys / values grp**: An iterator for lists that can traverse the list in both directions.

---

- A map entry (key-value pair). The IMap.EntrySet returns a set-view of the map, whose elements are of this class.
- A collection that contains no duplicate elements.
- Interface defining a typed String Collection
- A typed String/String Map
- A typed String/String Map
Populating a collection in mapper

```java
method public class DepartmentRecordCollection GetAll(input pWhere as character):
  query qryDept:query=prepare('preselect each Department where ' + pWhere + ' no-lock').
  query qryDept:query-open().
  data = new DepartmentRecordCollection().
  query qryDept:get-first().
  do while not query qryDept:query-off-end:
    dept = new data.s2k.DepartmentRecord().
    dept:DeptName = Department.DeptName.
    data:Add(dept).
  // Calculate field value
  for each Employee where Employee.DeptCode eq Department.DeptCode no-lock:
    assign numEmps = numEmps + 1
    totAge = totAge + interval(today, Employee.Startdate, 'months').
  assign dept:AvgEmpTenure = (totAge / numEmps).
  query qryDept:get-next().
  return data.
end method.
```

1. Create a new collection object
2. Simply add the ActiveRecord objects; no need to count here
3. Return to the caller
Collections enable ActiveRecord children

1. class data.s2k.DepartmentRecord:
2.   // DATABASE FIELD Department.DeptCode
3.   define public property DeptCode as character no-undo
4.       get. set.
5.   
6.   // DATABASE FIELD Department.DeptName
7.   define public property DeptName as character no-undo
8.       get. set.
9.   
10.  // CALCULATED FIELD: The average employee tenure/months
11.   define public property AvgEmpTenure as decimal no-undo
12.       get. set.
13.   
14.   // CHILD RECORDS
15.   define public property Employees as EmployeeRecordCollection no-undo
16.       get. set.
17.   
18. end class.
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. Retrieve many records

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. Update many records

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.
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.
Consuming records … still looks the same

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

3.

4.

5. assign deptDAO = new data.s2k.DepartmentDAO()
6.       dept = deptDAO:Get('where DeptCode eq "100"')
7.

8. // application/business logic does Stuff
9. message dept:DeptCode // 100
10.    dept:AvgEmpTenure  // 236.29
11.    dept:Employees:Size // 7
12. assign dept:DeptName = 'Department of One Hundred'.
13.
14. // write any changes to the DB
15. mapper:Update(dept).
```
Making record properties read-only

1. class data.s2k.DepartmentRecord:
2.   // DATABASE FIELD Department.DeptCode
3.   define public property DeptCode as character no-undo
4.     get. set.
5. 
6.   // DATABASE FIELD Department.DeptName
7.   define public property DeptName as character no-undo
8.     get. set.
9. 
10.  // CALCULATED FIELD: The average employee tenure/months
11.   define public property AvgEmpTenure as decimal no-undo
12.     get.
13. 
14.  // CHILD RECORDS
15.   define public property Employees as EmployeeRecordCollection no-undo
16.     get. set.
17. 
18. end class.
Read-only property, from a collection

1. class data.s2k.DepartmentRecord:
2.   // DATABASE FIELD Department.DeptCode
3.   define public property DeptCode as character no-undo
4.       get. set.
5.   
6.   // DATABASE FIELD Department.DeptName
7.   define public property DeptName as character no-undo
8.       get. set.
9.   
10.  // CHILD RECORDS
11.  define public property Employees as EmployeeRecordCollection no-undo
12.       get. set.
13.   
14.  // CALCULATED FIELD: The average employee tenure/months
15.  define public property AvgEmpTenure as decimal no-undo
16.      get():
17.         // sadly we can't define the property as DECIMALS 2
18.         return CalculateTenure('months')..
19.      end get.
20.  end class.
Read-only property, from a collection

1. method protected decimal CalculateTenure(input pUnit as character):
2.   define variable totTenure as integer no-undo.
3.   // define the variable we return as DECIMALS 2
4.   define variable avgTenure as decimal decimals 2 no-undo.
5.   define variable iterator as IIterator no-undo.
6.   
7.   if this-object:Employees:Size eq 0 then
8.     avgTenure = 0.00.
9.   else
10.   do:
11.     iterator = this-object:Employees:Iterator().
12.     do while iterator:HasNext():
13.       totTenure = totTenure
14.         + interval(today, cast(iterator:Next(), EmployeeRecord):StartDate, pUnit).
15.     end.
16.   avgTenure = (totTenure / this-object:Employees:Size).
17. end
18. return avgTenure.
19. 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](https://stackoverflow.com/a/3489187/18177)
Adding more to a mapper

Record Transaction Scope
Collection Transaction Scope
Authorized Buffer Operation
Logging Mapper

- Defines transaction scope for an update operation
- Adds authorization for a data operation
- 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

```plaintext
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 CollectionTransaction
  4. inherits Mapper, CollectionTransaction
```
Adding functionality using inheritance

```java
class Department<
  1. implements <none>
  2. implements ISupportAuthorization
  3. implements ISupportLogging
  4. implements ISupportAuthorization, ISupportLogging

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

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

2. **OR** Implement interface in individual Mapper
   
   Only this Mapper gets this behaviour

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

3. **OR** Implement interface in a Decorator
   
   Only certain Mappers get this behaviour
Our MapperDecorator class

```csharp
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

23. class data.shared.AuthorisedBufferOperation

24.    inherits MapperDecorator       // behaviour from IMapper

25.    implements ISupportAuthorization:   // Adds Authorization

26.

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

28.

29. constructor public AuthorisedBufferOperation (input pMapper as IMapper):

30.    super (input pMapper).

31. end constructor.

32.

33. method override public void Delete( input pData as IRecord ):

34.    if AuthManager:AuthorizeOperation('delete+' + pData:GetClass():TypeName) then

35.        super:Delete(input pData).

36.    end method.

37.

38. end class.
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. // add auth decorator for deletes
7. mapper = new AuthorisedBufferOperation(mapper).
8. 
9. // Delete this department
10. deptRecord = mapper:Get('DeptCode eq "100"').
11. mapper:Delete(deptRecord).
Building a decorated Mapper

1. define variable mapper as IMapper no-undo.
2. define variable deptRecord as IRecord no-undo.
3. // base IMapper
4. mapper = new BufferMapper(buffer "Department");
5. // add auth decorator
6. mapper = new AuthorisedBufferOperation(mapper);
7. // Delete this department
8. deptRecord = mapper:Get('DeptCode eq "100"').
9. 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](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?

1. function BuildMapper returns IMapper():
2.    return new CollectionTransactionScope(
3.        new LoggingMapper(
4.            new AuthorisedBufferOperation(
5.                new BufferMapper(buffer Department:handle, get-class(DepartmentRecord)).
6.            end function.
7.        end function.
8.    end function.
9. define variable mapper as IMapper no-undo.
10. define variable deptRecord as IRecord no-undo.
11. mapper = BuildMapper().
12. // Set the AuthManager in the mapper
13. cast(mapper, ISupportAuthorization):AuthManager = new common.shared.AuthorizationManager().
14. // Delete this department
15. deptRecord = mapper:Get('DeptCode eq "100"').
16. mapper:Delete(deptRecord).
Adapters

interface IAdaptable:


@returns P.L.Object The adapter. SHOULD be of the type specified by the input argument *@
@returns P.L.Object The adapter. SHOULD be of the type specified by the input argument *@
@returns P.L.Object The adapter. SHOULD be of the type specified by the input argument */
Make the MapperDecorator Adaptable

```plaintext
1. class data.shared.MapperDecorator abstract implements IMapper, IAdaptable:
2.     // the IMapper being decorated
3.     define protected property DecoratedMapper as IMapper no-undo get. private set.
4. 
5. /* Can this decorator op decorated object adapt in the way required? */
7.     if this-object:GetClass():IsA(pAdaptTo) then
8.         return this-object.
9.     if valid-object(DecoratedMapper) and type-of(DecoratedMapper, IAdaptable) then
10.        return cast(DecoratedMapper, IAdaptable):GetAdapter(pAdaptTo).
11.     return ?.
12. end method.
13. 
14. end class.
```
Building a decorated Mapper

1. function BuildMapper returns IMapper():
2.   return new CollectionTransactionScope(
3.     new LoggingMapper(
4.       new AuthorisedBufferOperation(
5.         new BufferMapper(buffer Department: handle, get-class(DepartmentRecord)).
6.   end function.
7.
8. define variable mapper as IMapper no-undo.
9. define variable deptRecord as IRecord no-undo.
10. define variable supportsAuth as ISupportAuthorization no-undo.
11.
12. mapper = BuildMapper().
13. // Set the AuthManager in the mapper
14. supportsAuth = mapper:GetAdapter(get-class(ISupportAuthorization)).
15. if valid-object(supportsAuth) then
16.     assign supportsAuth:AuthManager = new common.shared.AuthorizationManager().
17.
18. // Delete this department
19. deptRecord = mapper:Get('DeptCode eq "100"').
20. mapper:Delete(deptRecord).
Finding an adapter

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

1. class data.shared.CollectionTransactionScope inherits MapperDecorator:
2.   // IsA(ISupportAuthorization) = FALSE

1. class data.shared.LoggingMapper inherits MapperDecorator implements ISupportLogging:
2.   // IsA(ISupportAuthorization) = FALSE

1. class data.shared.AuthorisedBufferOperation inherits MapperDecorator implements ISupportAuthorization:
2.   // IsA(ISupportAuthorization ) = TRUE

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](https://github.com/PeterJudge-PSC/ooabl_patterns)