Progress®

# Composing Complex Applications

From NEW() to Factories and Beyond

**Peter Judge**
pjudge@progress.com

# Software goals

- Loosen dependencies between objects
  - Easier to change/replace/extend behaviour
  - Easier to test (swap out real objects for doppelgänger)
- 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

# Why use objects?

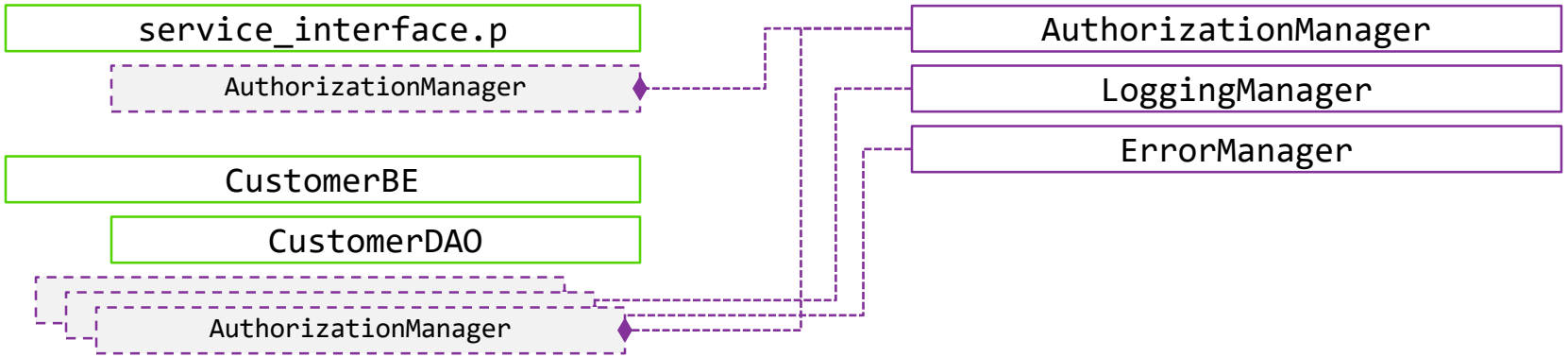
- Can define a compiler-enforceable API
  - No surprise! –type calls
  - No one in your private members
- Strong typing: compiler reduces errors by requiring stuff to be there

- We want something that's immediately useable
  - Externally visible `IsInitDone` does not pass smell test

Interface | Abstract
Manager | Service

# The beginning, a very good place to start

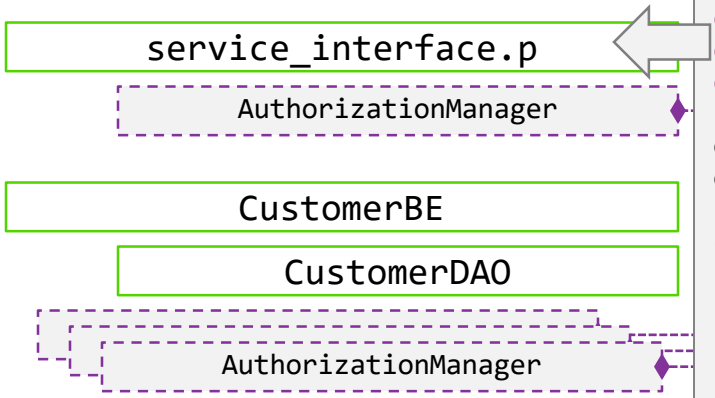
Services.* Managers.*



Interface | Abstract | Manager | Service

# The beginning, a very good place to start

Services.*

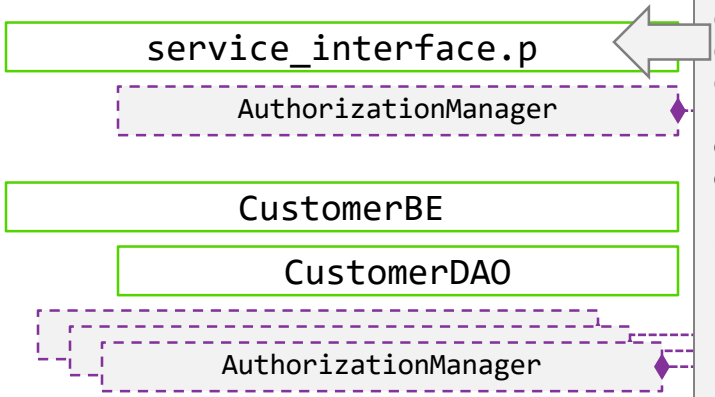



```
def input  param pcServiceName as char.
def input  param pcOperation as char.

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

# The beginning, a ~~very good~~ place to start

Services.*

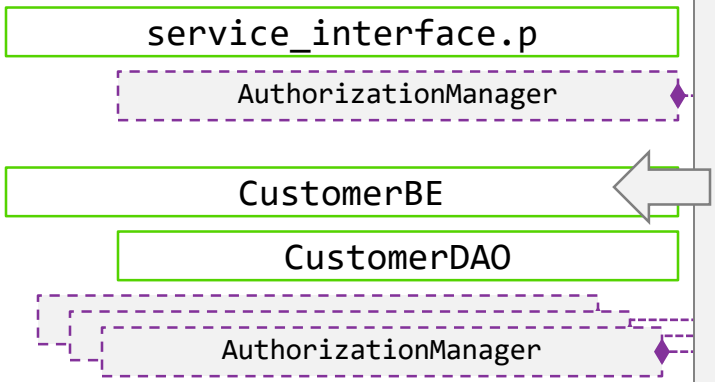



```
def input  param pcServiceName as char.
def input  param pcOperation as char.

def in-out param dataset-handle phServi    ta.
def in-out param dataset-handle phServ     ms.
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

# The beginning, a very good place to start

Services.*

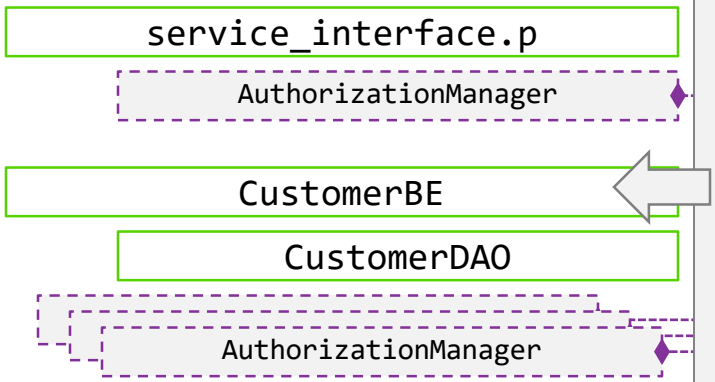



```
class Services.CustomerBE:
  def public prop DataAccess as CustomerDAO
      get. set.
  def public prop LogMgr as LoggingManager
      get. set.
  def public prop ErrorMgr as ErrorManager
      get. set.
  def public prop AuthMgr as AuthorizationManager
      get. set.

  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

# The beginning, a ~~very good~~ place to start

Services.*




```
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.
```

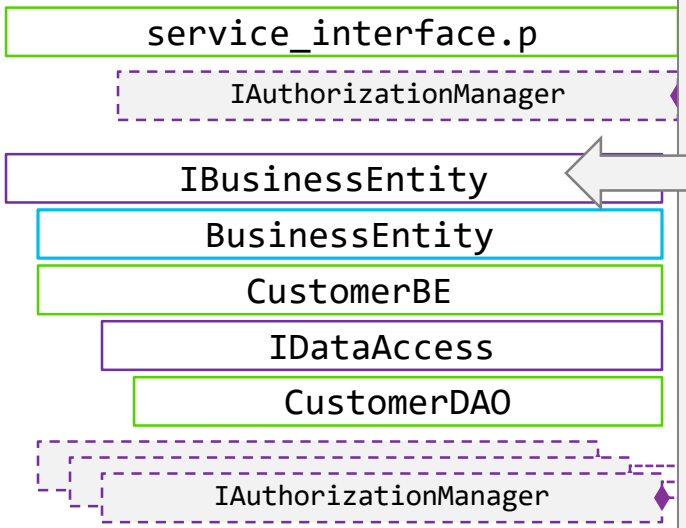
# Use 'contract' types in definitions

- 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-won't-break* contract with implementers
- Use inheritance for common or shared behaviour
  - Careful of deep hierarchies – reduces flexibility

Interface | Abstract | Manager | Service

# Use interfaces

Services.*

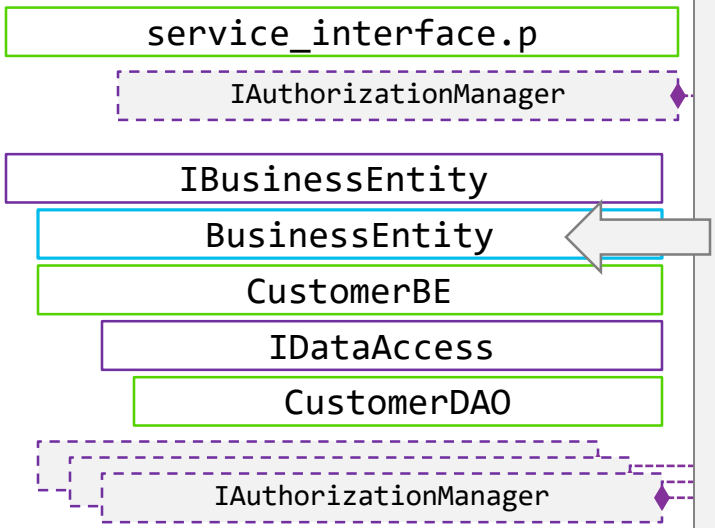



```
interface Services.IBusinessEntity:
  def public prop DataAccess as IDataAccess
      get. set.

  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

# Create abstract super-class

Services.*

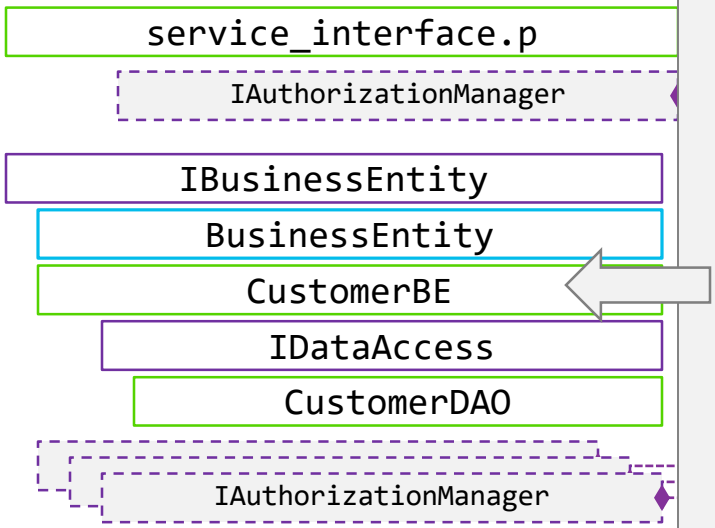



```
class Services.BusinessEntity
      abstract implements IBusinessEntity:
  def public prop DataAccess as IDataAccess get. set.
  def public prop LogMgr as ILoggingManager get. set.
  def public prop ErrorMgr as IErrorManager get. set.
  def public prop AuthMgr as IAuthorizationManager
      get.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

# Refactor to use super-class

Services.*

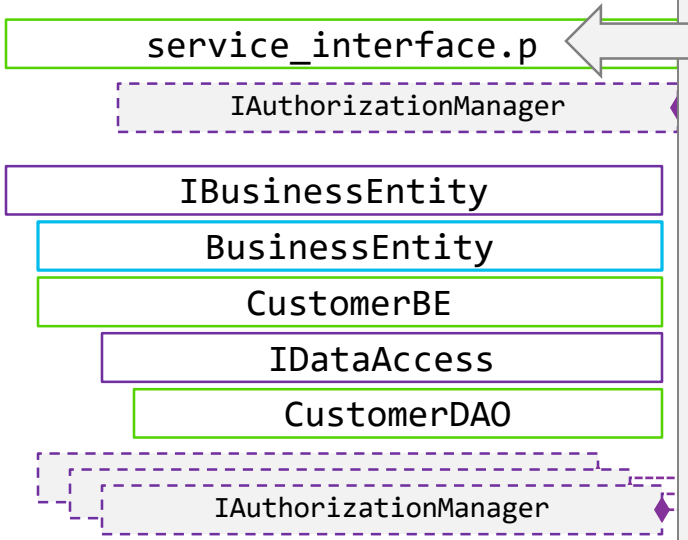



```
class Services.CustomerBE
     inherits BusinessEntity:

  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

# Refactor to call interfaces

Services.*

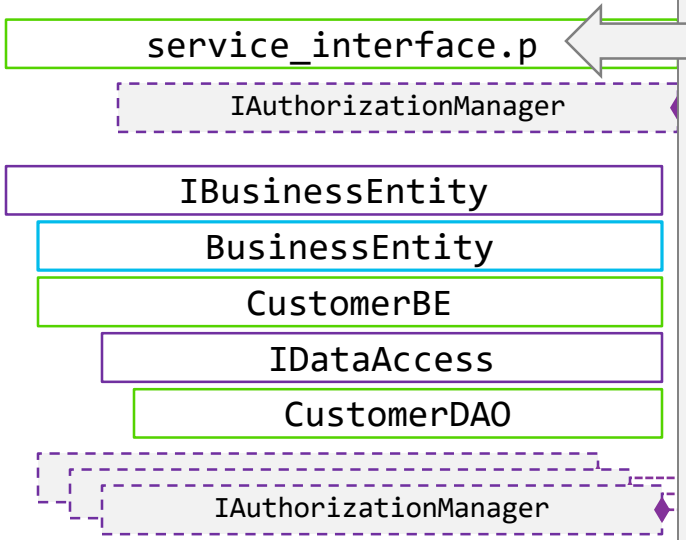



```
def input  param pcServiceName as char.
def input  param pcOperation as char.
def in-out param dataset-handle phServiceData.
def in-out param dataset-handle phServiceParams.
def var oBE as IBusinessEntity.

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

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

# Refactor to DYNAMIC-NEW

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).

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?*

# Factories & builders

- Abstract factory Provide an interface for creating *families* of related or dependent objects without specifying their concrete classes
- Builder Separate the construction of a complex object from its representation, allowing the same construction process to create various representations
- Factory method Define an interface for creating a *single* object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses

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

# Builder pattern

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

                get.

  method static public BusinessEntityBuilder Build(input pcService as character):
    define variable oBuilder as BusinessEntityBuilder no-undo.

    case pcService:
      /*default */
      otherwise oBuilder = new DefaultBEBuilder(pcService).
    end case.

    return oBuilder.
  end method.
end class.
```

Abstract factory

Factory method

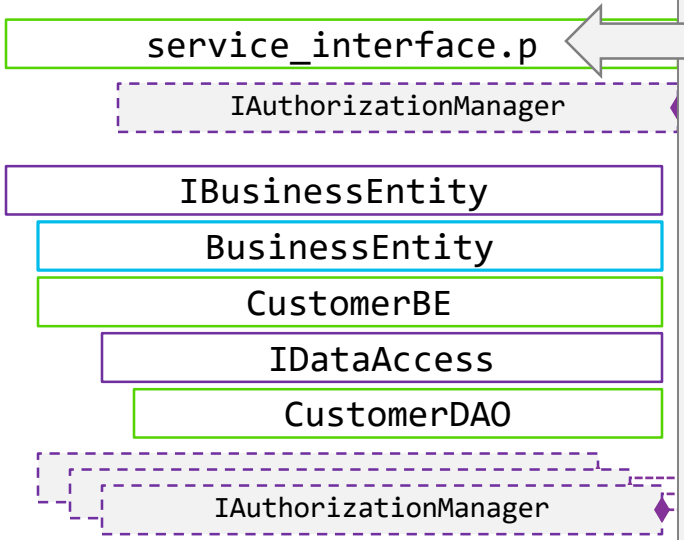
# Builder implementation

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

  /* This 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.' + mcServiceName  + '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.*

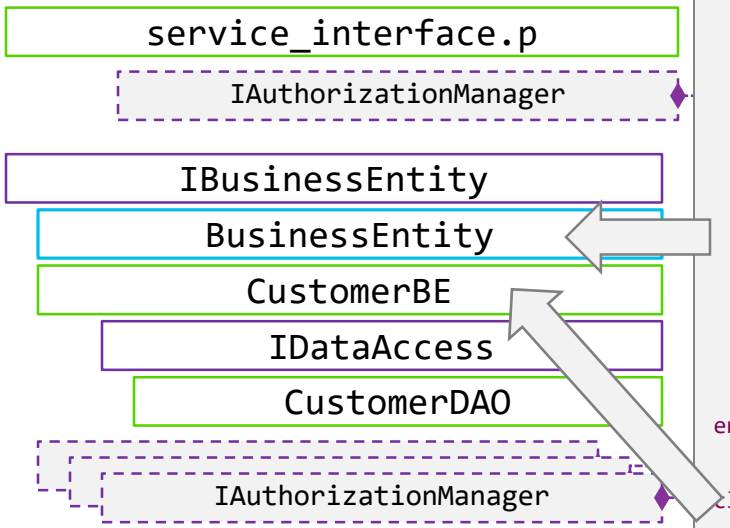



```
def input  param pcServiceName as char.
def input  param pcOperation as char.
def in-out param dataset-handle phServiceData.
def in-out param dataset-handle phServiceParams.
def var oBE as IBusinessEntity.

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

oBE = BusinessEntityBuilder
            :Build(pcServiceName)
            :Entity.

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

Interface | Abstract | Manager | Service

# Loosening child dependencies

Services.*

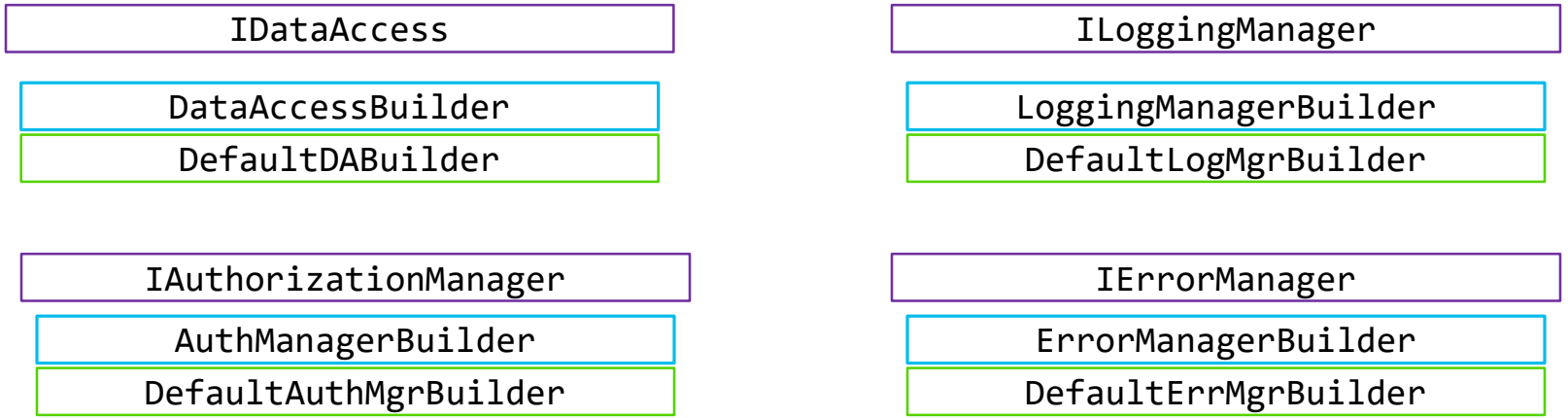


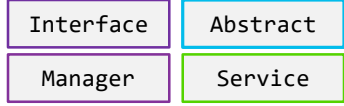

```
class Services.BusinessEntity
      abstract implements IBusinessEntity:
  def public prop DataAccess as IDataAccess get. set.
  def public prop LogMgr as ILoggingManager get. set.
  def public prop ErrorMgr as IErrorManager get. set.
  def public prop AuthMgr as IAuthorizationManager
      get.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

# Step 1: Extract API & builders

- Create new interfaces and/or parent classes
- Create new Builders






# Extract API & builders

Services.*

Managers.*

service_interface.p

IAuthorizationManager

IBusinessEntity

BusinessEntity

CustomerBE

IDataAccess

CustomerDAO

IAuthorizationManager

BusinessEntityBuilder

DefaultBEBuilder

IAuthorizationManager

LocalAuthorizationManager

ILoggingManager

FileLoggingManager

IErrorManager

DefaultErrorManager

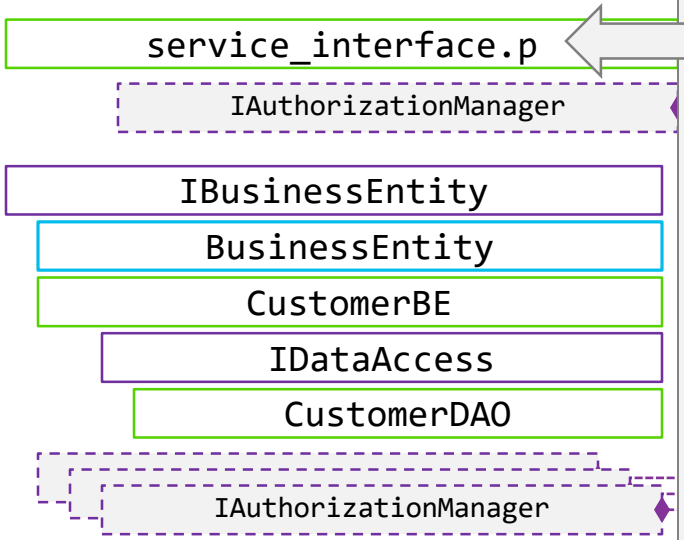
ErrorManagerBuilder

DefaultErrMgrBuilder

Interface | Abstract | Manager | Service

# More builders

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.
def var oAuthMgr as IAuthorizationManager.

oAuthMgr = AuthManagerBuilder
                :Build()
                :Manager.

oAuthMgr:AuthorizeServiceOperation(pcServiceName, pcOperation).

oBE = BusinessEntityBuilder
            :Build(pcServiceName)
            :Entity.

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

# 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.

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.

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

# Adding support for Data Access

- Since BE and DA closely linked, BE can create Data Access object
  - Using a builder, of course

```
class Services.CustomerBE inherits BusinessEntity:

  constructor public CustomerBE():
    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

# Adding support for Data Access

- Alternatively, have someone else create it and pass it in
  - ~~BE needs to some something about DA that isn't core to BE operation~~
  - ~~To use different DA, BE needs changes~~
    - This is called Dependency Injection
- Our BE now truly only has business (domain) logic in it

```
class Services.CustomerBE inherits BusinessEntity:
  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.
```

# Dependency Injection: Data Access into Business Entity

```
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.
```

# Dependency Injection: Data Access into Business Entity

```
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.
```

# Dependency Injection: Data Access into Business Entity

```
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.
```

# Adding support for the non-core dependencies

| Component | Required? | Core responsibility? |
|---|---|---|
| Error Manager | Yes | No |
| Authorization Manager | No | No |
| Logging Manager | No | No |

```
interface Managers.ISupportAuthorization:
  def public property AuthMgr as IAuthorizationManager get. set.

interface Managers.ISupportLogging:
  def public property LogMgr as ILoggingManager get. set.

interface Managers.ISupportErrorHandling:
  def public property ErrMgr as IErrorManager get. set.
```

# Adding support for optional dependencies

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

***n*** managers = ***n!*** combinations

1. EITHER Implement interface in BusinessEntity superclass
   - All BE's get this behaviour

```
class Services.BusinessEntity abstract implements IBusinessEntity, ISupportErrorHandling:
```

2. OR Implement interface in individual BusinessEntity
   - Only this BE gets this behaviour

```
class Services.CustomerBE inherits BusinessEntity implements ISupportAuthorization:
```

3. OR Implement interface in a Decorator
   - Only certain BE's get this behaviour

# Optional dependencies: decorator / façade

```
class Services.BusinessEntityDecorator abstract implements IBusinessEntity:
  define public property DecoratedBE as IBusinessEntity no-undo get. private set.

  /* properties, methods from interface */
  define public property DataAccess as IDataAccess no-undo
    get():
      return DecoratedBE:DataAccess.
    end get.

  constructor public BusinessEntityDecorator(input poBE as IBusinessEntity):
    assign DecoratedBE = poBE.
  end.

  method public void Fetch(<params>):
    DecoratedBE:Fetch(<args>).
  end method.

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

# Optional dependencies: decorator class

```
class Services.LoggingBE inherits BusinessEntityDecorator /* implements IBusinessEntity */
      implements ISupportLogging:                         /* and ISupportLogging        */

  define public property LogMgr as ILoggingManager no-undo get. set.

  constructor public LoggingBE (input poBE as IBusinessEntity):
    super (input poBE).
  end constructor.

  method override public void Fetch( <params> ):
    define variable iNumRecords as integer no-undo.

    super:Fetch(<args>).

    LogMgr:LogMessage(substitute('Records fetched: &1', iNumRecords)).

    catch oError as Progress.Lang.Error :
      LogMgr:LogError(oError).
    end catch.
  end method.
end class.
```

# Optional dependencies: builder

```
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.

      /* constructor injection */
      oBE = dynamic-new 'Services.' + mcServiceName  + 'BE' (oDAO).

      if GetConfigOption( 'SupportLog') and not type-of(oBE, ISupportLogging) then
        oBE = new LoggingBE(oBE).

      /* property injection */
      if type-of(oBE, ISupportLogging) then
        assign cast(oBE, ISupportLogging):LogMgr = LogManagerBuilder:Build():Manager.

      return oBE.
    end get.
```

# Optional dependencies: decorator class

```
class Services.AuthorizedBE inherits BusinessEntityDecorator
        implements ISupportAuthorization:

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

  constructor public AuthorizedBE (input poBE as IBusinessEntity):
    super (input poBE).
  end constructor.

  method override public void Fetch (<params>):
    AuthMgr:AuthorizeServiceOperation(<args>).

    super:Fetch(<args>).
  end method.
end class.
```

# Optional dependencies: builder – multiple decorators

```
class Services.DefaultBEBuilder inherits BusinessEntityBuilder:
  define private variable mcServiceName as character no-undo.
  define override public property Entity as IBusinessEntity no-undo
    get():
      oDAO = DataAccessBuilder:Build(mcServiceName):DataAccess.
      oBE = dynamic-new 'Services.' + mcServiceName  + 'BE' (oDAO).

      if GetConfigOption('SupportLog') and not type-of(oBE, ISupportLogging) then
        oBE = new LoggingBE(oBE).

      if GetConfigOption('SupportAuth') and not type-of(oBE, ISupportAuthorization) then
        oBE = new AuthorizedBE(oBE).

      if type-of(oBE, ISupportLogManager) then
        cast(oBE, ISupportLogManager):LogMgr = LogManagerBuilder:Build():Manager.

      if type-of(oBE, ISupportAuthorization) then
        cast(oBE, ISupportAuthorization):AuthMgr = AuthManagerBuilder:Build():Manager.

      return oBE.
  end get.
```

```
def var oBE as IBusinessEntity.
oBE = BusinessEntityBuilder:Build(pcServiceName):Entity.
oBE:Fetch(<args>).
```

```
class LoggingBE implements ISupportLogging:
  def public property DecoratedBE as IBusinessEntity get. private set.
  def public property LogMgr as ILoggingManager get. set.
  method public void Fetch(<params>).
    DecoratedBE:Fetch(<args>).
    LogMgr:LogMessage(substitute('Records fetched: &1', iNumRecords)).
```

```
class AuthorizedBE implements ISupportAuthorization:
  define public property DecoratedBE as IBusinessEntity get. private set.
  define public property AuthMgr as IAuthorizationManager get. set.
  method public void Fetch(<params>).
    AuthMgr:AuthorizeOperation(<args>).
    DecoratedBE:Fetch(<args>).
```

```
class BusinessEntity :
  method public void Fetch(<params>):
    DataAccess:Fetch(<args>).
```

```
def var oBE as IBusinessEntity.
oBE = BusinessEntityBuilder:Build(pcServiceName):Entity.
oBE:Fetch(<args>).
```

```
class LoggingBE implements ISupportLogging:
  def public property DecoratedBE as IBusinessEntity get. private set.
  def public property LogMgr as ILoggingManager get. set.
  method public void Fetch(<params>).
    DecoratedBE:Fetch(<args>).
    LogMgr:LogMessage(substitute('Records fetched: &1', iNumRecords)).
```

```
class AuthorizedBE implements ISupportAuthorization:
  define public property DecoratedBE as IBusinessEntity get. private set.
  define public property AuthMgr as IAuthorizationManager get. set.
  method public void Fetch(<params>).
    AuthMgr:AuthorizeOperation(<args>).
    DecoratedBE:Fetch(<args>).
```

```
class BusinessEntity :
  method public void Fetch(<params>):
    DataAccess:Fetch(<args>).
```

```
def var oBE as IBusinessEntity.
oBE = BusinessEntityBuilder:Build(pcServiceName):Entity.
oBE:Fetch(<args>).
```

```
class LoggingBE implements ISupportLogging:
  def public property DecoratedBE as IBusinessEntity get. private set.
  def public property LogMgr as ILoggingManager get. set.
  method public void Fetch(<params>).
    DecoratedBE:Fetch(<args>).
    LogMgr:LogMessage(substitute('Records fetched: &1', iNumRecords)).
```

```
class AuthorizedBE implements ISupportAuthorization:
  define public property DecoratedBE as IBusinessEntity get. private set.
  define public property AuthMgr as IAuthorizationManager get. set.
  method public void Fetch(<params>).
    AuthMgr:AuthorizeOperation(<args>).
    DecoratedBE:Fetch(<args>).
```

```
class BusinessEntity :
  method public void Fetch(<params>):
    DataAccess:Fetch(<args>).
```

# Inheritance vs. Decoration

```
class Customer<auth|log|err|auth-err-log|…> inherits BusinessEntity
1. implements ISupportAuthorization
2. Implements                        ISupportLogging
3. implements                                          ISupportErrorHandling
4. implements ISupportAuthorization, ISupportLogging, ISupportErrorHandling
5. implements ISupportAuthorization, ISupportLogging,
6. implements ISupportAuthorization,                   ISupportErrorHandling
7. implements                        ISupportLogging, ISupportErrorHandling

class BusinessEntity implements ISupportAuthorization, ISupportLogging:
  def public property DataAccess as IDataAccess get.
  method public void Fetch(<params>).
  method public void Save (<params>).

  def public property LogMgr as ILoggingManager get. set.
  def public property AuthMgr as IAuthorizationManager get. set.
```

# Builder pattern

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

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

```
  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).
```

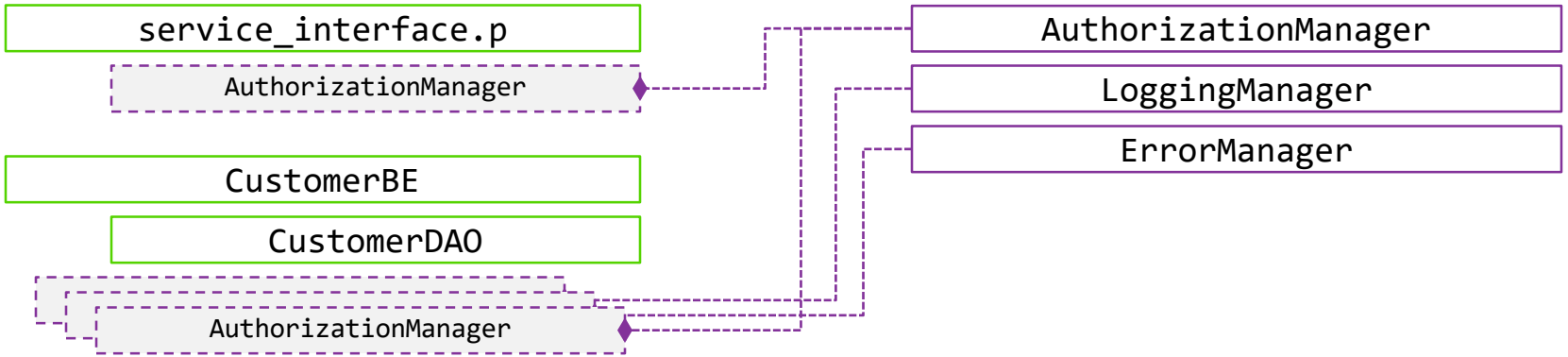
*Builder pattern*

Interface | Abstract
Manager | Service

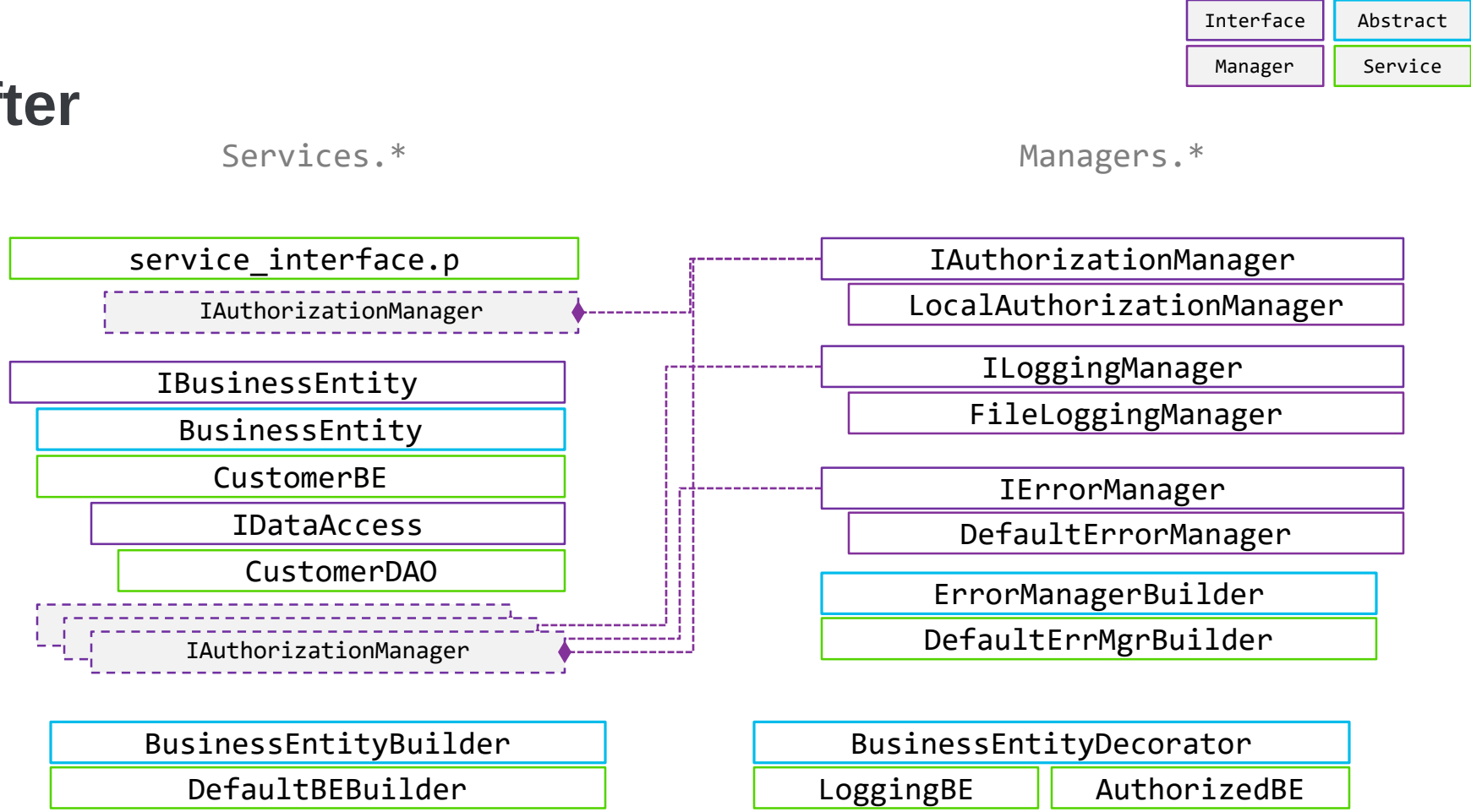
# Before

Services.*

Managers.*



# After



# Done for now: what have we got?

- We created a number of  API or contract types
  - Interfaces and abstract super-classes
  - Kept functionality small in scope
- Created a number of builders to construct objects that implement these APIs
  - Gave ourselves room for extensions
  - Far less code impact for changes
- Builders allow us to keep infrastructure out of our (domain/business) objects
- Considered how to add required and optional dependencies into objects
  - Via direct properties in main interfaces
  - Via decorators
  - Via simple interface implementation

# Advanced topics

- Building the object graph from JSON / XML / database tables
- Managing object lifecycles
  - Static-member-free Singletons
- Service Managers
- Inversion of Control Containers

- Further reading
  - Martin Fowler,  "Uncle" Bob Martin, Mark Seeman and many others
  - Stackoverflow
  - Wikipedia https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)
    https://en.wikipedia.org/wiki/Dependency_inversion_principle
    https://en.wikipedia.org/wiki/Service_locator_pattern
  - Portland pattern repository http://c2.com/cgi/wiki?WelcomeVisitors

# Questions?

pjudge@progress.com


Progress®