Consultingwerk
software architecture and development

Building great Interfaces with OOABL

Mike Fechner
Director


Consultingwerk
software architecture and development

Consultingwerk Software Services Ltd.

- Independent IT consulting organization
- Focusing on **OpenEdge** and **related technology**
- Located in Cologne, Germany, subsidiaries in UK and Romania
- Customers in Europe, North America, Australia and South Africa
- Vendor of developer tools and consulting services
- Specialized in GUI for .NET, Angular, OO, Software Architecture, Application Integration
- Experts in OpenEdge Application Modernization

Mike Fechner

- Director, Lead Modernization Architect and Product Manager of the SmartComponent Library and WinKit
- Specialized on object oriented design, software architecture, desktop user interfaces and web technologies
- 28 years of Progress experience (V5 … OE11)
- Active member of the OpenEdge community
- Frequent speaker at OpenEdge related conferences around the world

Agenda

- **Introduction**
- Interface vs. Implementation
- Enums
- Value or Parameter Objects
- Fluent Interfaces
- Builders
- Strong Typed Dynamic Query Interfaces
- Factories
- Facades and Decorators

Introduction

- Object oriented (ABL) programming is more than just a bunch of new syntax elements
- It’s easy to continue producing procedural spaghetti code in classes
- Successful adoption of object oriented programming requires knowledge of a few key principles and patterns

Definitions

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

Common terms

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

- Parameter value
- Fluent Interfaces
- Factory method
- Singleton

SOLID principles

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

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

Software goals

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

Agenda

- Introduction
- **Interface vs. Implementation**
- Enums
- Value or Parameter Objects
- Fluent Interfaces
- Builders
- Strong Typed Dynamic Query Interfaces
- Factories
- Facades and Decorators

Implementation

- The implementation of a class consists of any executable code
- Code in constructors, methods, properties GET and SET, destructors
- PRIVATE, PROTECTED an PUBLIC method declarations
- Coding standards promarily impact implementation of a class (e.g. size of methods)
- Developer should be able to alter implementation (optimization, bug fixing) of a class without impacting consuming code
- “Implementation details” should only concern developer of a class, not the developer of code consuming that class

Interface of a class

- Outside view on a class
- Set of all PUBLIC members (constructors, methods, properties, events, variables)
- Parameters of those members
- PRIVATE or PROTECTED Temp-Table and ProDataset's when used as parameters
- Interface of a class should not be changed (at least no breaking changes) as this might impact consuming code
- Only make those members public that need to be invoked by other classes or procedures – once it's public colleagues will use it!

Interface types

- Classes with no implementation, no executable code
- Part of a programming contract (in the sense of contractual design)
- Only PUBLIC members allowed, compiler enforced
- Interfaces define a set of members a class needs to provide so that the class becomes compatible to the Interfaces type, compiler enforced
- Classes can implement multiple interfaces
- Interface types make different implementations exchangeable
- Constructors not part of Interface types
- Interfaces should be small, only few methods

Interface vs. Implementation

- Prefer Interface types for parameters and return value definitions
- Foundation for mocking (Unit Testing)
- Foundation for customization through configuration
- Interfaces define common type without requiring to inherit from a common base class
- Inheritance should be an implementation detail
- Interface may however be inherited by base class

```
CLASS Demo.CustomerBusinessEntity
   INHERITS Consultingwerk.OERA.BusinessEntity
   IMPLEMENTS Consultingwerk.OERA.IBusinessEntity:
```

Naming standards

- Pick one … and live with that – as it must be clear to the whole team
- **.NET:** Interface names starting with a capital I, followed by another capital letter: IBusinessEntity, default implementation (if exists): BusinessEntity
- **Java:** Interface names with no prefix, default implementation sometimes suffixed with “Impl”: BusinessEntity and BusinessEntityImpl

Agenda

- Introduction
- Interface vs. Implementation
- **Enums**
- Value or Parameter Objects
- Fluent Interfaces
- Builders
- Strong Typed Dynamic Query Interfaces
- Factories
- Facades and Decorators

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

```
// Consider this pseudo code - ignoring the need to release locks!

PROCEDURE SetOrderStatus (INPUT pOrderNum AS INTEGER,
                          INPUT pStatus AS INTEGER):

        OpenEdge.Core.Assert:IsPositive(pStatus, 'Order status').

        FIND Order WHERE Order.OrderNum EQ pOrderNum EXCLUSIVE-LOCK.
        ASSIGN Order.OrderStatus = pStatus.
        // what happens when the integer isn't a valid status? How do we know ?
END PROCEDURE.

RUN SetOrderStatus (12345, 0).   // this throws an error via the Assert
RUN SetOrderStatus (12345, 1).   // what is this status?
RUN SetOrderStatus (12345, 2).
```

Defining enums

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

Using enums

```
// Consider this pseudo code - ignoring the need to release locks!

PROCEDURE SetOrderStatus (INPUT pOrderNum AS INTEGER,
                          INPUT pStatus AS Services.Orders.OrderStatusEnum):

        //ensures that we have a known, good status
        OpenEdge.Core.Assert:NotNull(pStatus, 'Order status').

        FIND Order WHERE Order.OrderNum EQ pOrderNum EXCLUSIVE-LOCK.

        ASSIGN Order.OrderStatus = pStatus:GetValue().
        //Alternative way of getting the value
        ASSIGN Order.OrderStatus = INTEGER(pStatus).
END PROCEDURE.

RUN SetOrderStatus (12345, OrderStatusEnum:None).                    // COMPILE ERROR
RUN SetOrderStatus (12345, OrderStatusEnum:Backordered).
RUN SetOrderStatus (12345, OrderStatusEnum:Ordered).
```

Storing Enum Values in the Database

```
DEFINE VARIABLE cValue AS CHARACTER   NO-UNDO .
DEFINE VARIABLE iValue AS INTEGER     NO-UNDO .
DEFINE VARIABLE oEnum  AS WeekdayEnum NO-UNDO .

/* Get Enum value as INTEGER or CHARACTER value */
ASSIGN cValue = STRING (WeekdayEnum:Thursday)
       iValue = INTEGER (WeekdayEnum:Thursday) .

/* Specific Enum */
oEnum = WeekdayEnum:GetEnum(cValue) .
oEnum = WeekdayEnum:GetEnum(4) .

/* More generic */
oEnum = CAST (Progress.Lang.Enum:ToObject
            ("Demo.OoInterfaces.WeekdayEnum":U, cValue),
            Demo.OoInterfaces.WeekdayEnum).
```

Enums vs. System/Master Tables

- Enums provide a fixed set of values determined at compile time
- Enums cannot be extended by end users without altering source code and recompiling
- Enums may be accompanied by a System Table defining further details, such as opening times, # of days in a given year
- Weekday, Month are good examples for Enums
- Color usually doesn’t work well as an Enum (unless you’re Henry Ford)
- OrderStatus may or may not be a good Enum
  - Is there a fixed set of order statuses
  - Can (super-)users define order statuses and their workflow dynamically

Enums and Translation

- An Enum maps an INT64 value to a Character name
- Character name must follow rules of ABL identifiers
  - Cannot contain space or similar white spaces, comma, (semi)colon, etc.
  - Cannot start with a number
- Is not translatable (also not with Translation Manager)
- Character name of an Enum member usually not suited for interaction with users
- Enums require custom solution (application or framework) to provide human-friendly representation including translation

Agenda

- Introduction
- Interface vs. Implementation
- Enums
- **Value or Parameter Objects**
- Fluent Interfaces
- Builders
- Strong Typed Dynamic Query Interfaces
- Factories
- Facades and Decorators

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

- Class with typically no or very little executable code
- Foundation structure of DDD (Domain Driven Design)
- The Primitive Types of an application
- Single record temp-table
  - With inheritance and Interface implementation supported
  - Temp-Tables not really an OO construct in the ABL

Customer value object

```
CLASS Demo.Customer:

    DEFINE PUBLIC PROPERTY CustNum AS INTEGER NO-UNDO
    GET.
    PRIVATE SET.

    DEFINE PUBLIC PROPERTY Name AS CHARACTER NO-UNDO
    GET.
    PRIVATE SET.

    DEFINE PUBLIC PROPERTY City AS CHARACTER NO-UNDO
    GET.
    PRIVATE SET.

    DEFINE PUBLIC PROPERTY PostalCode AS CHARACTER NO-UNDO
    GET.
    PRIVATE SET.
```

Customer constructor

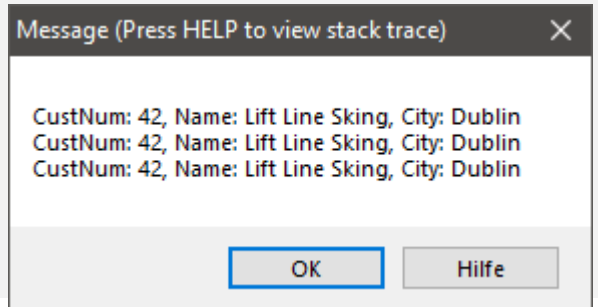
```
CONSTRUCTOR PUBLIC Customer (pCustNum AS INTEGER,
                             pName AS CHARACTER,
                             pCity AS CHARACTER,
                             pPostalCode AS CHARACTER):

    ASSIGN THIS-OBJECT:CustNum = pCustNum
           THIS-OBJECT:Name = pName
           THIS-OBJECT:City = pCity
           THIS-OBJECT:PostalCode = pPostalCode
           .

  END CONSTRUCTOR.
```

Simplify Logging and Debugging

- The ToString() method of Progress.Lang.Object is overriable

```
METHOD PUBLIC OVERRIDE CHARACTER ToString():

  RETURN SUBSTITUTE ("CustNum: &1, Name: &2, City: &3":U,
                     THIS-OBJECT:CustNum,
                     THIS-OBJECT:Name,
                     THIS-OBJECT:City) .
END METHOD.
```

```
oCustomer = NEW Customer (42, "Lift Line Sking", "Dublin", "12345") .

MESSAGE oCustomer SKIP
        STRING (oCustomer) SKIP
        oCustomer:ToString ()
    VIEW-AS ALERT-BOX.
```



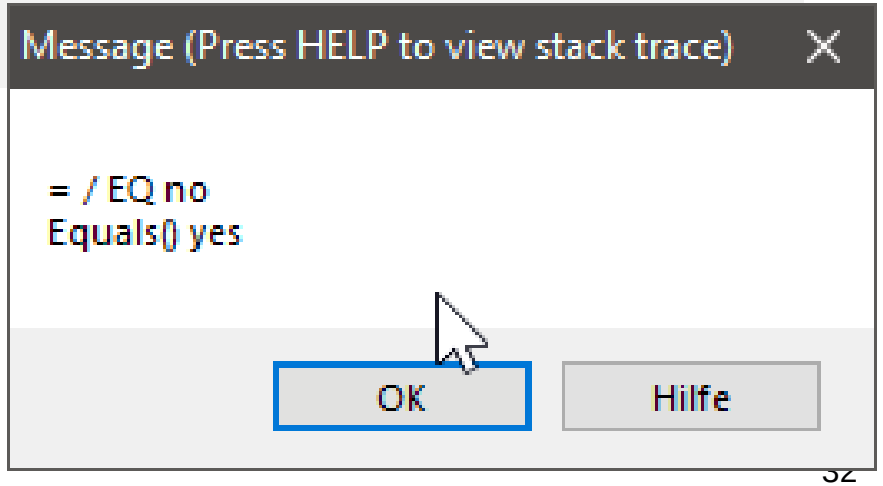
Equality checks

- The Equals() method of Progress.Lang.Object is overriable
- Equals() needs to be invoked, it's not overriding the = or EQ operator

```
METHOD PUBLIC OVERRIDE LOGICAL Equals (objectRef AS Object):

  IF objectRef = THIS-OBJECT THEN RETURN TRUE .
  IF NOT TYPE-OF (objectRef, Customer) THEN RETURN FALSE .

  IF CAST (objectRef, Customer):CustNum    = THIS-OBJECT:CustNum AND
    CAST (objectRef, Customer):Name      = THIS-OBJECT:Name AND
    CAST (objectRef, Customer):City      = THIS-OBJECT:City AND
    CAST (objectRef, Customer):PostalCode = THIS-OBJECT:PostalCode THEN
     RETURN TRUE .

  RETURN FALSE .
END METHOD.
```

Equality checks

```
DEFINE VARIABLE oCustomer1 AS Customer NO-UNDO .
DEFINE VARIABLE oCustomer2 AS Customer NO-UNDO .

oCustomer1 = NEW Customer (42, "Lift Line Sking", "Dublin", "12345") .
oCustomer2 = NEW Customer (42, "Lift Line Sking", "Dublin", "12345") .

MESSAGE "= / EQ" oCustomer1 = oCustomer2        SKIP
      "Equals()" oCustomer1:Equals (oCustomer2)
   VIEW-AS ALERT-BOX.
```



Parameter Objects

- Parameter Objects are Value Objects
- Purpose is to serve as a (single) parameter to a method requiring a complex parameter
- Simplify adding additional parameters by just adding properties to the parameter object
- Parameter Objects can (and should) be used with procedural code

Simplified object creation using CLI Interface

- Command line interfaces trendy in web development

```
> scl-gen Parameter Customer CustNum:integer Name City PostalCode

DLC:      C:\Progress\OpenEdge117_64
Buildfile: c:\Work_STREAM\SmartComponentLibrary\Develop\ABL\Consultingwerk\Studio\Scaffolding\scl-gen.xml

generate:
  [scl-gen] Script Directory:  c:\Work_STREAM\SmartComponentLibrary\Develop\ABL\Consultingwerk\Studio\Scaffolding\
  [scl-gen] Current Directory: C:\Work_STREAM\SmartComponentLibrary\Develop\ABL\demo
  [scl-gen] Template:         Parameter
  [scl-gen] TemplateParameter: CustNum:integer Name City PostalCode
  [scl-gen] Name:             Customer
  [scl-gen] PROPATH:          c:\Work_STREAM\SmartComponentLibrary\Develop\ABL
  [scl-gen]
  [scl-gen] SmartComponent Library code generator.
  [scl-gen] Consultingwerk Internal Development / 117_64
  [scl-gen] (c)2008-2018 Consultingwerk Ltd. - All rights reserved.
  [scl-gen]
  [scl-gen] Writing to: C:\Work_STREAM\SmartComponentLibrary\Develop\ABL\demo\Customer.cls
```

Interfaces for Value Objects?

- Not always a requirement - but a good habit in my opinion
- It's not a frequent requirement to swap the implementation
  - Typically no executable code, so the demand for mocking (Unit Tests) or customization is generally low
- However you might have a *Supplier* type and a *Customer* type
- Customer and Supplier both have Address fields
- *Address* can define a *super-type* for Customer and Supplier through
  - Inheritance
  - Implementing IAddress

Agenda

- Introduction
- Interface vs. Implementation
- Enums
- Value or Parameter Objects
- **Fluent Interfaces**
- Builders
- Strong Typed Dynamic Query Interfaces
- Factories
- Facades and Decorators

Fluent Interfaces

> In software engineering, a fluent interface .. is a method for designing object oriented APIs based extensively on method chaining with the goal of making the readability of the source code close to that of ordinary written prose, essentially creating a domain-specific language within the interface.

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

Fluent Interfaces

- Useful when typically a sequence of calls into an object is required
- Order of those calls typically not relevant
- Might be used to initialize (parameter) objects
- Typically more verbose than a number of parameters, makes understanding of code by other developers trivial
- But – usually confusing to developers that haven't seen this before

Example

```
DEFINE VARIABLE oParameter AS FluentParameterSample NO-UNDO .

/* Constructor with parameters */
oParameter = NEW FluentParameterSample (11, 42, 1/1/2018, 12/31/2018) .

/* Fluent Interface in a single line */
oParameter = (NEW FluentParameterSample()):CustomerNumber(11):ItemNumber(42):
StartDate (1/1/2018):EndDate (12/31/2018) .

/* Block formatting of Fluent Interface */
oParameter = (NEW FluentParameterSample())
                :CustomerNumber (11)
                :ItemNumber (42)
                :StartDate (1/1/2018)
                :EndDate (12/31/2018) .
```

Implementation of Example

- VOID methods simply return THIS-OBJECT instead
- Use methods as alternative PROPERTY SET

```
METHOD PUBLIC FluentParameterSample ItemNumber (piItemNumber AS INTEGER):

   ASSIGN THIS-OBJECT:ItemNumber = piItemNumber .
   RETURN THIS-OBJECT .

END METHOD.
```

Agenda

- Introduction
- Interface vs. Implementation
- Enums
- Value or Parameter Objects
- Fluent Interfaces
- **Builders**
- Strong Typed Dynamic Query Interfaces
- Factories
- Facades and Decorators

Builders

> The Builder is a design pattern designed to provide a flexible solution to various object creation problems in object-oriented programming. The intent of the Builder design pattern is to separate the construction of a complex object from its representation. It is one of the Gang of Four design patterns.

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

OpenEdge HTTP Client

- Extensive ABL implementation of the HTTP protocol
- GET/POST/PUT/PATCH/DELETE to HTTP servers, REST services, SOAP, …
- Object model similar to the Web Handlers (HTTP service implementation)
- OpenEdge 11.6 + (available but unsupported in 11.5)
- Many parameters ….
- Basic syntax on next slide

```
DEFINE VARIABLE oLibrary       AS IHttpClientLibrary       NO-UNDO .
DEFINE VARIABLE oClient        AS IHttpClient              NO-UNDO .
DEFINE VARIABLE oRequest       AS IHttpRequest             NO-UNDO .
DEFINE VARIABLE oResponse      AS IHttpResponse            NO-UNDO .
DEFINE VARIABLE oRequestBuilder AS RequestBuilder          NO-UNDO .
DEFINE VARIABLE cUri           AS CHARACTER                NO-UNDO .

oLibrary = ClientLibraryBuilder:Build():Library.
oClient = ClientBuilder:Build():UsingLibrary(oLibrary):Client .

ASSIGN cUri = SUBSTITUTE  ("http://data.fixer.io/api/latest?access_key=&1&&base=&2&&symbols=&3":U,
  FIXER_API_KEY, pcFromSymbol, pcToSymbol) .

oRequestBuilder = RequestBuilder:Get(cURI):AcceptJson() .
oRequest = oRequestBuilder:Request.
oResponse = oClient:Execute (oRequest).

RETURN CAST (oResponse:Entity, JsonObject):GetJsonObject ("rates":U):GetDecimal (pcToSymbol) .
```

Demo

- Using the HTTP Client to retrieve currency rates

A closer look

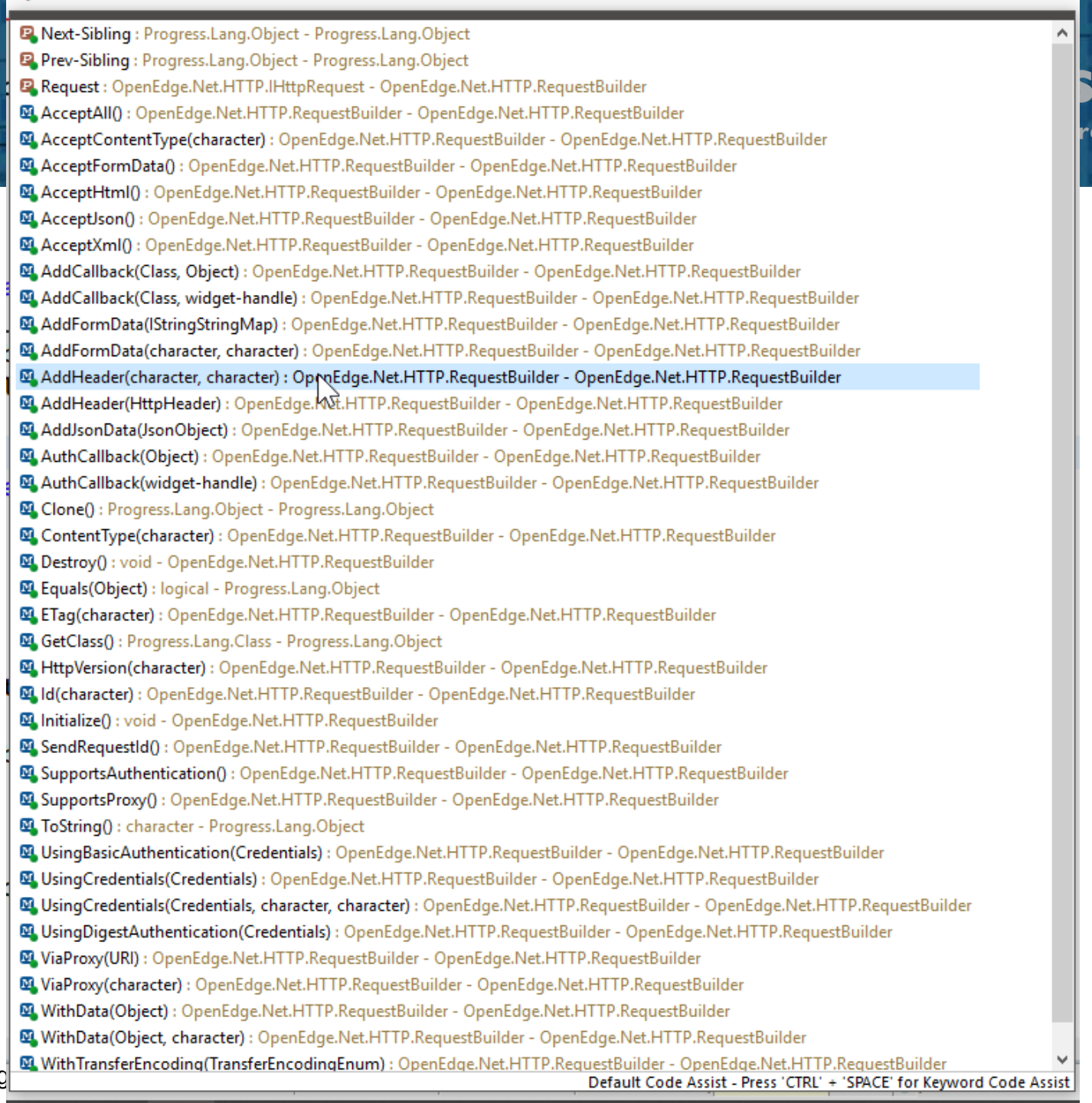
- *RequestBuilder* – static reference to the RequestBuilder class
- Each method returns an instance of the RequestBuilder
- *Request* property returns reference to the IHttpRequest instance that was built
- We don't know, we don't need to know the class implementing IHttpRequest – that's if at all a concern of the RequestBuilder class

```
DEFINE VARIABLE oRequest        AS IHttpRequest          NO-UNDO .
DEFINE VARIABLE oRequestBuilder AS RequestBuilder        NO-UNDO .

oRequestBuilder = RequestBuilder:Get(cURI):AcceptJson() .
oRequest = oRequestBuilder:Request.
```

A simplification

- In many cases we don't require the reference to the RequestBuilder

```
DEFINE VARIABLE oRequest       AS IHttpRequest          NO-UNDO .

oRequest = RequestBuilder:Get(cURI):AcceptJson():Request.
```



Fluent Builder Interface

- Builder may use Fluent Interfaces (previous section) to allow for method chaining

```
oRequest = RequestBuilder:Get(cURI)
                    :AddHeader ("some-name", "42")
                    :SupportsProxy()
                    :AcceptJson()
                    :Request .
```

Agenda

- Introduction
- Interface vs. Implementation
- Enums
- Value or Parameter Objects
- Fluent Interfaces
- Builders
- **Strong Typed Dynamic Query Interfaces**
- Factories
- Facades and Decorators

Dynamic Queries

- Dynamic Queries are great - Dynamic Queries are bad
- Dynamic Queries are required for layered architectures
- Dynamic Queries are required for flexible filtering by users or consumers
- Dynamic Queries are hard to debug
- Typos in Dynamic Queries will bite you only during runtime – both for field names and syntax or argument data types
- Debugging in production not really the best method

Strong Typed Query Interface

- Generated for Business Entities
- Compliant with the CCS-BE Structured Query specification
- Support standard ABL query operators for fields based on Data-Type
- Support additional operators based on Data-Type like *InRange* or *InList*
- Query defined as list of object describing each individual criteria
- Simplifies Query manipulation, validation and optimization

- Our implementation uses Fluent Interface – but no Builder
- Using the builder seemed too much code generation

OrderQuery sample

```
DEFINE VARIABLE oQuery AS OrderQuery NO-UNDO .

oQuery = NEW OrderQuery () .

oQuery:CustNum:Eq(42)
     :Carrier:Eq("UPS") .

MESSAGE oQuery:ToQueryString (TRUE)
   VIEW-AS ALERT-BOX.
```

```
oQuery = NEW OrderQuery (42) . /* select by OrderNum */
```

OrderQuery sample

```
oQuery = NEW OrderQuery () . /* select by OrderNum */

oQuery:CustNum:Eq(42):And ((NEW OrderQuery()):OrderStatus:Eq("Shipped")
                          :Or:OrderStatus:Eq("Ordered"))
    :And:Carrier:Eq("UPS") .
```

```
DEFINE VARIABLE cOrderStatus AS CHARACTER NO-UNDO EXTENT
   INITIAL ["Shipped", "Ordered"].

oQuery = NEW OrderQuery () . /* select by OrderNum */

oQuery:CustNum:Eq(42)
    :OrderStatus:InList (cOrderStatus)
    :Carrier:Eq("UPS") .
```

Demo

- Strong Typed but Dynamic Queries

Agenda

- Introduction
- Interface vs. Implementation
- Enums
- Value or Parameter Objects
- Fluent Interfaces
- Builders
- Strong Typed Dynamic Query Interfaces
- **Factories**
- Facades and Decorators

Factories

- Factories have the sole responsibility of creating object instances
- They can/should be the only place knowing requirements to create object instances
- Separate the concern of creating an Object instance from using it
- May be based on configuration
- May support Unit Testing and *Mocking*

Different Implementations

- Factory Service: CreateObject (Type) – Type is typically an Interface or Enum etc.
- Factory Methods
- Abstract Factories

- Avoid direct usage of NEW on unrelated type in your code

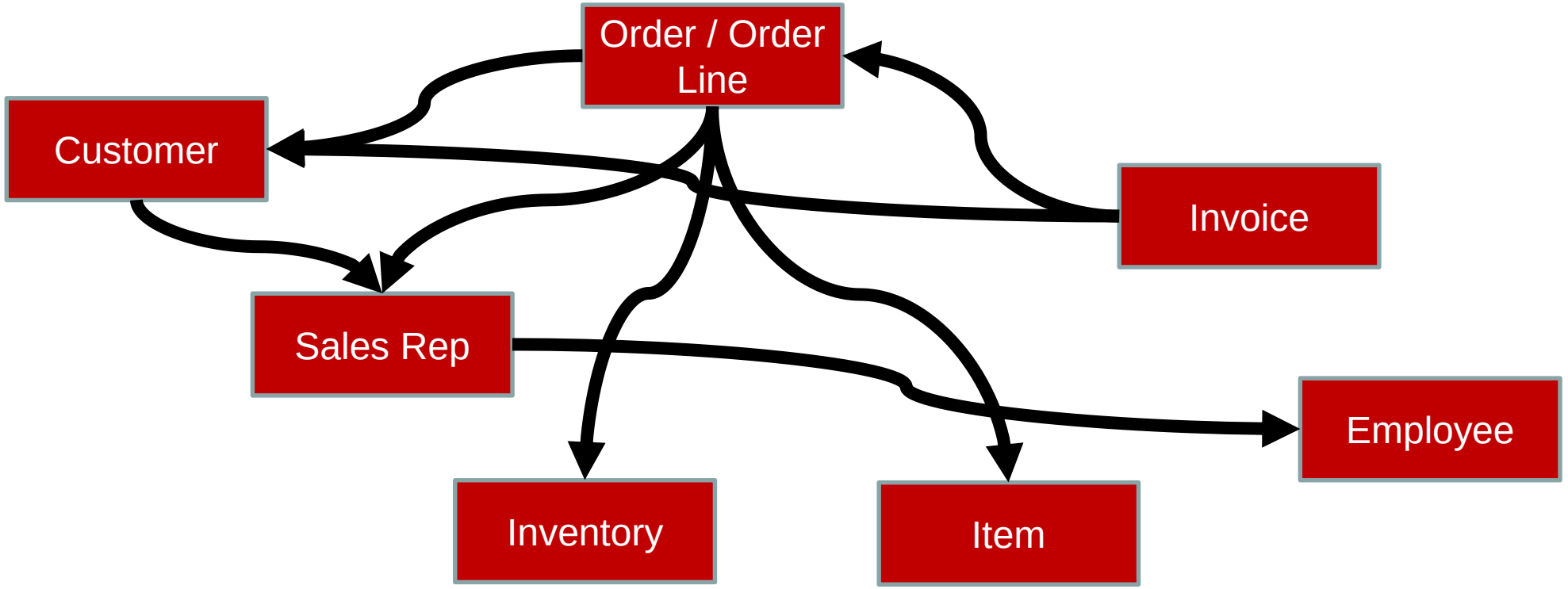
Agenda

- Introduction
- Interface vs. Implementation
- Enums
- Value or Parameter Objects
- Fluent Interfaces
- Builders
- Strong Typed Dynamic Query Interfaces
- Factories
- **Facades and Decorators**

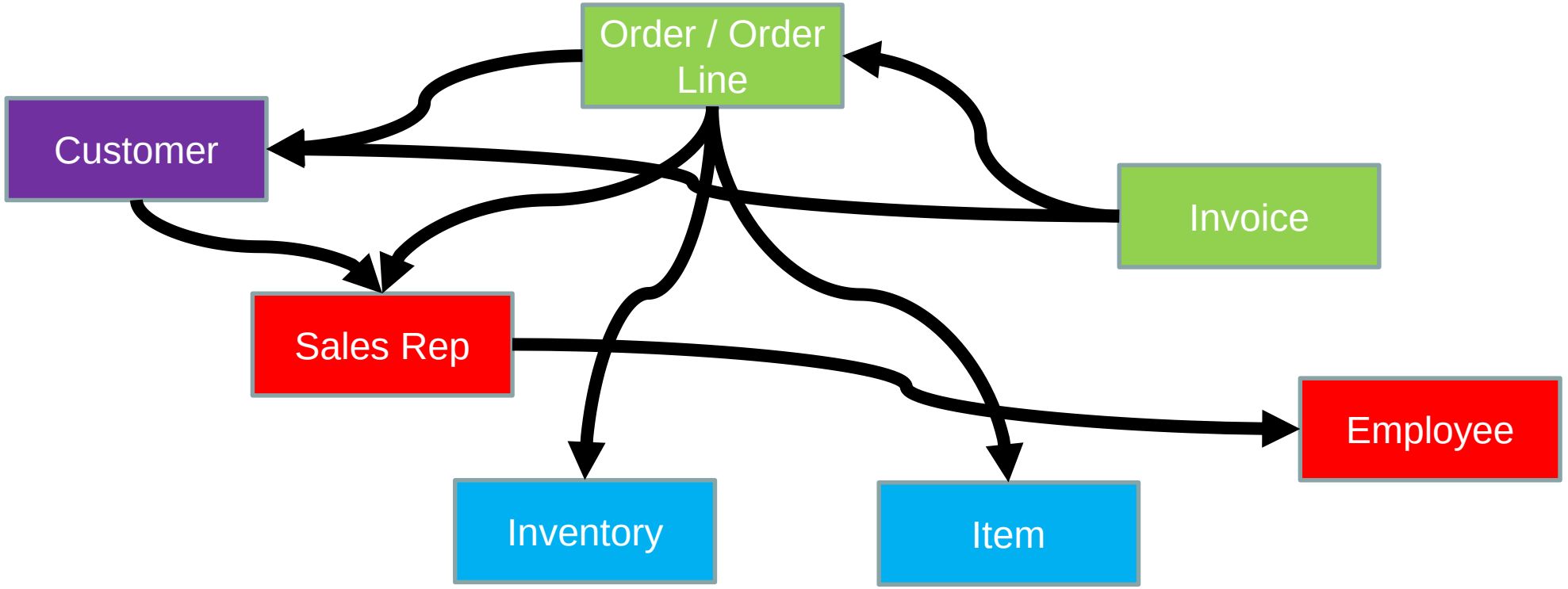
Facades

> The **facade pattern** (also spelled *façade*) is a software-design pattern commonly used with object-oriented programming. Analogous to a facade in architecture, a facade is an object that serves as a front-facing interface masking more complex underlying or structural code.
>
> A facade can improve the readability and usability of a software library by masking interaction with more complex components behind a single API
>
> https://en.wikipedia.org/wiki/Facade_pattern

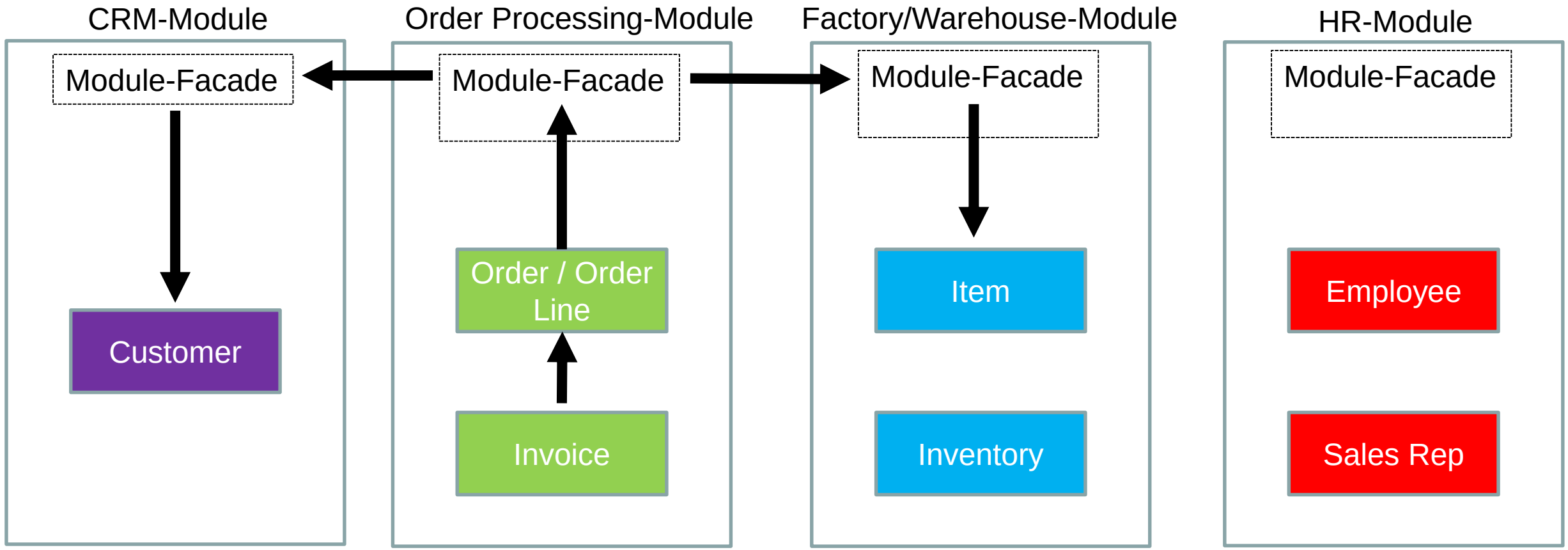
The OO Spaghetti Monster



Start ordering your objects in packages / modules



Divide and conquer – Modules as sub-systems



Open/Close Principle

- Design as open for enhancement – while closed for modifications
- Design to a contract (Interface) on the sub-system level, not just a single class
- Modules manage complexity and impact of change
- A change in functionality in one module does not require changes to other modules
- Simplifies testing. Allows mocking of a whole sub-system

Open/Close Principle

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

Decorators

> In object-oriented programming, the **decorator pattern** is a design pattern that allows behavior to be added to an individual object, dynamically, without affecting the behavior of other objects from the same class. The decorator pattern is often useful for adhering to the Single Responsibility Principle, as it allows functionality to be divided between classes with unique areas of concern.
>
> https://en.wikipedia.org/wiki/Decorator_pattern

Decorator Pattern

- Allows extension of an object instance through another object instance
- Requires objects to implement the same Interface
  - Decorated object implementing with standard functionality
  - Decorator implementing behavior by calling into methods of the decorated object plus X
- Decorator can replace or extend the original implementation
- Multiple decorators may be applied to one object
- Configuration/factory used to abstract the creation of decorator or base object

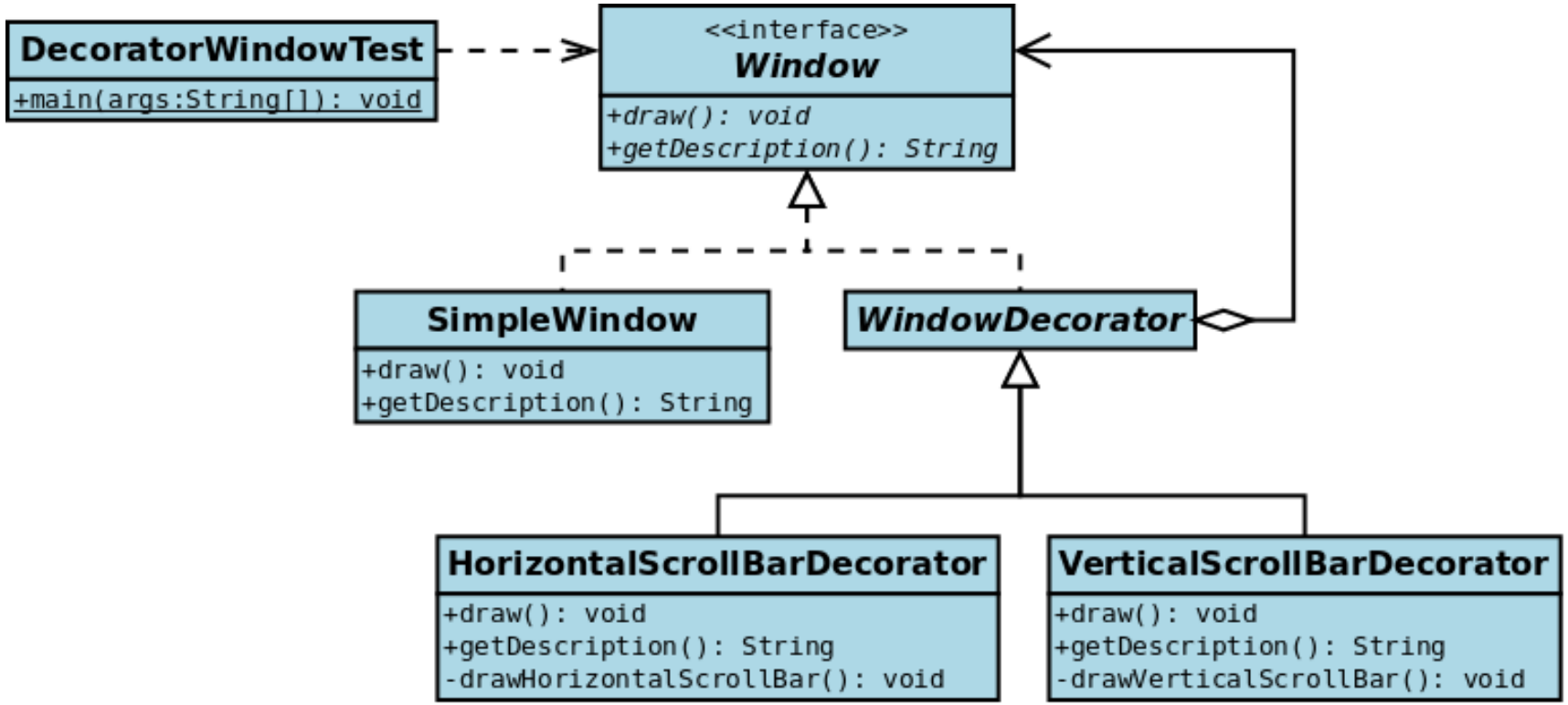
Decorator Example

```
CLASS Demo.OoInterfaces.Decorator.PriceCalculatorDiscountDecorator
   IMPLEMENTS IPriceCalculator:

   DEFINE PRIVATE VARIABLE oDecorated AS IPriceCalculator NO-UNDO .

   CONSTRUCTOR PUBLIC PriceCalculatorDiscountDecorator (poDecorated AS IPriceCalculator):
      ASSIGN oDecorated = poDecorated .
   END CONSTRUCTOR.

   METHOD PUBLIC DECIMAL CalculateSalesPrice (poParameter AS IPriceCalculationParameter):
      DEFINE VARIABLE deResult AS DECIMAL NO-UNDO.

      deResult = oDecorated:CalculateSalesPrice (poParameter) .

      FIND Customer WHERE Customer.CustNum = poParameter:CustomerNumber .

      RETURN deResult * (1 - Customer.Discount / 100) .
   END METHOD.
```

Decorator vs. Inheritance

- Inheritance is static – adding a decorator is more dynamic
- Decorator can be added to an existing object instance at runtime
- Decorator references decorated object, implements all methods, typically calling the method of the decorated object
- Decorator pattern may serve as method to achieve multiple inheritance
- Decorator is not the same instance/reference – whereas when using inheritance there's only a single instance made from the base and child class

Decorator sample



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

Questions


Consultingwerk
software architecture and development