Progress®

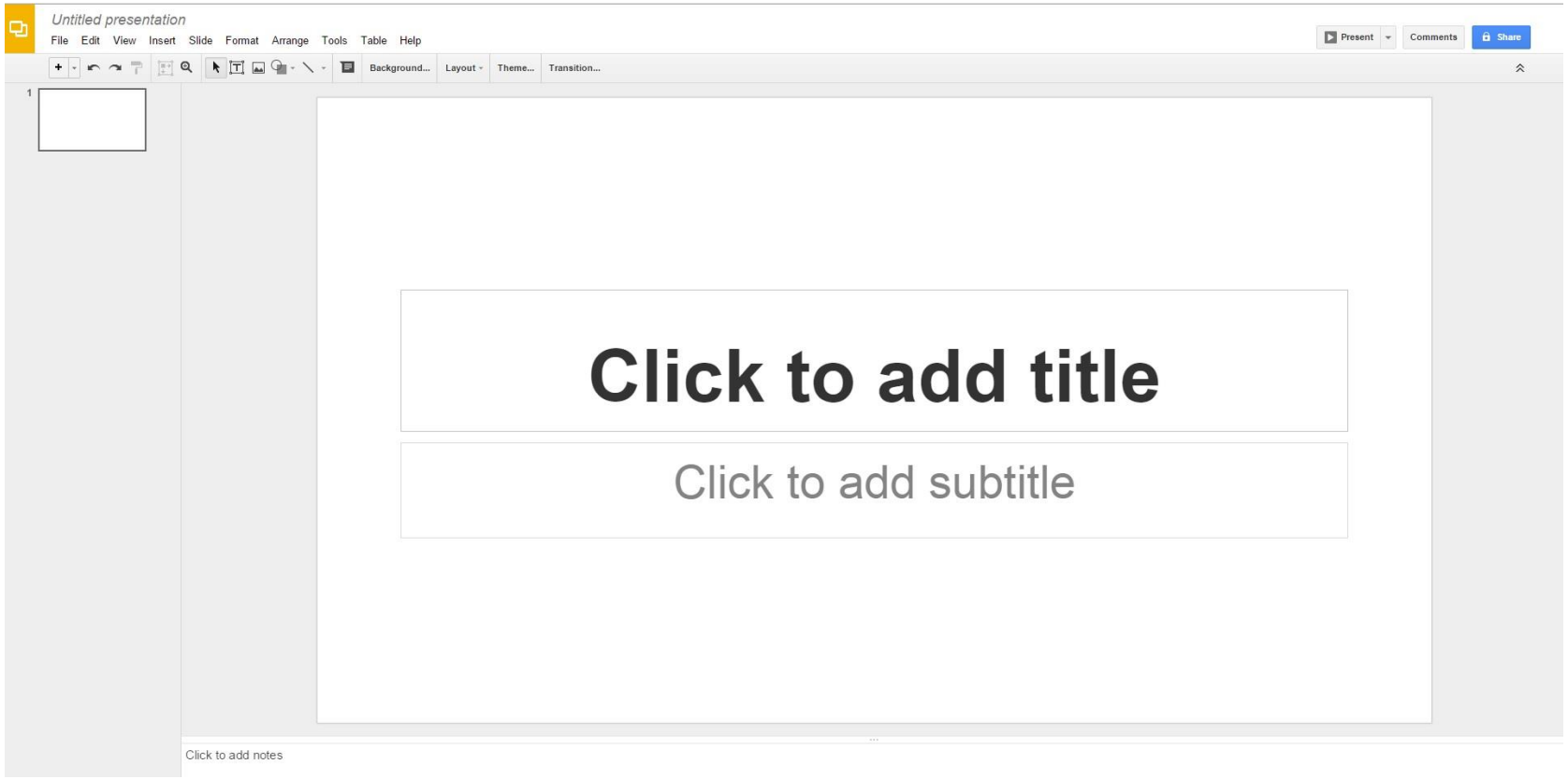
Basic design patterns

… in OO ABL

Roland de Pijper
Principal consultant PS Benelux
20 Juni 2019

Goal of this Presentation

- Design and implement a “Presentation”



First of all: determine the right ‘size’

- Granularity has changed over the years

Right size

Focus = THIS presentation

Separation of concerns

‘Invented’ in 1974

- Modularity
- Encapsulation
- Information hiding

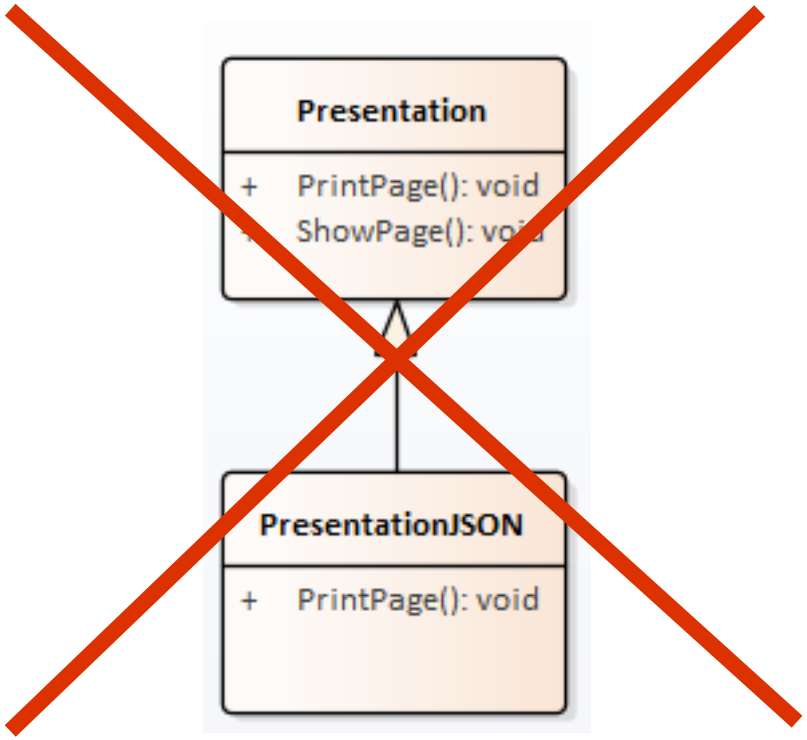
Solution: The code

- Has the ability to show a page
- Can print itself in pdf format

```
CLASS EMEAPUGPresentation:

  METHOD PUBLIC VOID PrintPage( ):
     /* OUTPUT TO LPT1: */
  END METHOD.

 METHOD PUBLIC VOID ShowPage( ):
     MESSAGE "Hello world“.
  END METHOD.

END CLASS.
```

New requirement!

- Print the presentation to a generic format: JSON

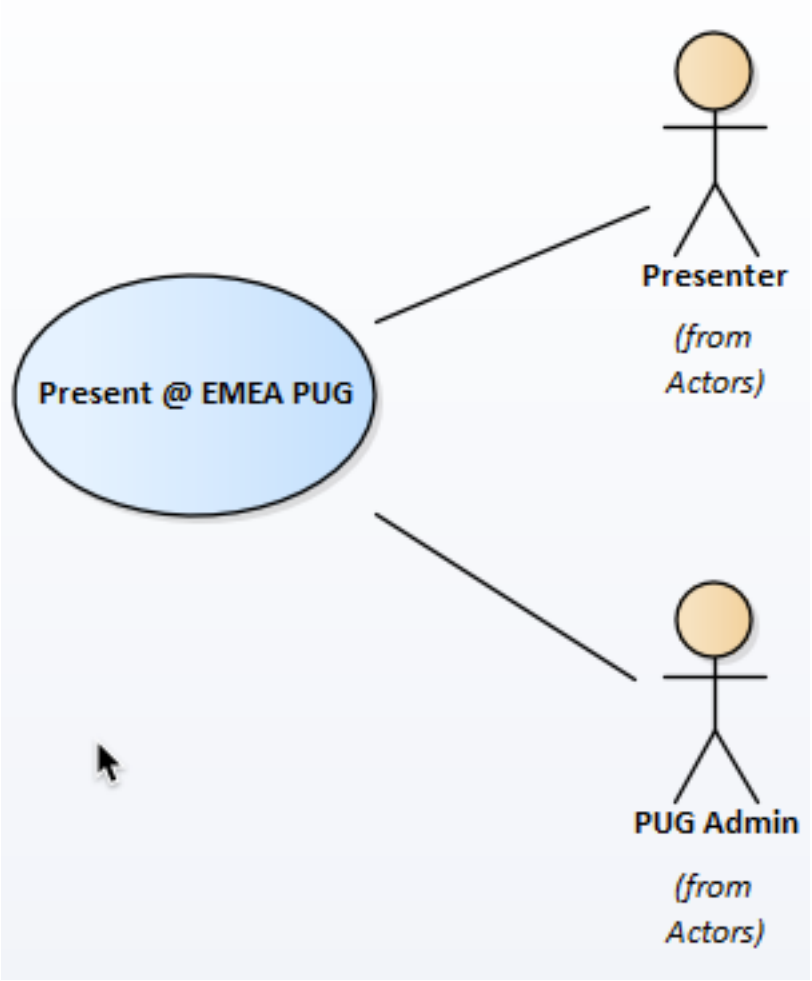


Single Responsibility Principle

A class should have only one reason to change.

So a responsibility is a family of functions that serves one particular actor.

An actor for a responsibility is the single source of change for that responsibility.



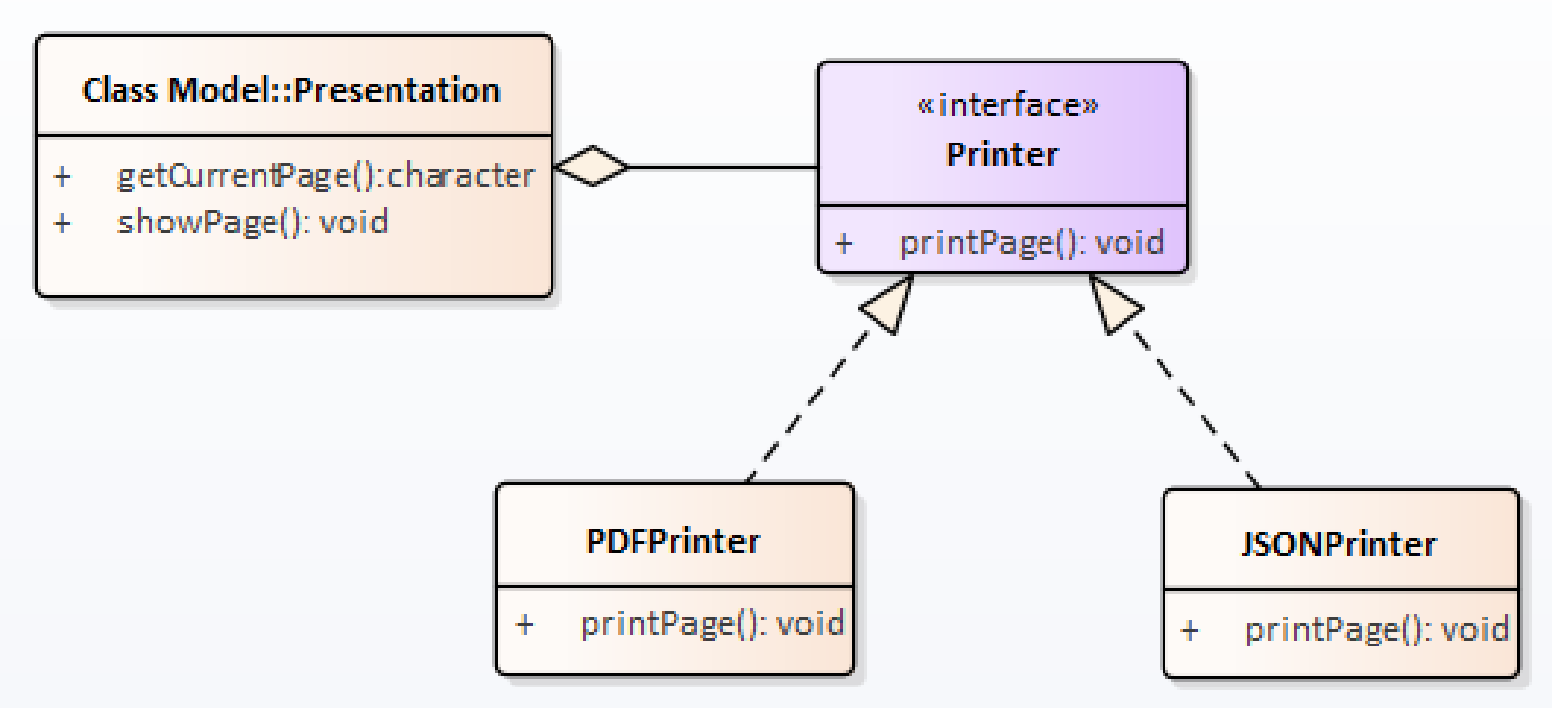
Design patterns to the rescue

Here, take a cookie!

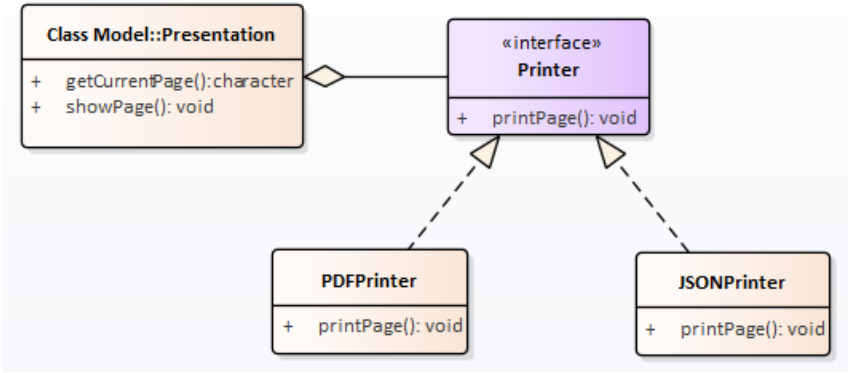
- A design pattern is like a recipe to bake cookies

Solution: The pattern

The Strategy Design Pattern



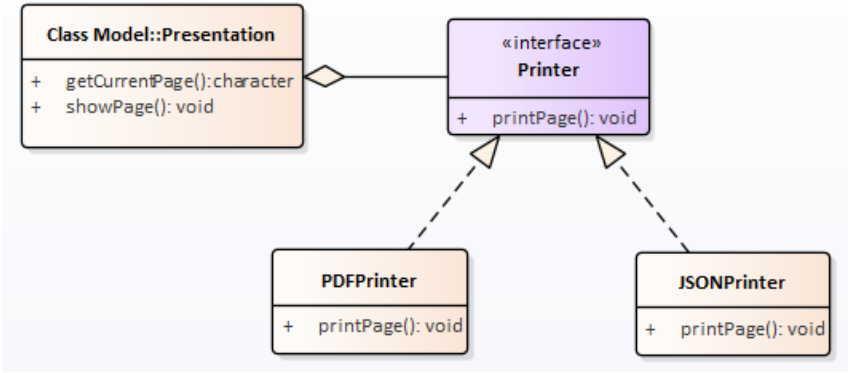
Solution: The code



CLASS **Presentation**:

```
    METHOD PUBLIC CHARACTER getCurrentPage( ):
        RETURN "This is not a presentation".
    END METHOD.


    METHOD PUBLIC VOID ShowPage( ):
        MESSAGE "Hello world“.
    END METHOD.

END CLASS.
```

Solution: The code

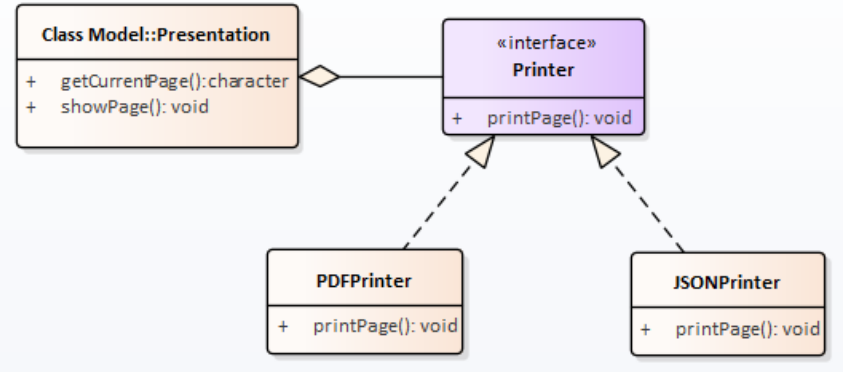


INTERFACE **Printer**:

METHOD PUBLIC VOID **printPage**(ipcText AS CHARACTER).

END INTERFACE.

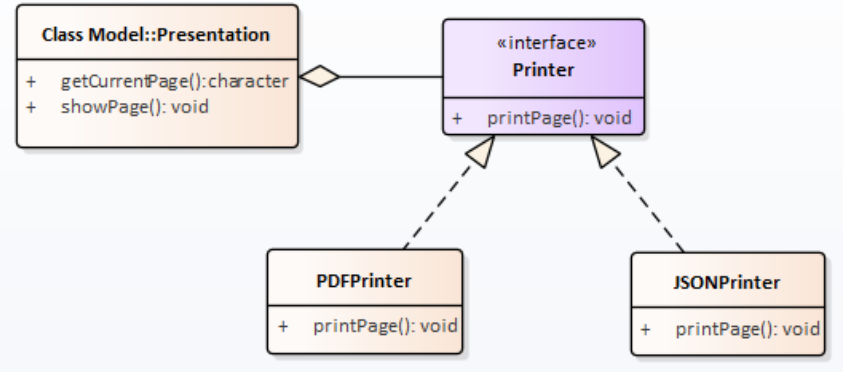
Solution: The code



CLASS **PDFPrinter** IMPLEMENTS **Printer**:

```
METHOD PUBLIC VOID printPage(INPUT ipcText AS CHARACTER):
    /* create output in PDF format */
     MESSAGE ipcText " in PDF format"
    VIEW-AS ALERT-BOX.
END METHOD.

END CLASS.
```

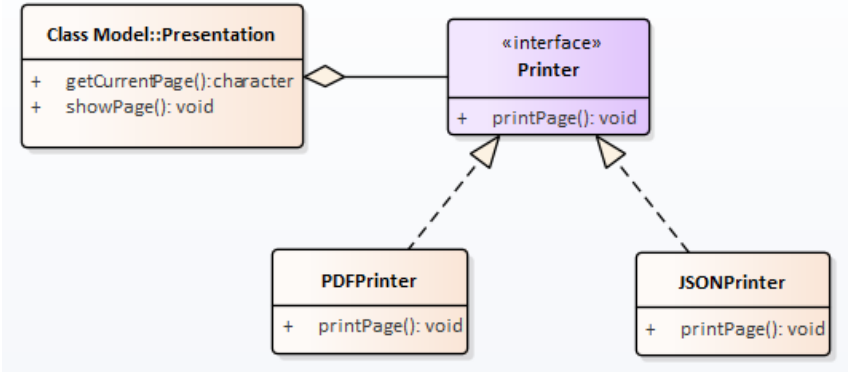
Solution: The code



CLASS **JSONPrinter** IMPLEMENTS **Printer**:

```
METHOD PUBLIC VOID printPage(INPUT ipcText AS CHARACTER):
    /* create output in JSON format */
    MESSAGE ipcText " in JSON format"
    VIEW-AS ALERT-BOX.
END METHOD.

END CLASS.
```

Solution: The code

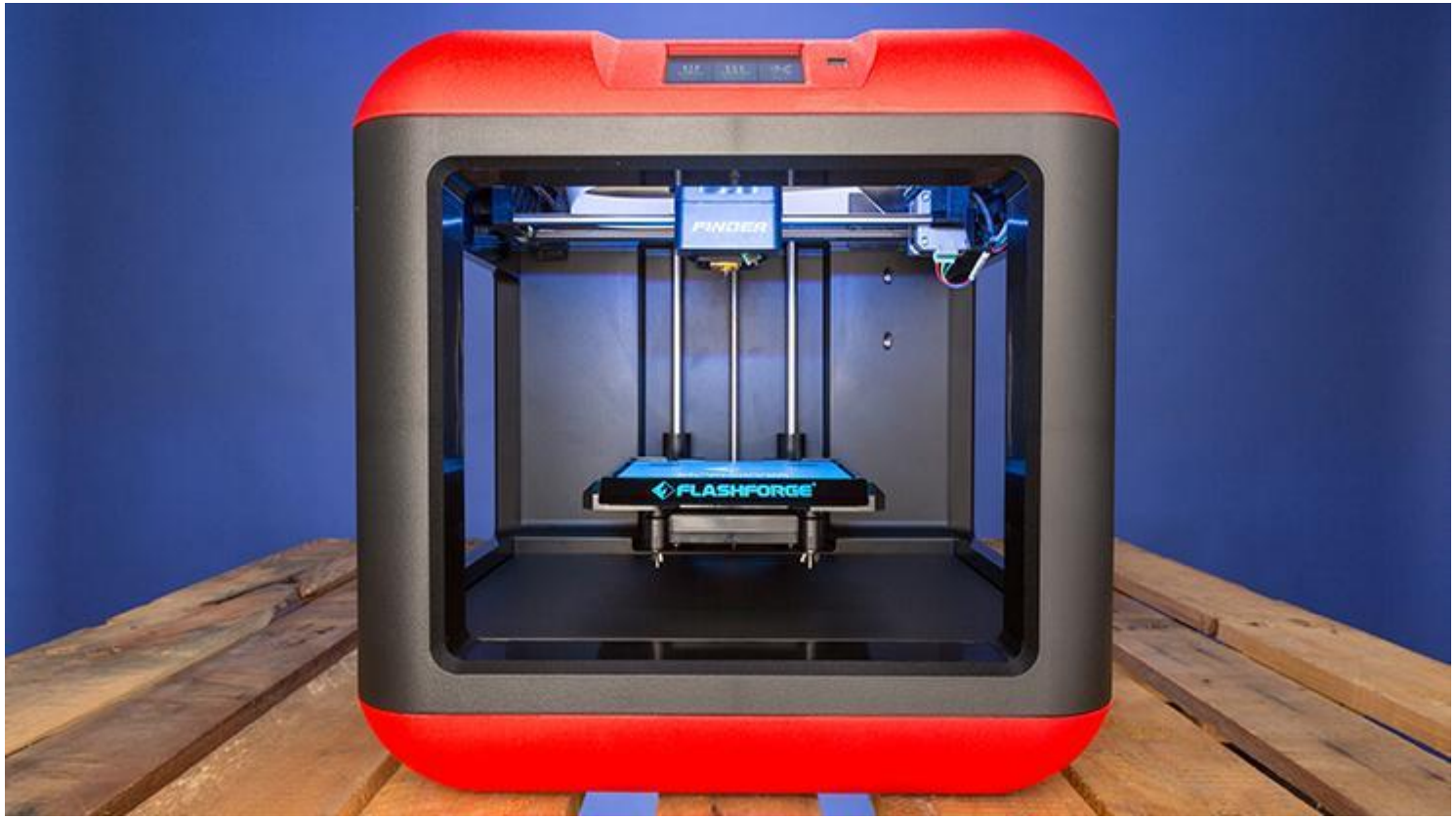



```
DEFINE VARIABLE objPresentation AS Presentation.
DEFINE VARIABLE objPDFPrinter   AS  PDFPrinter .
DEFINE VARIABLE objJSONPrinter AS JSONPrinter NO-UNDO.

ASSIGN objPresentation = NEW Presentation()
       objPDFPrinter   = NEW PDFPrinter()
       objJSONPrinter = NEW JSONPrinter().

objPDFPrinter:printPage(objPresentation:getCurrentPage()).
objJSONPrinter:printPage(objPresentation:getCurrentPage()).
```

Solution: The demo

There’s more to a printer than meets the eye…

- setRGBValues
- choosePaper
- chooseFormat
- resetPage
- …



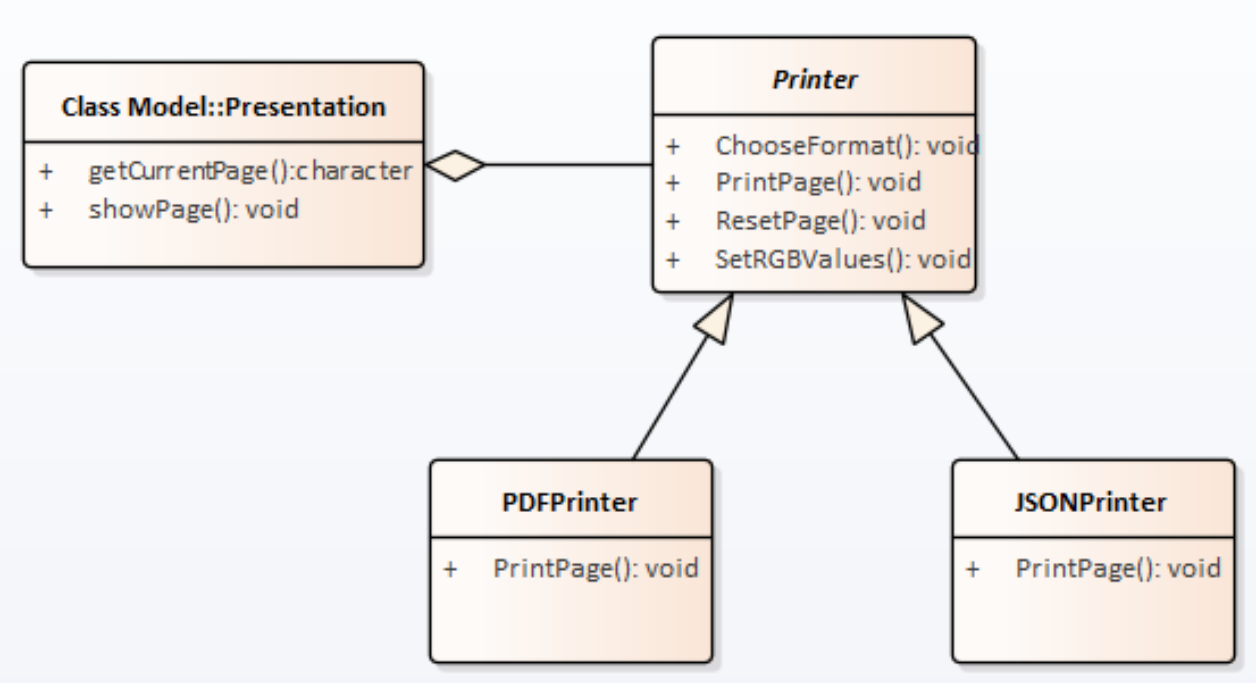
Open Closed Principle

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

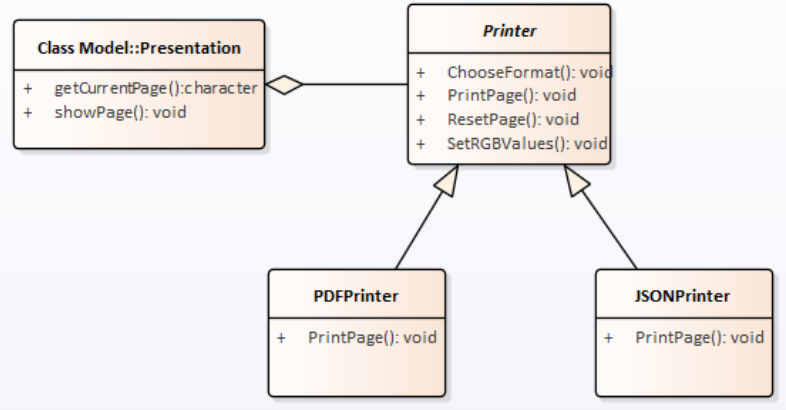
Design modules, classes and functions in a way that when a new functionality is needed, we should not modify our existing code but rather write new code that will be used by existing code.

Solution: The pattern

- The Template Design Pattern



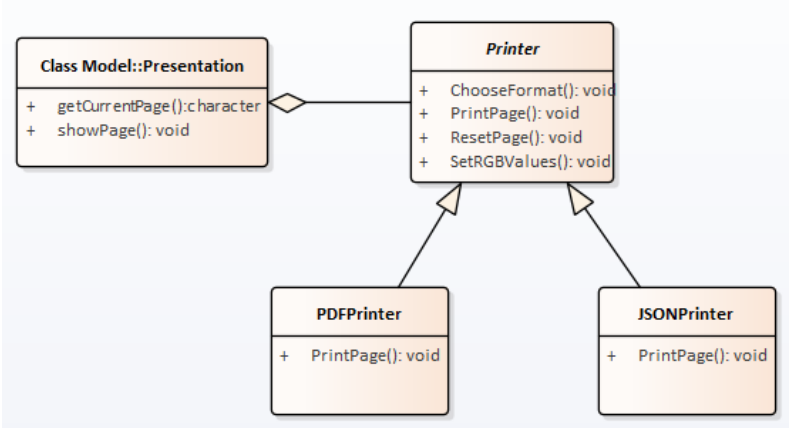
Solution: The code



CLASS **Presentation**:

```
    METHOD PUBLIC CHARACTER getCurrentPage( ):
        RETURN "This is not a presentation".
    END METHOD.

    METHOD PUBLIC VOID ShowPage( ):
        /* MESSAGE "Hello world" */
    END METHOD.

END CLASS.
```

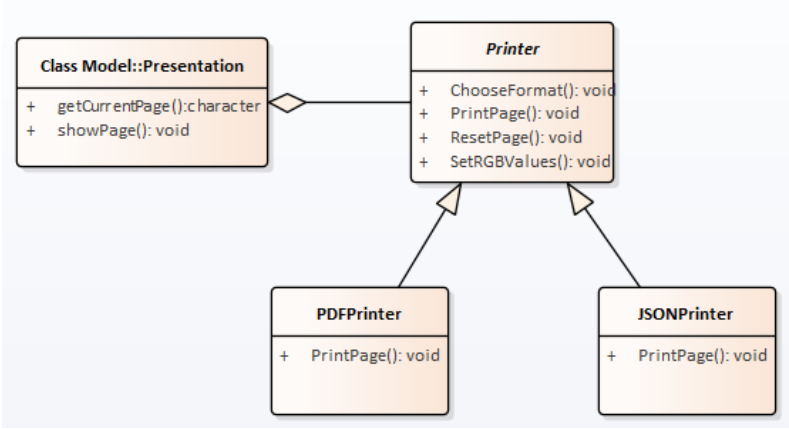
Solution: The code



CLASS **Printer ABSTRACT**:

```
  METHOD PUBLIC VOID ChooseFormat():
  END METHOD.

  METHOD PUBLIC VOID ResetPage():
    MESSAGE "This page has been reset“ VIEW-AS ALERT-BOX.
END METHOD.

  METHOD PUBLIC VOID SetRGBValues():
  END METHOD.

  METHOD PUBLIC VOID printPage(ipcText AS CHARACTER):
    MESSAGE ipcText “ in undefined format“ VIEW-AS ALERT-BOX.
  END METHOD.

END CLASS.
```

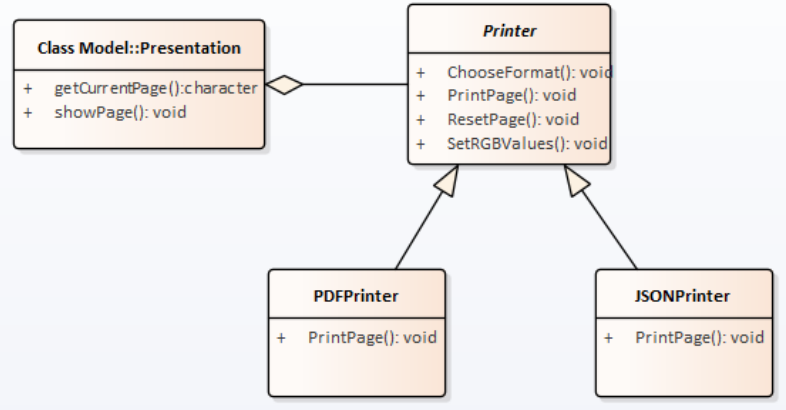
Solution: The code



CLASS **PDFPrinter** INHERITS **Printer**:

```
METHOD PUBLIC OVERRIDE VOID printPage
                (INPUT ipcText AS CHARACTER):
   /* create output in PDF format */
    MESSAGE ipcText " in PDF format"
   VIEW-AS ALERT-BOX.
END METHOD.

END CLASS.
```

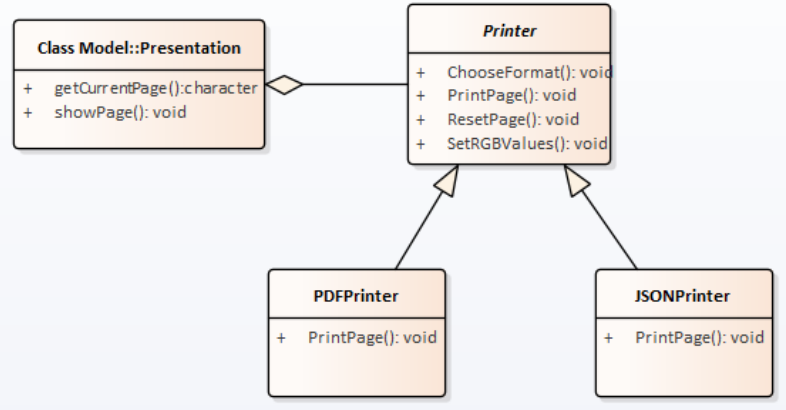
Solution: The code



CLASS **JSONPrinter** INHERITS **Printer**:

```
METHOD PUBLIC OVERRIDE VOID printPage
                (INPUT ipcText AS CHARACTER):
  /* create output in JSON format */
  MESSAGE ipcText " in JSON format"
  VIEW-AS ALERT-BOX.
END METHOD.

END CLASS.
```

Solution: The code




```
DEFINE VARIABLE objPresentation AS Presentation.
DEFINE VARIABLE objPDFPrinter   AS PDFPrinter.
DEFINE VARIABLE objJSONPrinter AS JSONPrinter NO-UNDO.

ASSIGN objPresentation = NEW Presentation()
       objPDFPrinter    = NEW PDFPrinter()
       objJSONPrinter = NEW JSONPrinter().

objPDFPrinter:printPage(objPresentation:getCurrentPage()).
objJSONPrinter:printPage(objPresentation:getCurrentPage()).
objPDFPrinter:ResetPage().
```

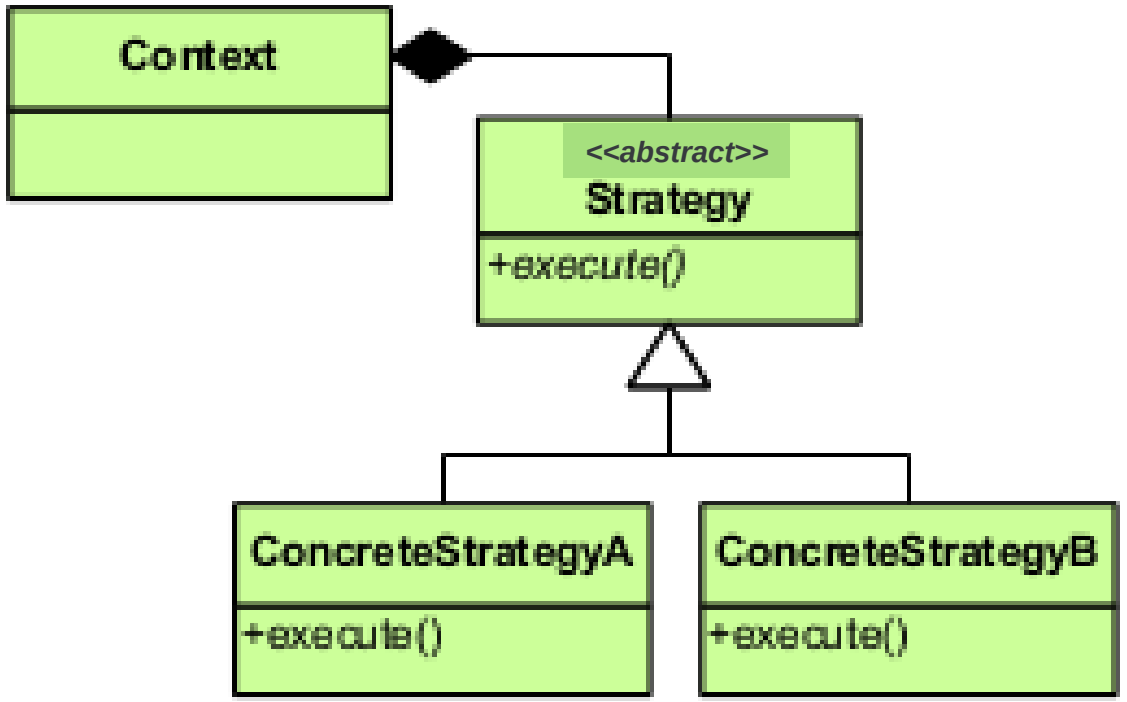
Solution: The demo

And there’s more: SOLID

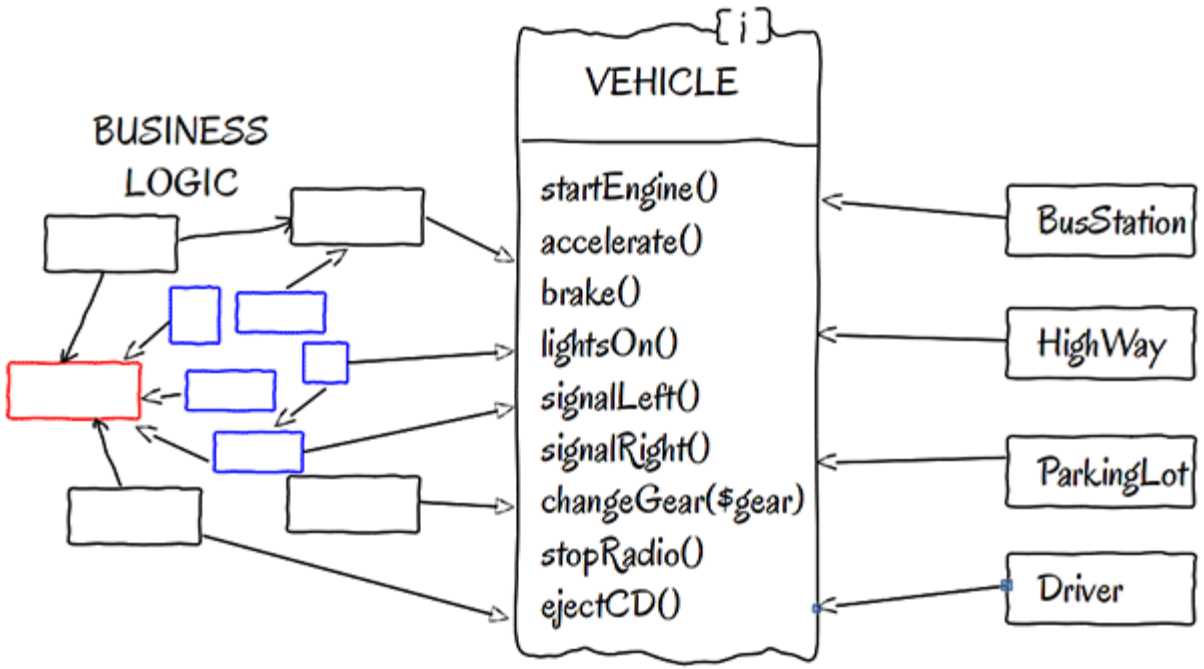
- Single Responsibility Principle (SRP)
- Open Closed Principle (OCP)
- **Liskov Substitution Principle (LSP)**
- **Interface Segregation Principle (ISP)**
- **Dependency Inversion Principle (DIP)**

Liskov Substitution Principle (LSP)

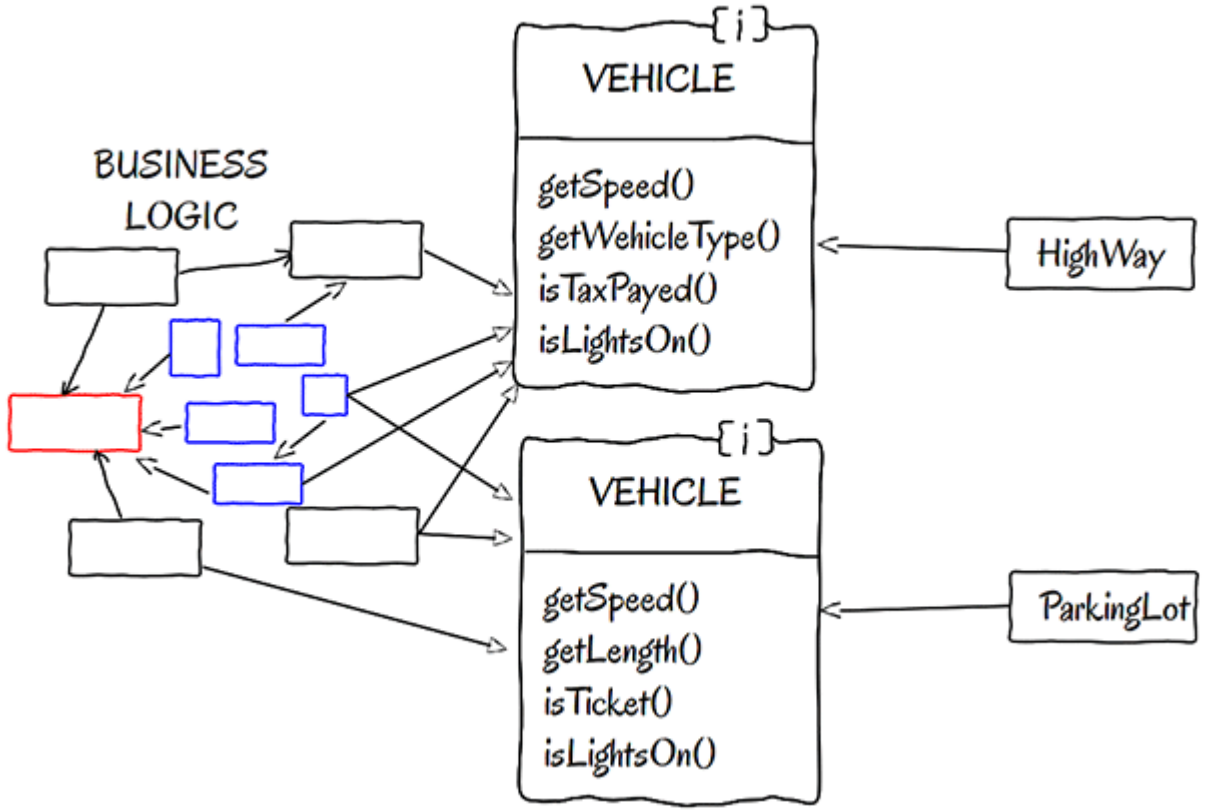
Subtypes must be substitutable for their base types.



Interface Segregation Principle (ISP)



Interface Segregation Principle (ISP)

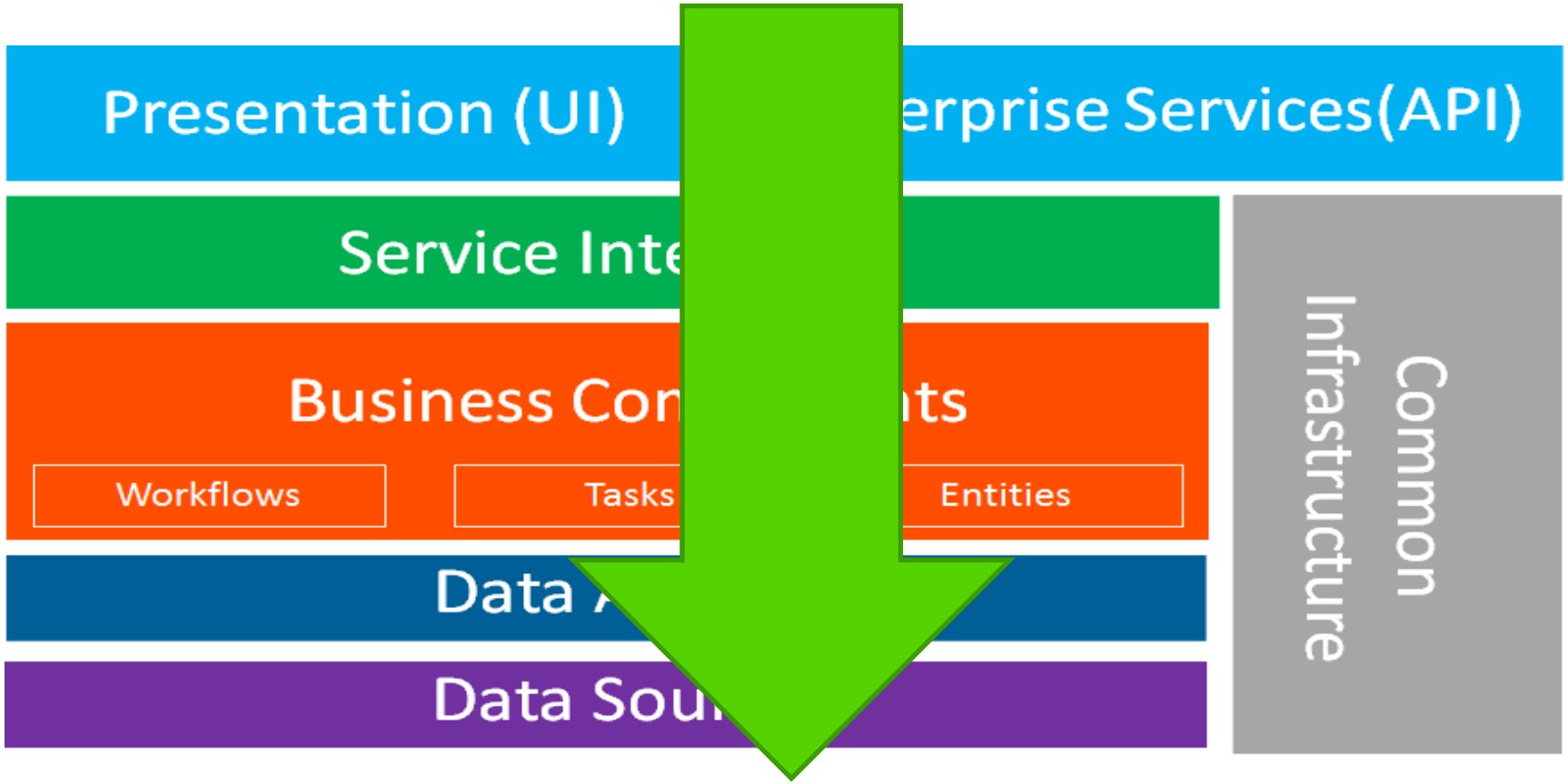


Dependency Inversion Principle (DIP)

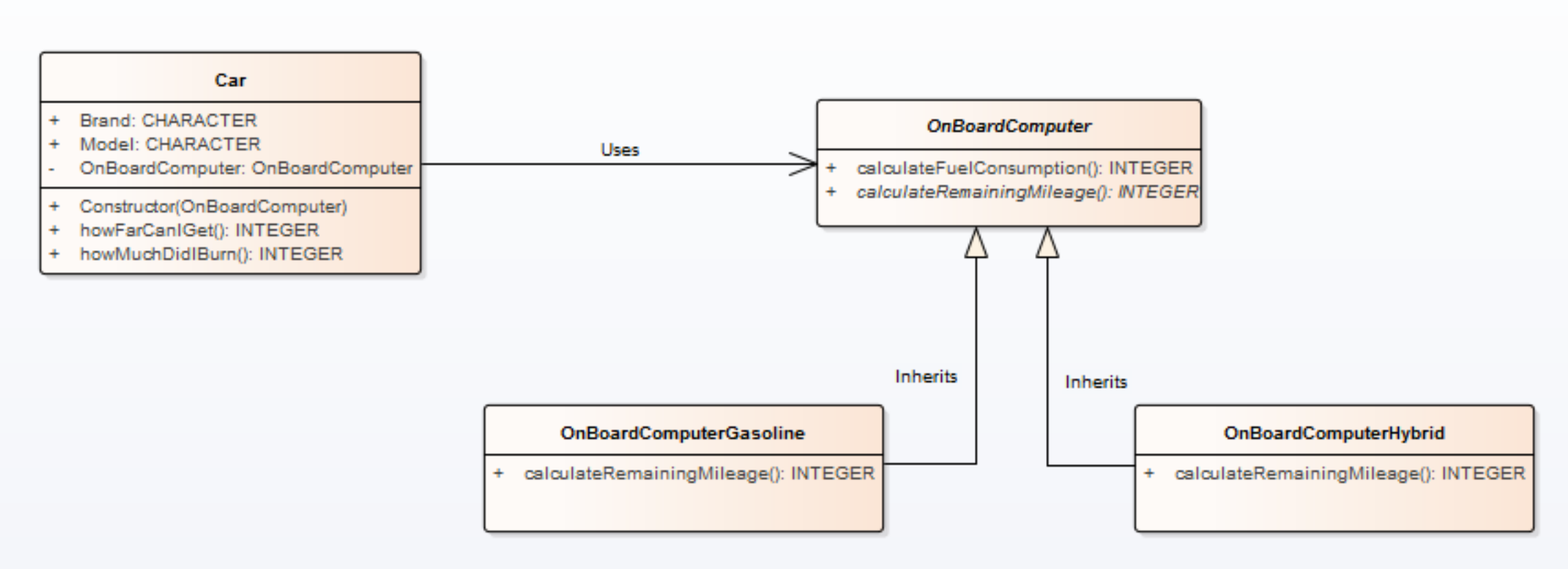
A. *High-level modules should not depend on low-level modules. Both should depend on abstractions.*

B. *Abstractions should not depend upon details. Details should depend upon abstractions.*

Dependency Inversion Principle (DIP)



Dependency Inversion Principle (DIP)



Favor Composition over Inheritance

When some behaviour of a domain object can change with other features remaining the same.

- Changes in the interface of a super class can break code that uses subclasses…
- E.g. Use the Strategy Pattern as alternative

- "*Make sure inheritance models the is-a relationship*"
  - Is an Employee a Person?

Program to an Interface

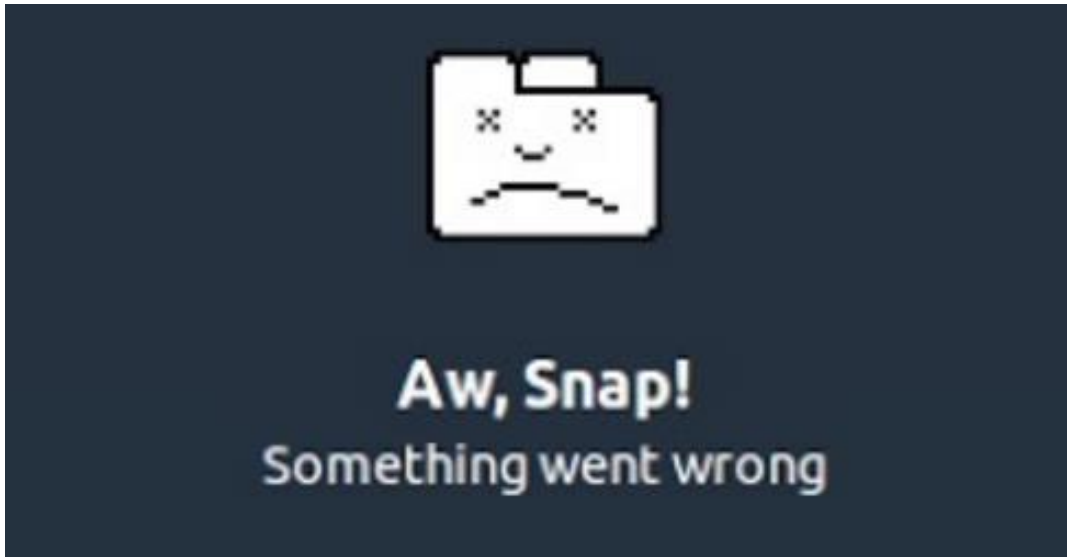
, not an implementation

- There is a distinct difference between Public and Published!

- Disadvantage: if you add methods to the interface, all clients need to adapt
- Abstract classes don’t have this disadvantage

Now let’s start our Presentation

RUN EMEAPUGPresentation().



Solution: NEW the Presentation

```
DEFINE VARIABLE objPresentation AS EMEAPUGPresentation NO-UNDO.
objPresentation = NEW EMEAPUGPresentation().
```

Works! But … no flexibility.

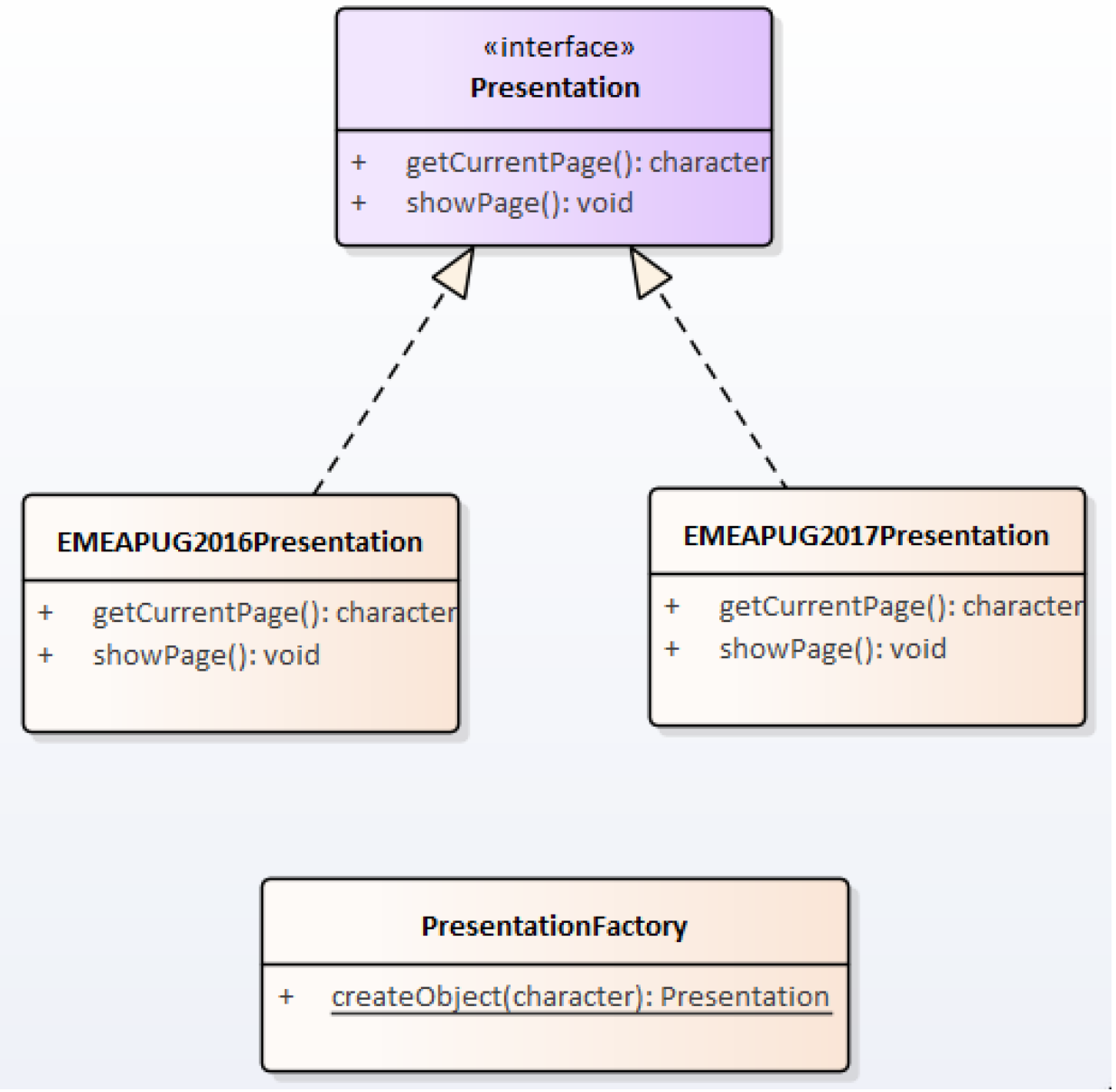
Solution: the Factory (Method) Design Pattern

- One of the most commonly used patterns
  - Delegate the construction to a **specific Factory**
  - Or a **generic Factory**

The specific Factory

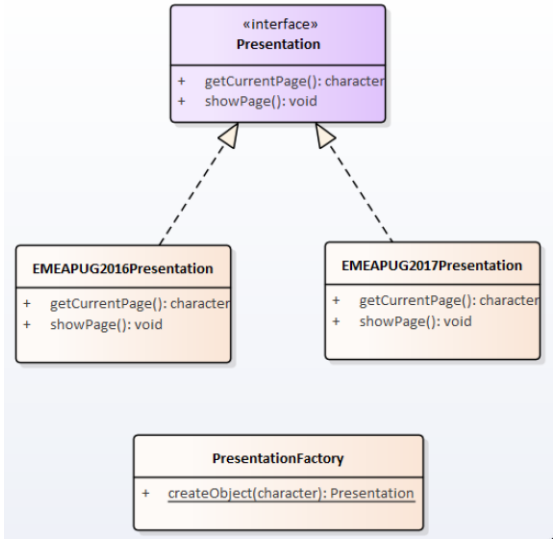
- Uses a CASE statement
- That NEWs the Presentation

Solution: The model



Solution: The code

INTERFACE **Presentation**:

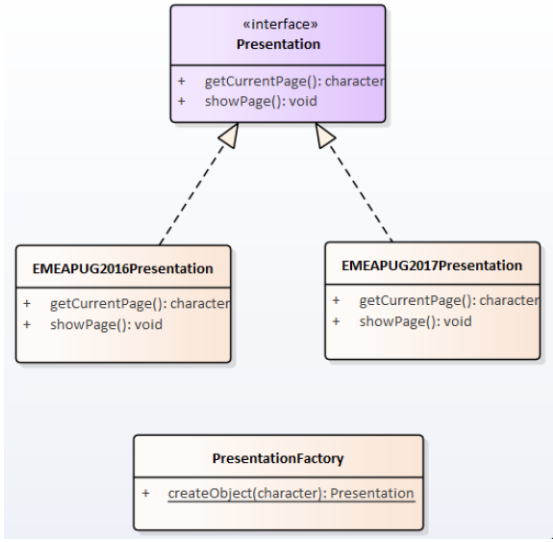


METHOD PUBLIC CHARACTER **getCurrentPage**( ):

METHOD PUBLIC VOID ShowPage( ):

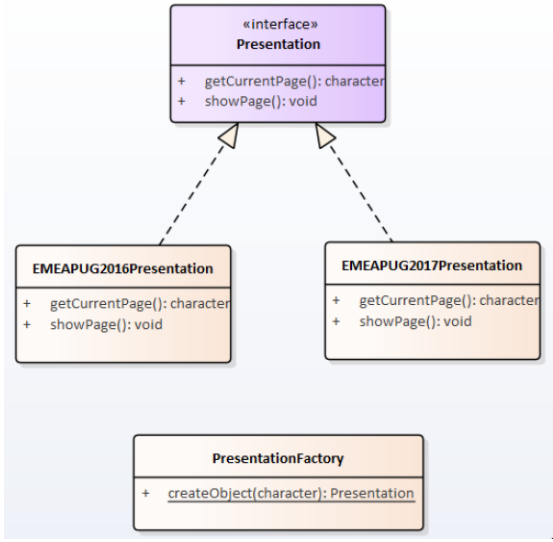
END INTERFACE.

Solution: The code



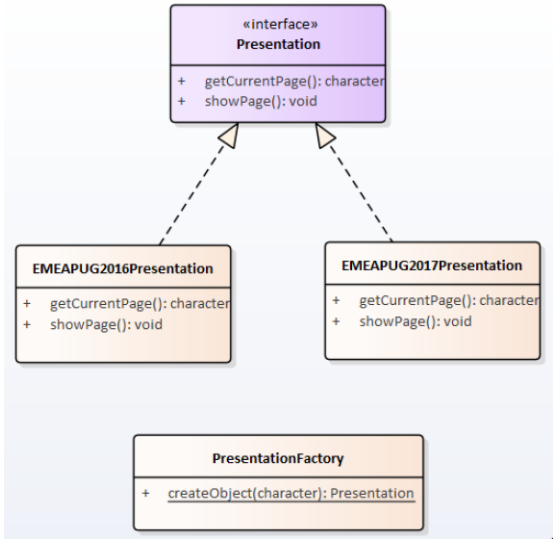

```
CLASS EMEAPUG2016Presentation
        IMPLEMENTS Presentation:

    METHOD PUBLIC CHARACTER getCurrentPage( ):
        RETURN "a digital representation of a 2016 presentation".
    END METHOD.

    METHOD PUBLIC VOID ShowPage( ):
        /* MESSAGE "Hello world" */
    END METHOD.

END CLASS.
```

Solution: The code




```
CLASS EMEAPUG2017Presentation
        IMPLEMENTS Presentation:

    METHOD PUBLIC CHARACTER getCurrentPage( ):
        RETURN "a digital representation of a 2017 presentation".
    END METHOD.

    METHOD PUBLIC VOID ShowPage( ):
        /* MESSAGE "Hello world" */
    END METHOD.

END CLASS.
```

Solution: The code

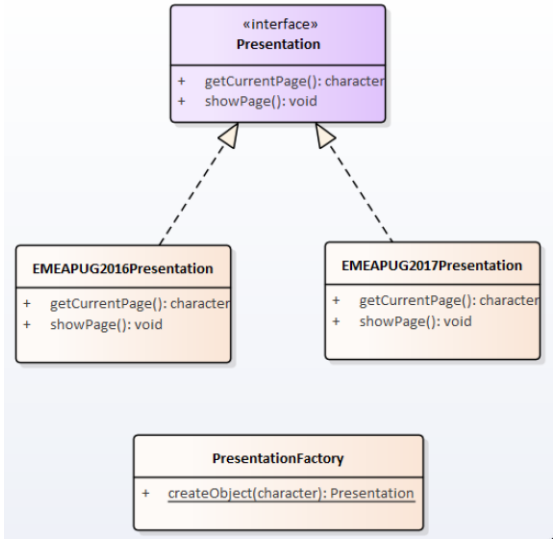


CLASS **PresentationFactory**:

```
  METHOD PUBLIC STATIC Presentation createObject(
                                    INPUT pcPresentation AS CHARACTER):
    DEFINE VARIABLE objPresentation AS Presentation NO-UNDO.

    CASE pcPresentation:
      WHEN "2016“ THEN objPresentation = NEW EMEAPUG2016Presentation().
      WHEN "2017“ THEN objPresentation = NEW EMEAPUG2017Presentation().
    END CASE.

    RETURN objPresentation.

  END METHOD.

END CLASS.
```

Solution: The code

```
DEFINE VARIABLE obj2016 AS Presentation NO-UNDO.
DEFINE VARIABLE obj2017 AS Presentation NO-UNDO.

obj2016 = PresentationFactory:createObject("2016").
obj2017 = PresentationFactory:createObject("2017").

MESSAGE
"It took our team a year to move from “
obj2016:getCurrentPage()
", to “
obj2017:getCurrentPage()
VIEW-AS ALERT-BOX.
```



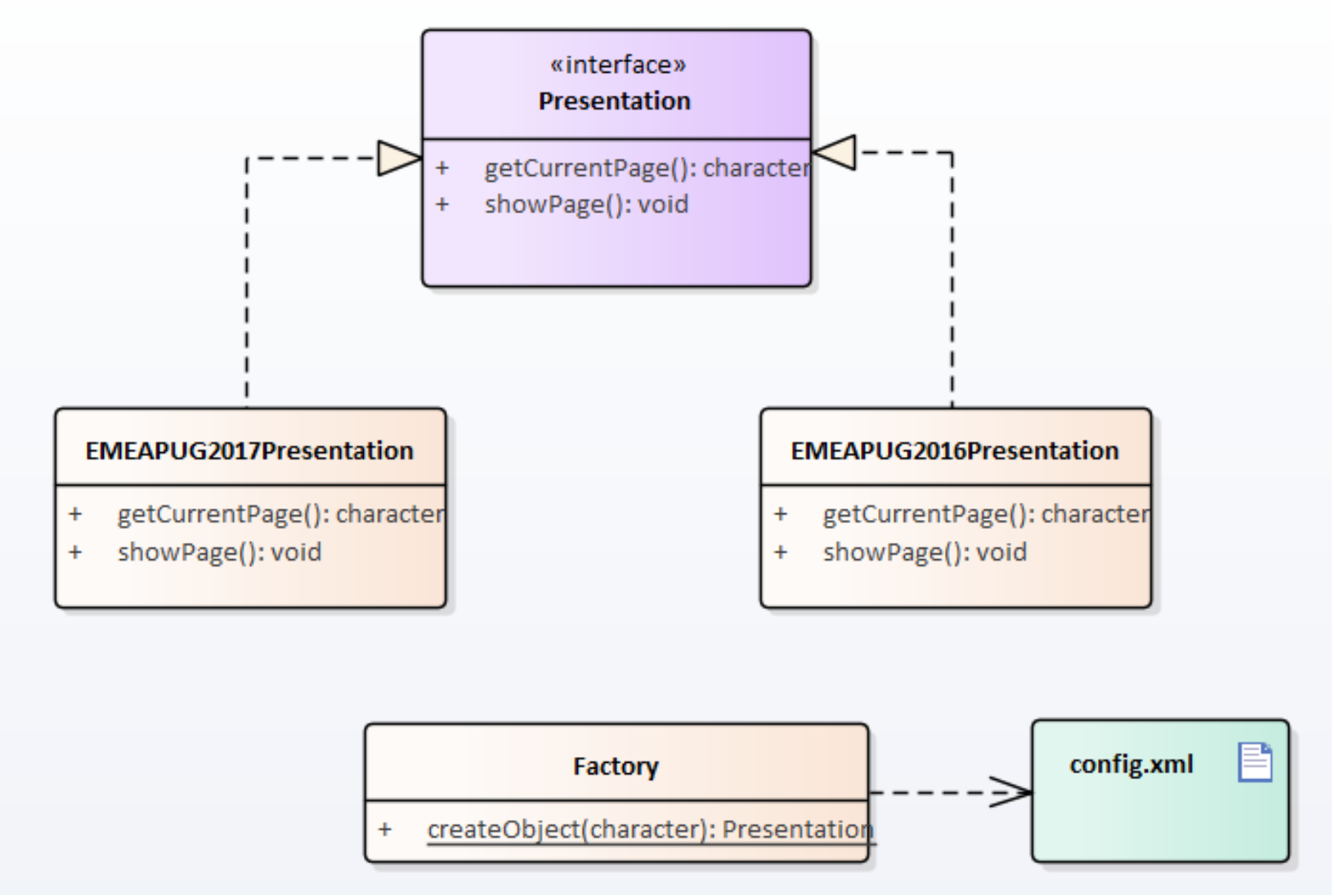
Solution: The demo

Disadvantages!

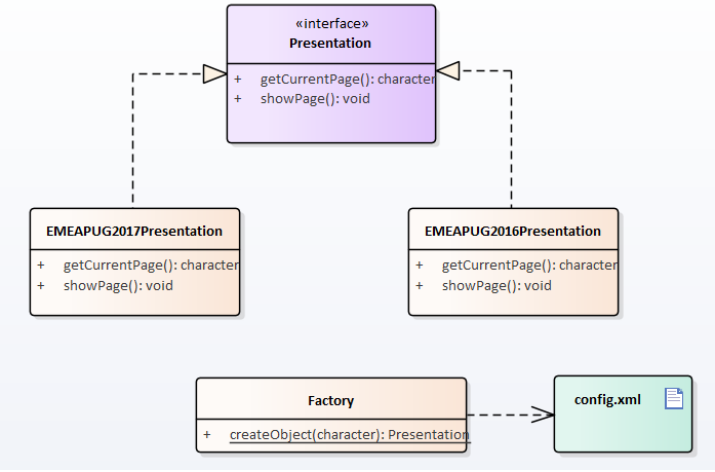
The generic Factory

- Is based on configuration data
- Uses DYNAMIC-NEW
- Works with INTERFACE classes

Solution: The model

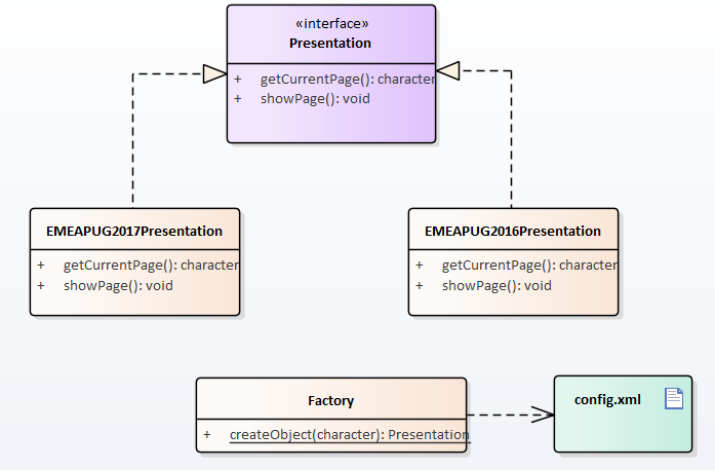


Solution: The configuration



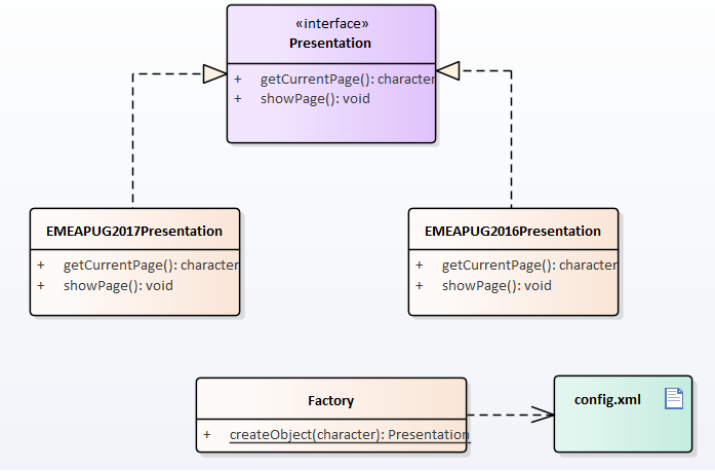
Node	Content
?=? xml	version="1.0"
e dsConfig	
@ xmlns:xsi	http://www.w3.org/2001/XMLSchema-instance
e ttConfig	
e cType	2016
e cClass	PUGDemo4.EMEAPUG2016Presentation
e ttConfig	
e cType	2017
e cClass	PUGDemo4.EMEAPUG2017Presentation

Solution: The code




```
CLASS Factory:
    DEFINE STATIC TEMP-TABLE ttConfig
    FIELD cType AS CHARACTER
    FIELD cClass AS CHARACTER
    INDEX idxType IS PRIMARY UNIQUE cType.
  DEFINE STATIC DATASET dsConfig FOR ttConfig.

  CONSTRUCTOR STATIC Factory ( ):
    DATASET dsConfig:READ-XML("FILE", SEARCH("config.xml"),?,?,?).
  END CONSTRUCTOR.

  METHOD PUBLIC STATIC Progress.Lang.Object createObject(ipcType AS CHARACTER):
    DEFINE VARIABLE objObject AS Progress.Lang.Object NO-UNDO.
    FIND FIRST ttConfig WHERE ttConfig.cType = ipcType.
    objObject =DYNAMIC-NEW ttConfig.cClass().
    RETURN objObject.
  END METHOD.

END CLASS.
```

Solution: The code

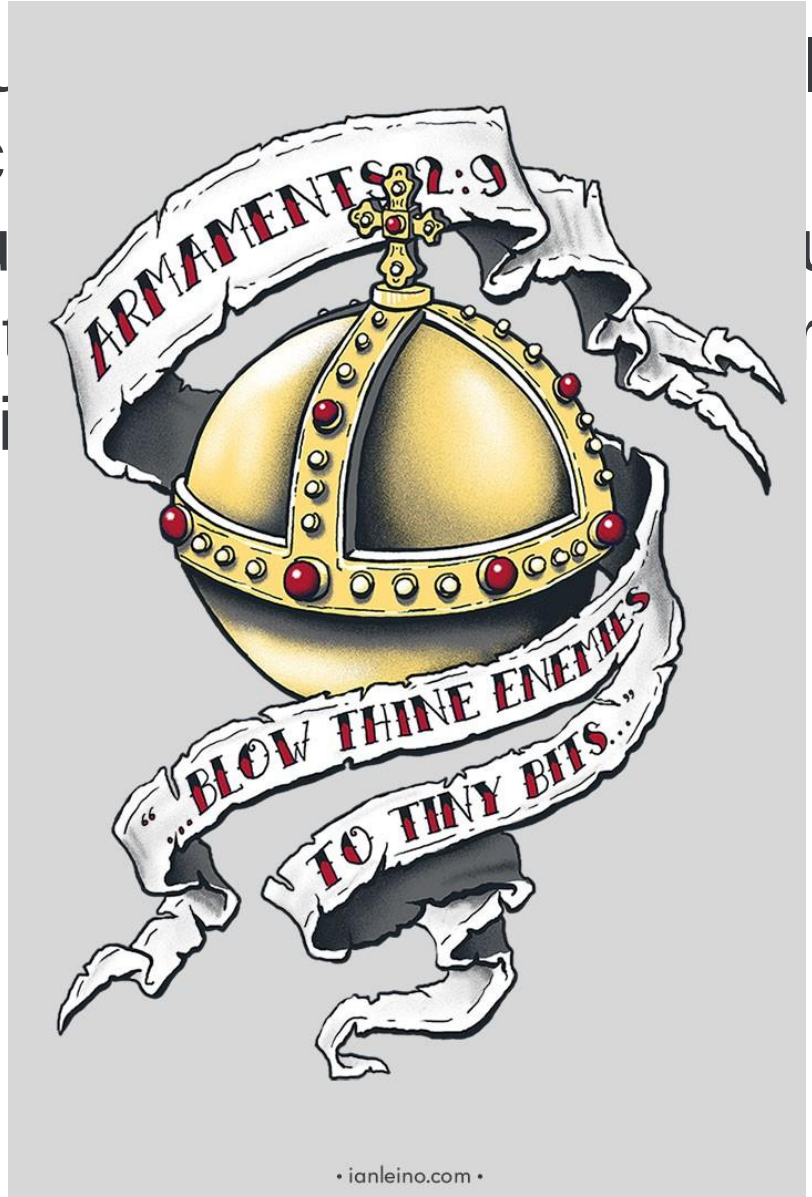



```
DEFINE VARIABLE obj2016 AS Presentation NO-UNDO.
DEFINE VARIABLE obj2017 AS Presentation NO-UNDO.

obj2016 =CAST(Factory:createObject("2016"),Presentation).
obj2017 =CAST(Factory:createObject("2017"),Presentation).

MESSAGE "It took our team a year to move from “
obj2016:getCurrentPage()
", to “
obj2017:getCurrentPage()
VIEW-ASALERT-BOX.
```

Solution: The demo

The number of presentations

- Then, shalt thou cou less. One shalt be the number thou shalt c the counting shall be One. Two shalt thou unt thou zero, excepting that thou ree is right out. Once the number one, bei

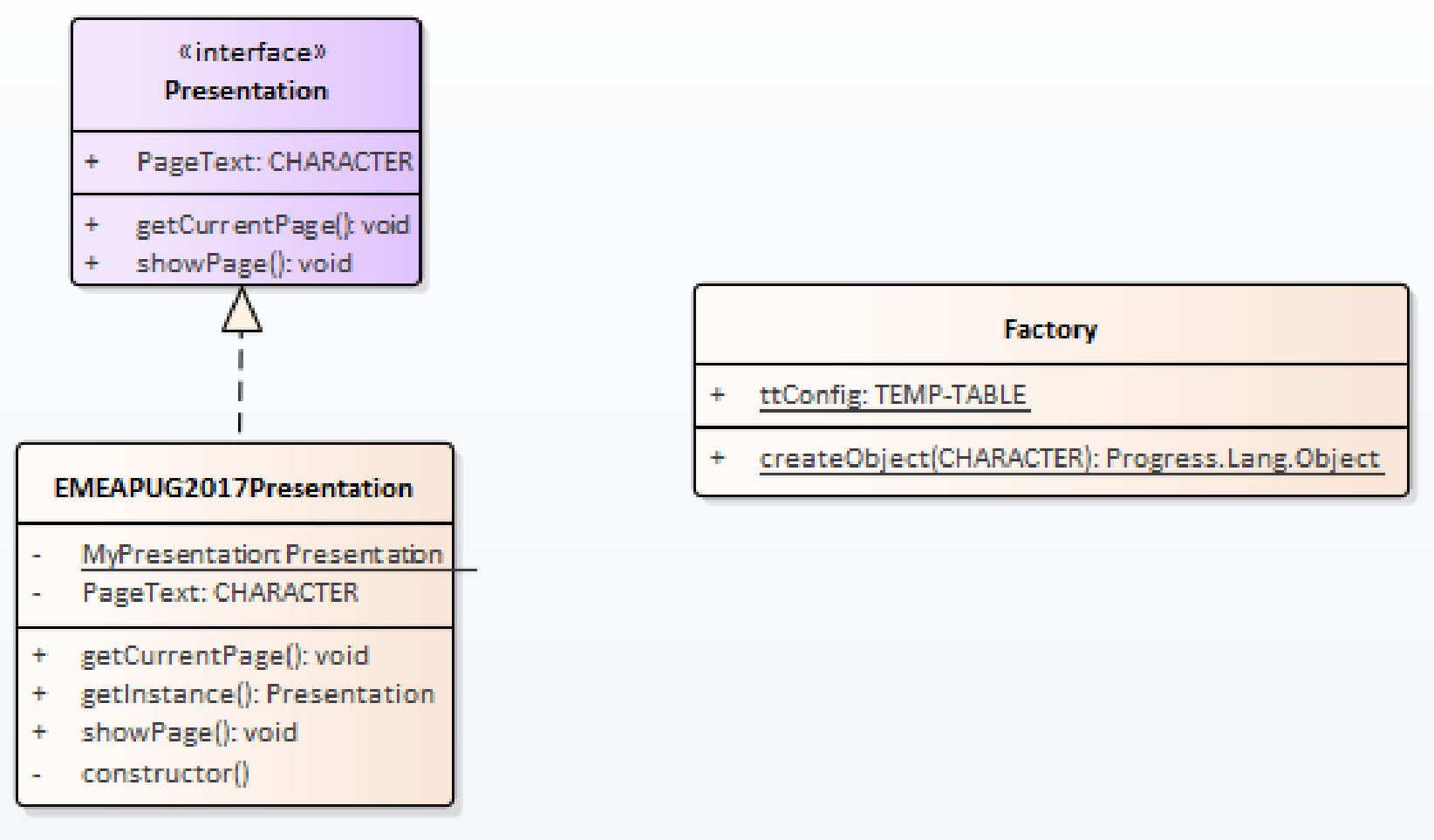


The Singleton Pattern

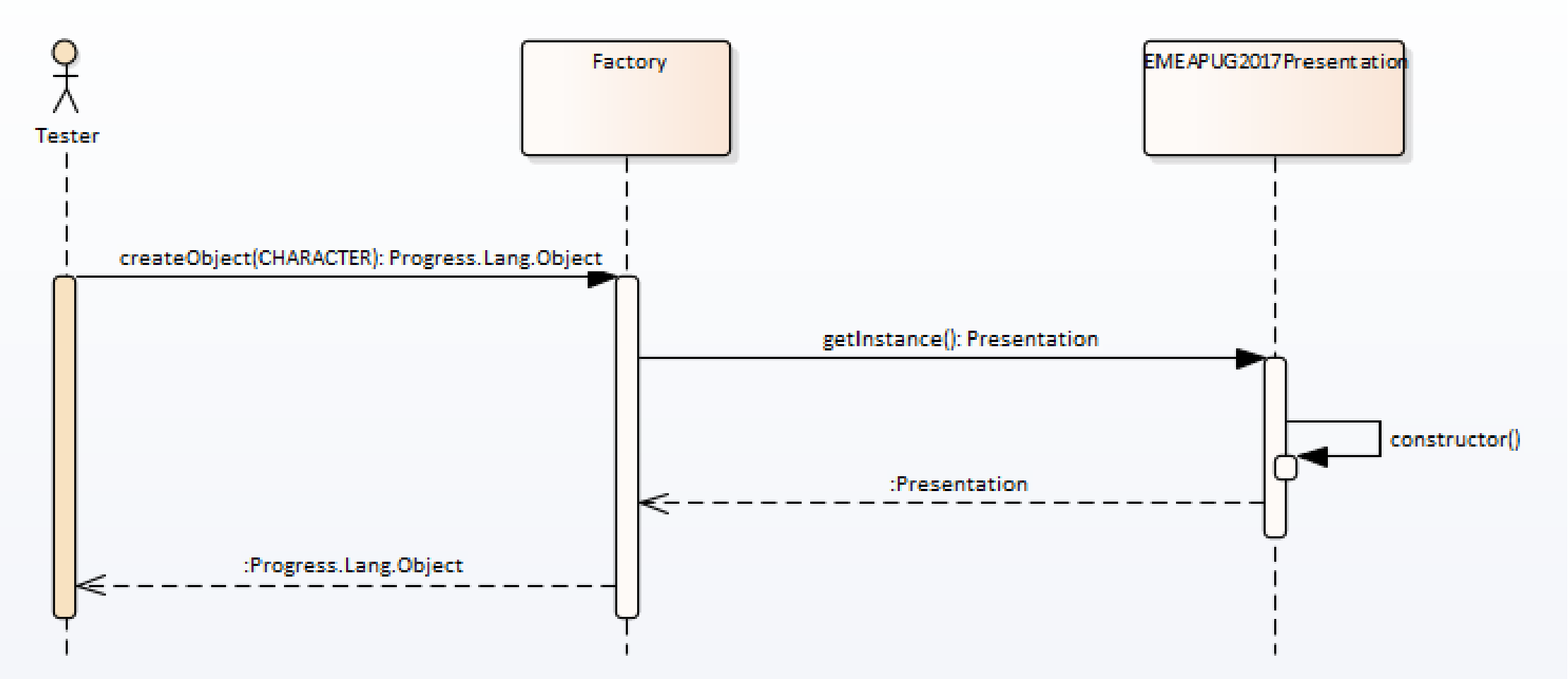
A software design pattern that restricts the instantiation of a class to one object.

- *An implementation of the singleton pattern must:*
  - *ensure that only one instance of the singleton class ever exists*
  - *provide global access to that instance.*
- *Typically, this is done by:*
  - *declaring all constructors of the class to be private*
  - *providing a static method that returns a reference to the instance.*

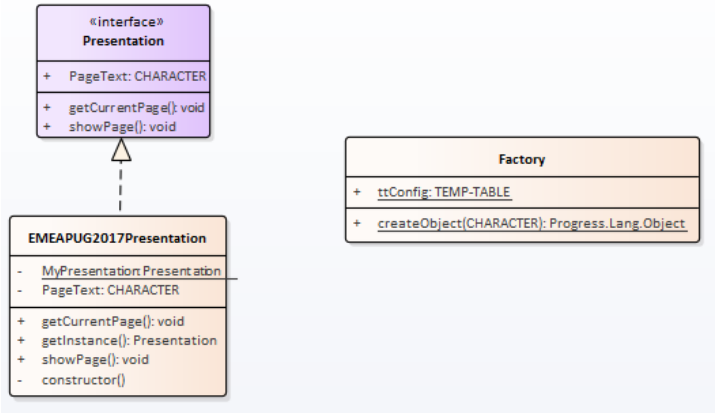
Solution: The model



Solution: The model



Solution: The code



INTERFACE **Presentation**:

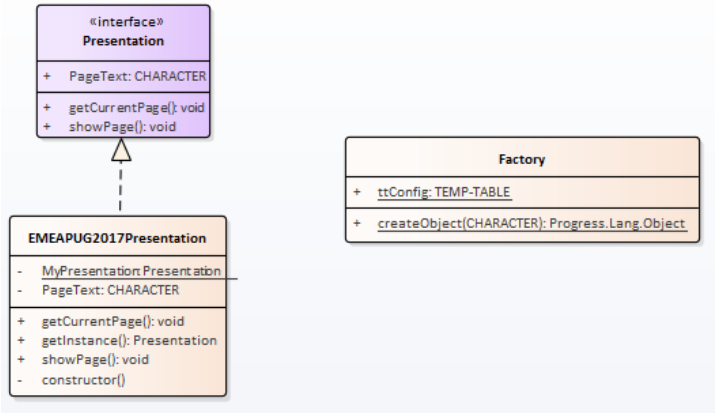
DEFINE PUBLIC PROPERTY **PageText** AS CHARACTER NO-UNDO
GET.
SET.

METHOD PUBLIC CHARACTER getCurrentPage( ):

METHOD PUBLIC VOID ShowPage( ):

END INTERFACE.

Solution: The code



CLASS EMEAPUG2017Presentation **IMPLEMENTS** Presentation:

DEFINE **PRIVATE STATIC** PROPERTY **MyPresentation** AS Presentation NO-UNDO GET. SET.

DEFINE PUBLIC PROPERTY **PageText** AS CHARACTER NO-UNDO GET. SET.

METHOD PUBLIC CHARACTER getCurrentPage( ):
    RETURN PageText.
END METHOD.

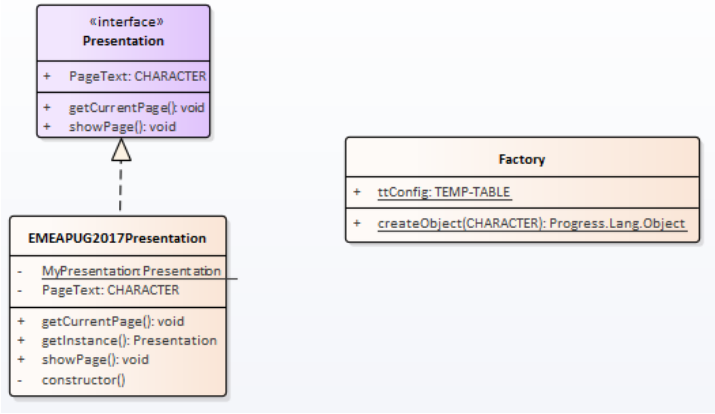
CONSTRUCTOR **PRIVATE** EMEAPUG2017Presentation ( ):
    SUPER().
END CONSTRUCTOR.

METHOD PUBLIC **STATIC** Presentation **getInstance**():
    IF NOT VALID-OBJECT(MyPresentation) THEN
        MyPresentation = NEW EMEAPUG2017Presentation().
    RETURN MyPresentation.
END METHOD.

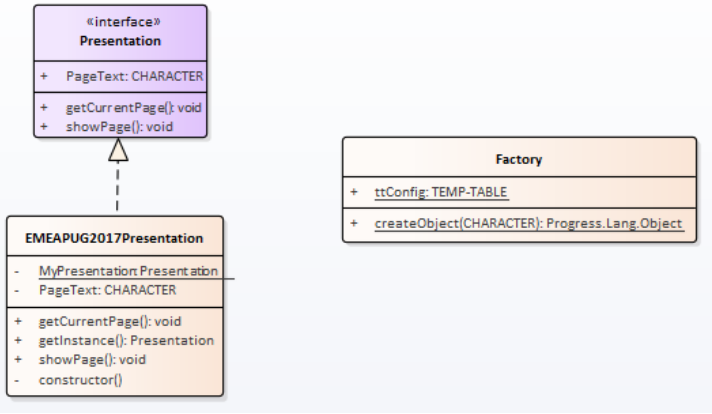
END CLASS.

Solution: The code

```
CLASS Factory:
    DEFINE STATIC TEMP-TABLE ttConfig
    FIELDcType AS CHARACTER
    FIELDcClass AS CHARACTER
    INDEX idxType IS PRIMARY UNIQUE cType.
  DEFINE STATIC DATASET dsConfig FOR ttConfig.

  CONSTRUCTOR STATIC Factory ( ):
    DATASETdsConfig:READ-XML("FILE", SEARCH("config.xml"),?,?,?).
  END CONSTRUCTOR.

  METHOD PUBLIC STATIC Progress.Lang.Object createObject(ipcType AS CHARACTER):
    DEFINE VARIABLE objObject AS Progress.Lang.Object NO-UNDO.

    FIND FIRST ttConfig WHERE ttConfig.cType = ipcType.
    objObject = DYNAMIC-INVOKE(ttConfig.cClass, "getInstance").
    RETURN objObject.

  END METHOD.
END CLASS.
```



Solution: The code




```
DEFINE VARIABLE obj2017First      AS Presentation.
DEFINE VARIABLE obj2017Second AS Presentation.

obj2017First      = CAST(Factory:createObject("2017"), Presentation).
obj2017Second = CAST(Factory:createObject("2017"), Presentation).

obj2017First:PageText =" Prague ".

MESSAGE "This is the presentation for “ obj2017First:getCurrentPage()
 VIEW-AS ALERT-BOX.
MESSAGE "This is the presentation for “ obj2017Second:getCurrentPage()
VIEW-AS ALERT-BOX.
```

Solution: The demo

All done!


Progress®

Instructor-led training: Advanced OO ABL in the OERA

15:00 - 16:00
Noordwijk

Advanced Design patterns in OO ABL

Roland de Pijper, Peter Judge, Progress Software

In version 10 Progress introduced Object Oriented ABL. Over time the OO elements have been extended and give developers the option to choose between good old ABL and OO ABL. Out of the box, an OO language doesn't necessarily improve the quality of your code. Especially within a team of developers, sticking to the same rules is crucial. To tackle this challenge the software industry has come up with 'Design Patterns', offering best practices for programmers to solve common problems. In 2 sessions, we will show you ways to implement the most commonly used design patterns in OO ABL. Some design patterns tackle the more complex challenges in software design. Like how to link objects to a relational database schema using an ORM. How does this fit into OO ABL? And why some people consider this to be an anti-pattern. Or how to enforce loosely coupling using MVC. Or does that just make things more complicated?