

Les Design Patterns ou patrons de conception

QUELQUES PISTES POUR APPLIQUER CETTE MÉTHODOLOGIE DANS DES
DÉVELOPPEMENTS VFP

Marc Thivolle - Lille (20 et 21 mars 2014)

Avertissement : qui suis-je ? d'où viens-je ?

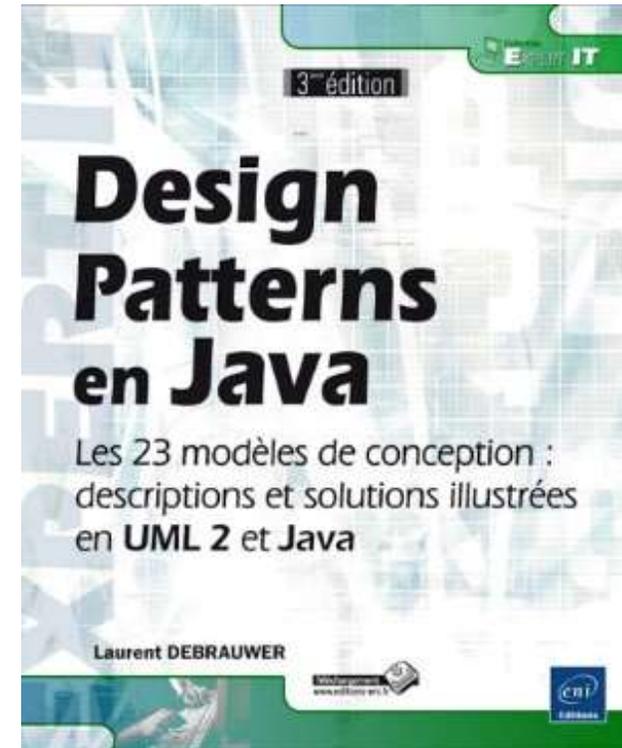
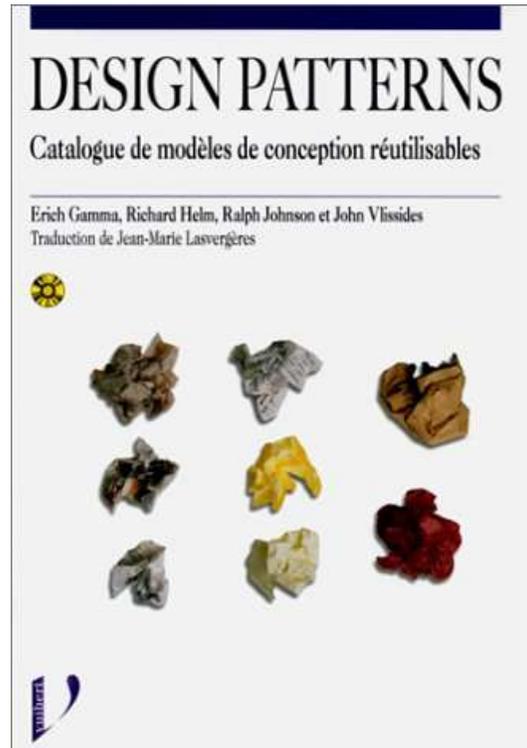
- ▶ 30 années de développement de logiciels de gestion dans le cadre de sociétés éditrices de logiciels
- ▶ Gestion de bases de données de volume important
- ▶ Interface utilisateur riche, intuitive et conviviale
- ▶ Pratique professionnelle de Delphi (version 3) et VFP (essentiellement version 6)
- ▶ Lecture, et vaguement écriture, de Java, Python, Perl et C#

Un peu de théorie

Un peu d 'histoire

- ▶ Travaux de l'architecte Christopher Alexander
- ▶ Formalisation en informatique avec l'ouvrage « Design Patterns – Elements of Reusable Object-Oriented Software » d'Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides (parution le 21-10-1994 avec un copyright de 1995)
- ▶ Les quatre auteurs forment le GoF (Gang of Four)
- ▶ Traduction française aux éditions International Thomson Publishing France (1996) puis aux éditions Vuibert (2007) sous le titre « Design Patterns. Catalogue de modèles de conception réutilisables »

L'ouvrage de référence et un ouvrage d'apprentissage efficace



Qu'est-ce qu'un patron de conception? (1)

- ▶ « Chaque modèle décrit un problème qui se manifeste constamment dans notre environnement, et donc décrit le cœur de la solution de ce problème, d'une façon telle que l'on peut réutiliser cette solution des millions de fois, sans le faire deux fois de la même manière. » (Christopher Alexander)
- ▶ Dans le domaine de l'architecture des logiciels un patron de conception représente la description d'une solution à un problème récurrent de conception. Il relève de ce que l'on appelle « les bonnes pratiques » de développement.

Qu'est-ce qu'un patron de conception? (2)

- ▶ Un patron de conception est un ensemble de relations et de règles liant plusieurs interfaces et/ou classes. Il exprime la solution à un problème donné dans un contexte précis.
- ▶ Les patrons de conception améliorent la qualité et la fiabilité du développement en minimisant les couplages entre les objets.
- ▶ Ils renforcent la cohésion au sein d'un ensemble de classe reliées fonctionnellement.
- ▶ Attention cependant à ne pas en abuser.

Ce que sont les patrons de conception

- ▶ La description d'une solution à un problème récurrent de conception.
- ▶ La description d'une partie de la solution, les autres parties étant les environnements logiciels et matériels.
- ▶ Une technique d'architecture logicielle.

Ce que ne sont pas les patrons de conception

- ▶ Une brique logicielle (le patron dépend de son contexte).
- ▶ Une règle que l'on peut appliquer mécaniquement (parce que justement il dépend de son contexte).
- ▶ Une méthode de prise de décision. Le patron est la décision.

Description d'un patron de conception

- ▶ Le nom du modèle
- ▶ Le problème dont le modèle est une solution (ou intention)
- ▶ La solution
- ▶ Les conséquences

Les deux portées des patrons de conception

- ▶ soit une portée de classe en se focalisant sur les relations entre les classes et leurs sous-classes, le mécanisme utilisé étant l'héritage
- ▶ soit une portée d'objet (ou d'instance) en se focalisant sur les relations entre les objets, le mécanisme utilisé étant la composition

Les trois types de patron de conception

- ▶ Créationnels : mécanismes pour l'instanciation d'objets
- ▶ Structuraux : organisation des interfaces et des classes en utilisant les mécanismes d'agrégation et de composition
- ▶ Comportementaux : communication entre les classes et répartition des responsabilités

Les 23 patrons de conception

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter
				Template Method
	Object	Abstract Factory	Adapter	Chain of Responsibility
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Proxy	Flyweight
				Observer
				State
				Strategy
				Visitor

Bref retour sur les interfaces

- ▶ Selon les langages on parlera d'interface, d'héritage multiple ou de classes abstraites
- ▶ VFP n'implémente pas ces notions. Mais rien n'empêche d'en assumer les contraintes à travers des règles strictes de programmation et des commentaires appropriés
- ▶ Une interface est un contrat que s'engage à remplir un objet
- ▶ Ce contrat se matérialise comme un ensemble de signatures (en VFP on pourra considérer que c'est l'ensemble des méthodes publiques)
- ▶ La signature d'une méthode est le triplet « type de retour, nom, types des paramètres »

Interfaces, classes, implémentations, objets

- ▶ Programmer et penser interface plutôt qu'implémentation
- ▶ Dans les langages qui l'autorisent, utiliser les interfaces et/ou les classes abstraites pour définir les interfaces communes à un ensemble de classe
- ▶ Déclarer les paramètres-objets comme instances des classes abstraites dont ils héritent plutôt que comme instances des classes particulières qui ont procédé à leur construction

Relations entre objets : héritage

- ▶ On parle aussi de spécialisation
- ▶ C'est la méthode de loin la plus naturelle pour un objet qui ne doit satisfaire qu'à une seule interface
- ▶ Attention à la prolifération de sous-classes hyper-spécialisées

Relations entre objets : exemple d'héritage

Exemple : mémoriser les positions d'une fenêtre afin de les restituer lors de l'ouverture de celle-ci

```
DEFINE CLASS myForm1 AS FORM
```

```
    PROCEDURE INIT  
        THIS.onRestore()  
    ENDPROC
```

```
    PROCEDURE MOVED()  
        THIS.onSave()  
    ENDPROC
```

```
    PROCEDURE RESIZE()  
        THIS.onSave()  
    ENDPROC
```

```
    PROCEDURE onSave  
    ENDPROC
```

```
    PROCEDURE onRestore  
    ENDPROC
```

```
ENDDEFINE
```

Relations entre objets : agrégation

- ▶ Une propriété du premier objet est une référence vers l'instance du second objet.
- ▶ On simule ainsi l'héritage multiple.
- ▶ Plus modulaire que la spécialisation elle permet d'adapter finement le comportement à l'exécution par le choix au dernier moment de la classe réelle de l'objet instancié.

Relations entre objets : exemple d'agrégation

```
loParam=CREATEOBJECT('oParam')  
loForm=CREATEOBJECT('myForm3',loParam)
```

```
DEFINE CLASS myForm3 AS FORM
```

```
loParam=NULL
```

```
PROCEDURE INIT(poParam)  
  THIS.loParam=poParam  
  THIS.loParam.onRestore(THIS)  
ENDPROC
```

```
PROCEDURE MOVED()  
  THIS.loParam.onSave(THIS)  
ENDPROC
```

```
PROCEDURE RESIZE()  
  THIS.loParam.onSave(THIS)  
ENDPROC
```

```
ENDDEFINE
```

```
DEFINE CLASS oParam AS CUSTOM
```

```
  PROCEDURE onSave(poForm)  
  ENDPROC
```

```
  PROCEDURE onRestore(poForm)  
  ENDPROC
```

```
ENDDEFINE
```

Relations entre objets : composition

- ▶ Identique à l'agrégation à la différence près que le cycle de vie du second objet est contrôlé par le premier.
- ▶ Le second objet est créé (puis détruit) pour un usage exclusif. Cette méthode s'impose si des variables d'état sont indispensables.

Relations entre objets : exemple de composition

```
DEFINE CLASS myForm2 AS FORM
```

```
  loParam=NULL
```

```
  PROCEDURE INIT
```

```
    THIS.loParam=CREATEOBJECT('oParam',THIS)
```

```
    THIS.loParam.onRestore()
```

```
  ENDPROC
```

```
  PROCEDURE MOVED()
```

```
    THIS.loParam.onSave()
```

```
  ENDPROC
```

```
  PROCEDURE RESIZE()
```

```
    THIS.loParam.onSave()
```

```
  ENDPROC
```

```
ENDDDEFINE
```

```
DEFINE CLASS oParam AS CUSTOM
```

```
  loForm=NULL
```

```
  PROCEDURE INIT(poForm AS FORM)
```

```
    THIS.loForm=poForm
```

```
  ENDPROC
```

```
  PROCEDURE onSave
```

```
  ENDPROC
```

```
  PROCEDURE onRestore
```

```
  ENDPROC
```

```
ENDDDEFINE
```

Remarque : j'ai préféré la syntaxe `THIS.loParam=CREATEOBJECT('oParam',THIS)` à la syntaxe `ADDOBJECT('loParam','oParam',THIS)` car elle met bien en évidence l'appartenance de l'objet `loParam` à l'objet `myForm2`

Relations entre objets : conclusion

- ▶ Les auteurs les plus sérieux affirment qu'il faut préférer la composition d'objets à l'héritage.
- ▶ Les dépendances d'implémentation sont ainsi évitées.
- ▶ La programmation est centrée sur la réalisation d'une tâche et les comportements peuvent changer en cours d'exécution.
- ▶ Dans la littérature agrégation et composition sont souvent des notions variables.
- ▶ Attention : dans l'ouvrage de référence du Gof, agrégation = composition et accointance = agrégation

Un peu de pratique

« Adaptateur » (définition)

- ▶ Convertit l'interface d'une classe en une interface conforme à l'attente du client
- ▶ Ce patron permet à des classes de collaborer alors que leurs interfaces sont incompatibles

« Adaptateur » (exemple)

- Interface à adapter : WinApi qui intègre la fonction onShellExecute dont la signature (en VFP) est **INT onShellExecute STRING STRING STRING STRING INT**

```
DEFINE CLASS WinApi AS CUSTOM
```

```
PROCEDURE INIT
```

```
DECLARE INTEGER ShellExecute ;
```

```
IN SHELL32.DLL ;
```

```
INTEGER nWinHandle,;
```

```
STRING cOperation,;
```

```
STRING cFileName,;
```

```
STRING cParameters,;
```

```
STRING cDirectory,;
```

```
INTEGER nShowWindow
```

```
ENDPROC
```

```
FUNCTION onShellExecute(tcFileName,tcWorkDir,tcOperation,tcParam,tnWindow)
```

```
LOCAL lcFileName,lcWorkDir,lcOperation,lcParam,lnWindow
```

```
IF EMPTY(tcFileName)
```

```
RETURN -1
```

```
ENDIF
```

```
lcFileName=ALLTRIM(tcFileName)
```

```
lcWorkDir=IIF(TYPE('tcWorkDir')='C',ALLTRIM(tcWorkDir),")
```

```
IF(TYPE('tcOperation')='C' AND NOT EMPTY(tcOperation)
```

```
lcOperation=ALLTRIM(tcOperation)
```

```
ELSE
```

```
lcOperation='Open'
```

```
ENDIF
```

```
lcParam=IIF(TYPE('tcParam')='C',ALLTRIM(tcParam),")
```

```
lnWindow=IIF(TYPE('tnWindow')='N',INT(tnWindow),1)
```

```
RETURN ShellExecute(0,lcOperation,lcFileName,lcParam,lcWorkDir,lnWindow)
```

```
ENDFUNC
```

```
ENDDEFINE
```

« Adaptateur » (exemple)

- ▶ Interface client : iDocument dont les signatures sont :
 - ▶ INT DocOpen STRING
 - ▶ INT DocPrint STRING

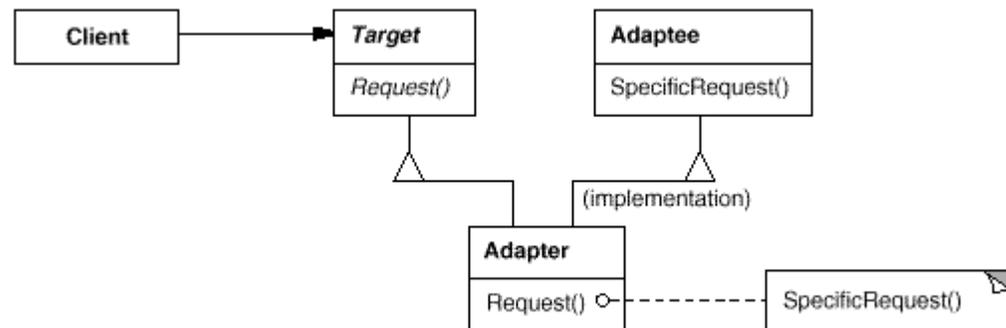
```
DEFINE CLASS IDocument AS CUSTOM

    FUNCTION DocOpen(docName AS STRING) AS INTEGER
        RETURN -2
    ENDFUNC

    FUNCTION DocPrint(docName AS STRING) AS INTEGER
        RETURN -2
    ENDFUNC

ENDDFINE
```

« Adaptateur » (classe – diagramme OMT)



« Adaptateur » (classe - exemple)

```
DEFINE CLASS AdapterWinApi1 AS WinApi  && implémente IDocument
```

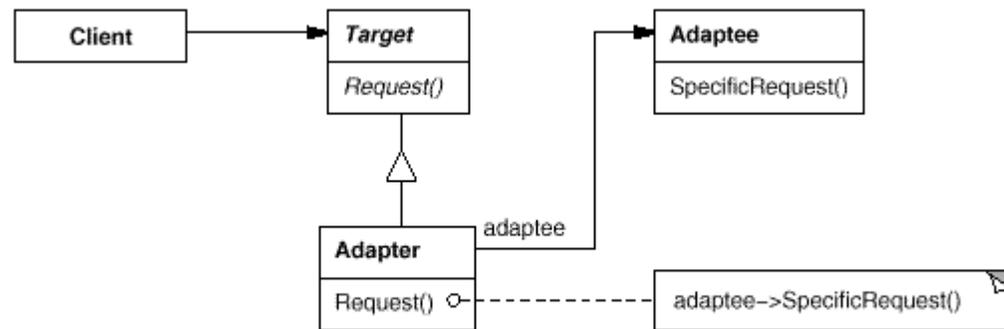
```
    FUNCTION DocOpen(docName AS STRING) AS INTEGER  
        RETURN THIS.onShellExecute(docName,','open',",",1)  
    ENDFUNC
```

```
    FUNCTION DocPrint(docName AS STRING) AS INTEGER  
        RETURN THIS.onShellExecute(docName,','print',",",1)  
    ENDFUNC
```

```
ENDDFINE
```

```
oClient1=CREATEOBJECT('AdapterWinApi1')  
oClient1.DocOpen('C:\AtoutFox\Lille2014\VFP\config9.txt')  
oClient1.DocPrint('C:\AtoutFox\Lille2014\VFP\config9.txt')  
RELEASE oClient1
```

« Adaptateur » (objet – diagramme OMT)



« Adaptateur » (objet - exemple)

```
DEFINE CLASS AdapterWinApi2 AS CUSTOM  && implémente IDocument

    loWinApi=NULL

    PROCEDURE INIT
        THIS.loWinApi=CREATEOBJECT('WinApi')
    ENDPROC

    FUNCTION DocOpen(docName AS STRING) AS INTEGER
        RETURN THIS.loWinApi.onShellExecute(docName,','open',",1)
    ENDFUNC

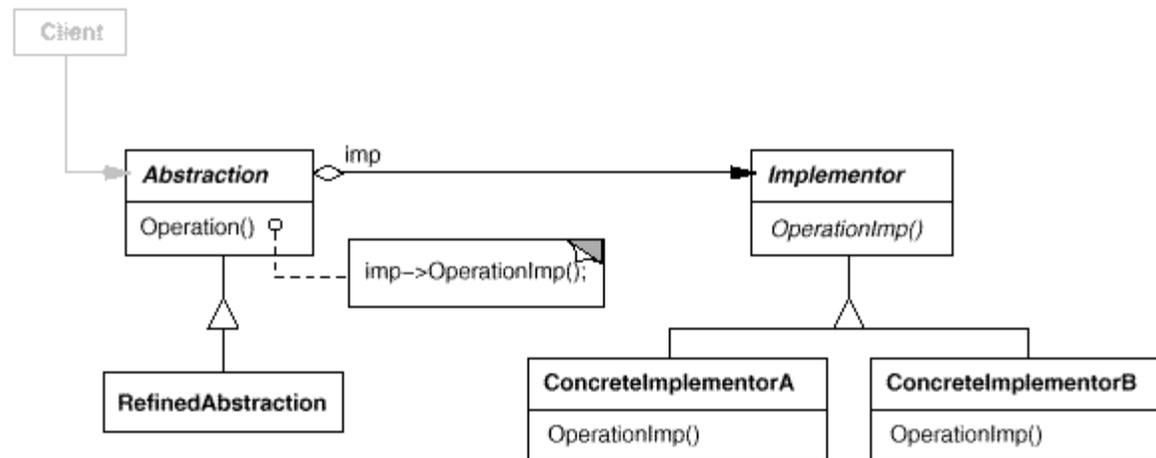
    FUNCTION DocPrint(docName AS STRING) AS INTEGER
        RETURN THIS.loWinApi.onShellExecute(docName,','print',",1)
    ENDFUNC

ENDDEFINE

oClient2=CREATEOBJECT('AdapterWinApi2')
oClient2.DocOpen('C:\AtoutFox\Lille2014\VFP\config9.txt')
oClient2.DocPrint('C:\AtoutFox\Lille2014\VFP\config9.txt')
RELEASE oClient1
```

« Pont » (définition)

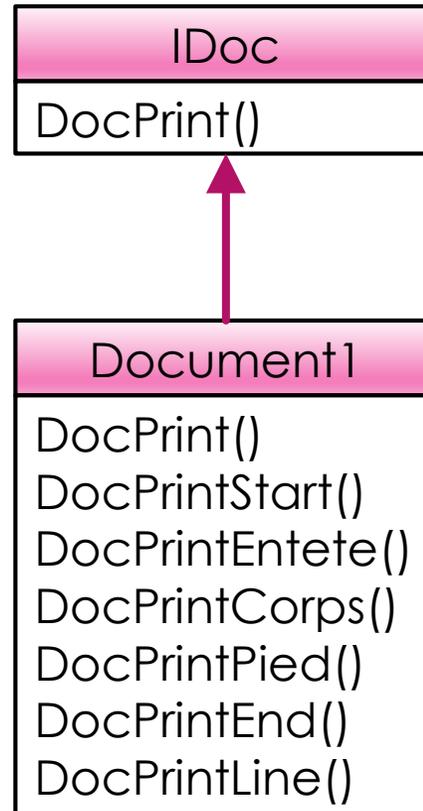
- Découple une abstraction de son implémentation afin que les deux éléments puissent être modifiés indépendamment l'un de l'autre.



« Pont » (exemple - définition)

- ▶ Une interface de gestion de document IDoc avec une seule méthode publique DocPrint
- ▶ L'analyse produit la décomposition suivante :
 - ▶ Initialisation (DocPrintStart)
 - ▶ Impression de l'en-tête du document (DocPrintEntete)
 - ▶ Impression du corps du document (DocPrintCorps)
 - ▶ Impression du pied du document (DocPrintPied)
 - ▶ Finalisation (DocPrintEnd)
 - ▶ Impression d'une ligne (DocPrintLine)

« Pont » (exemple - définition)



« Pont » (exemple – première évolution)

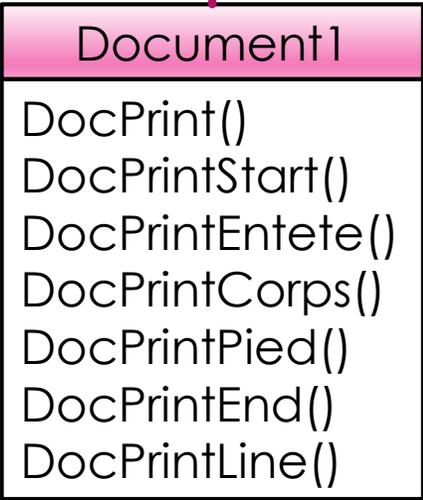
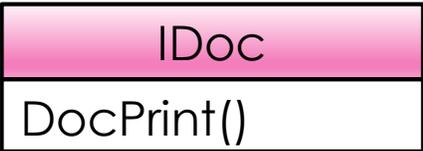
- ▶ Première spécialisation : type de document
 - ▶ devis avec l'impression d'une lettre de présentation
 - ▶ facture avec l'impression des coordonnées bancaires

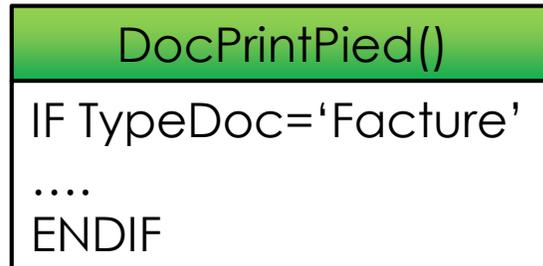
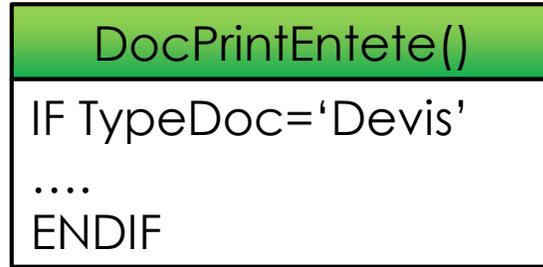
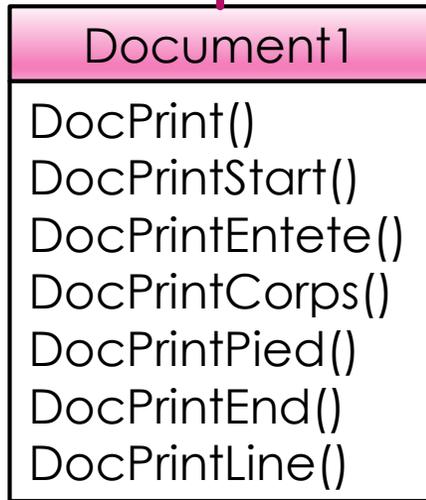
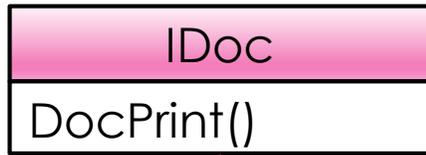
« Pont » (exemple – deuxième évolution)

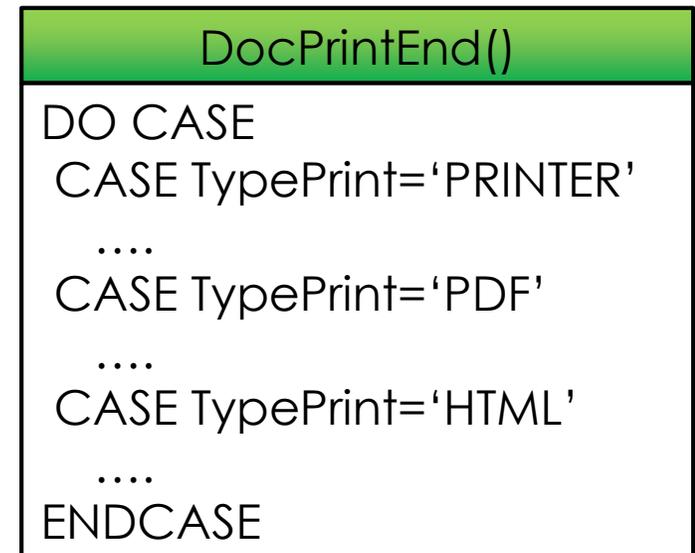
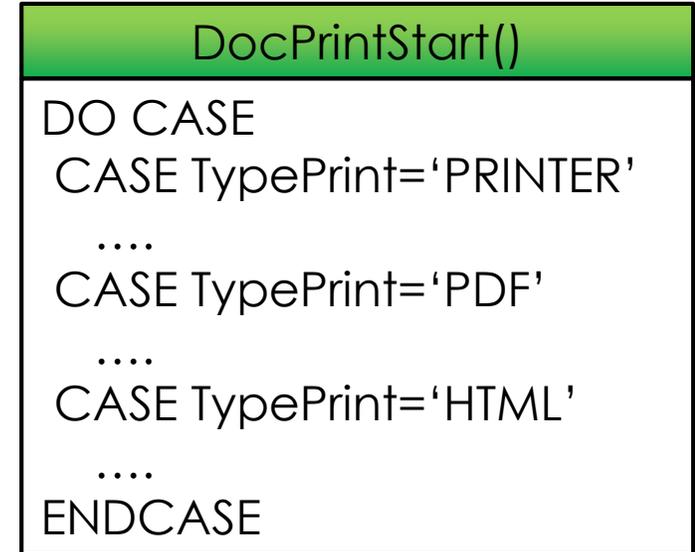
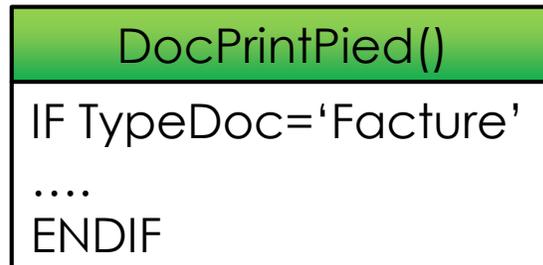
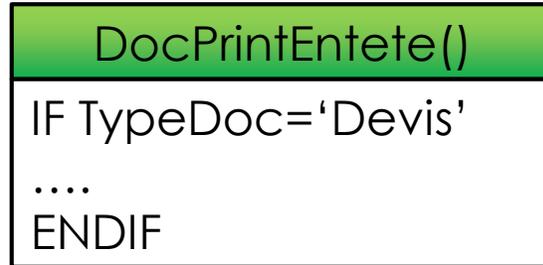
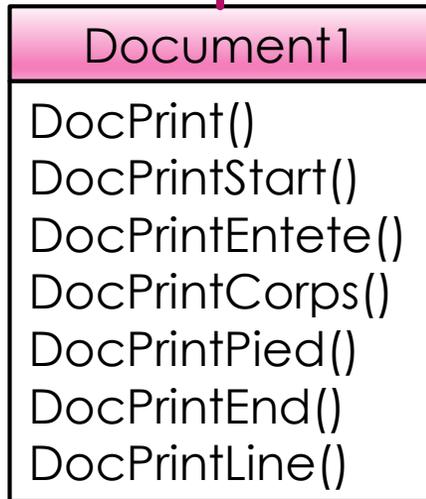
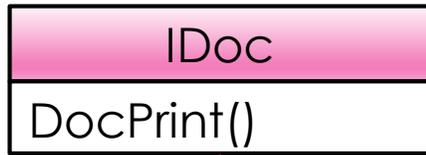
- ▶ Deuxième spécialisation : type de la sortie
 - ▶ imprimante,
 - ▶ PDF
 - ▶ HTML

« Pont » (exemple – la tradition - un)

- ▶ La première piste :
 - ▶ Des tests à l'intérieur des méthodes sur le type de document
 - ▶ Des tests à l'intérieur des méthodes sur le type de sortie
- ▶ Le code en devient vite illisible

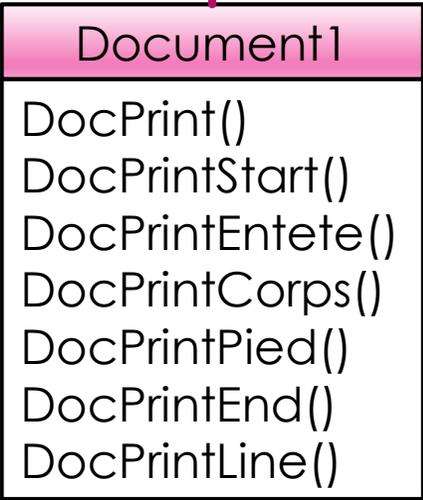
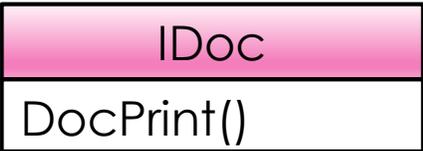


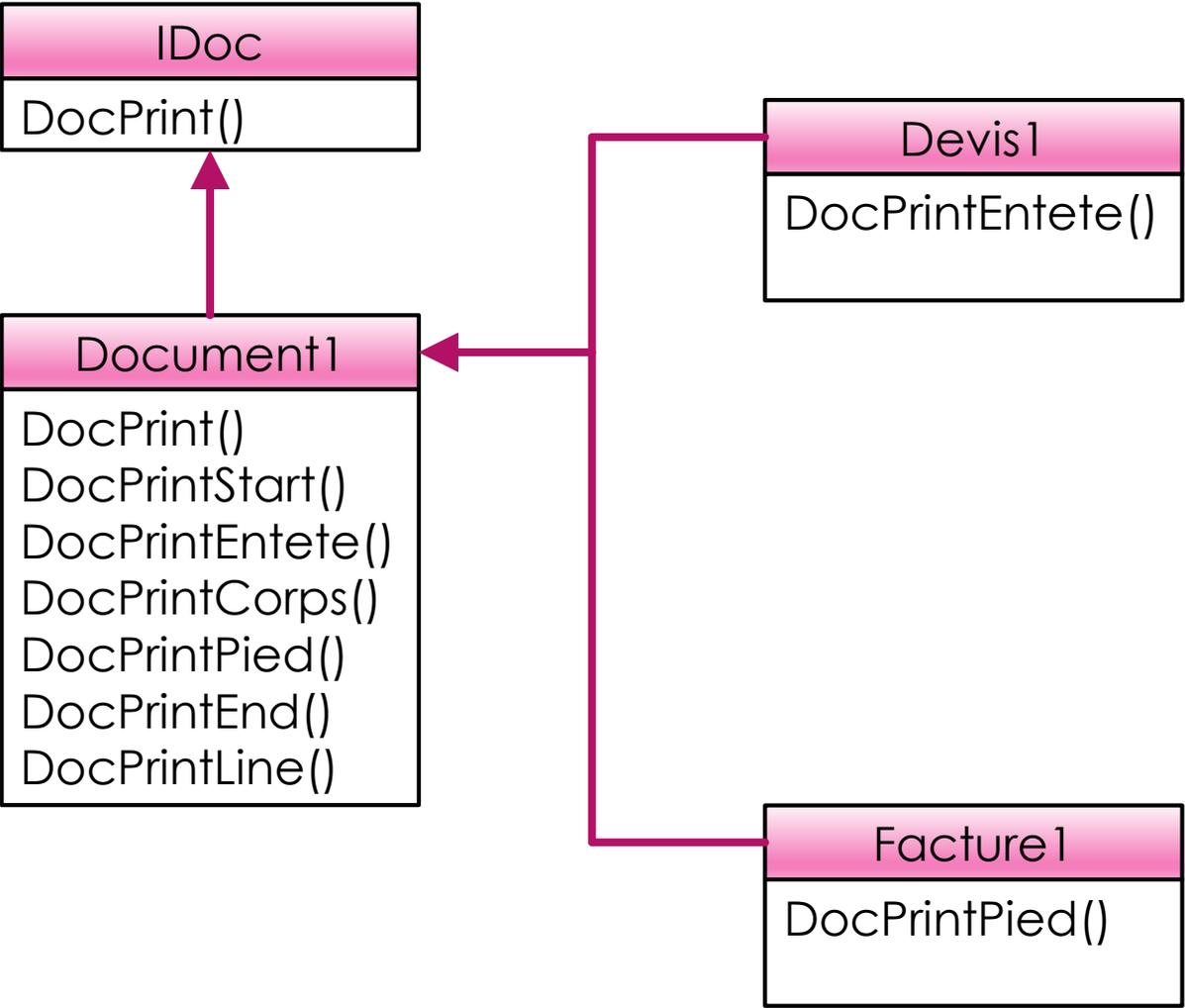


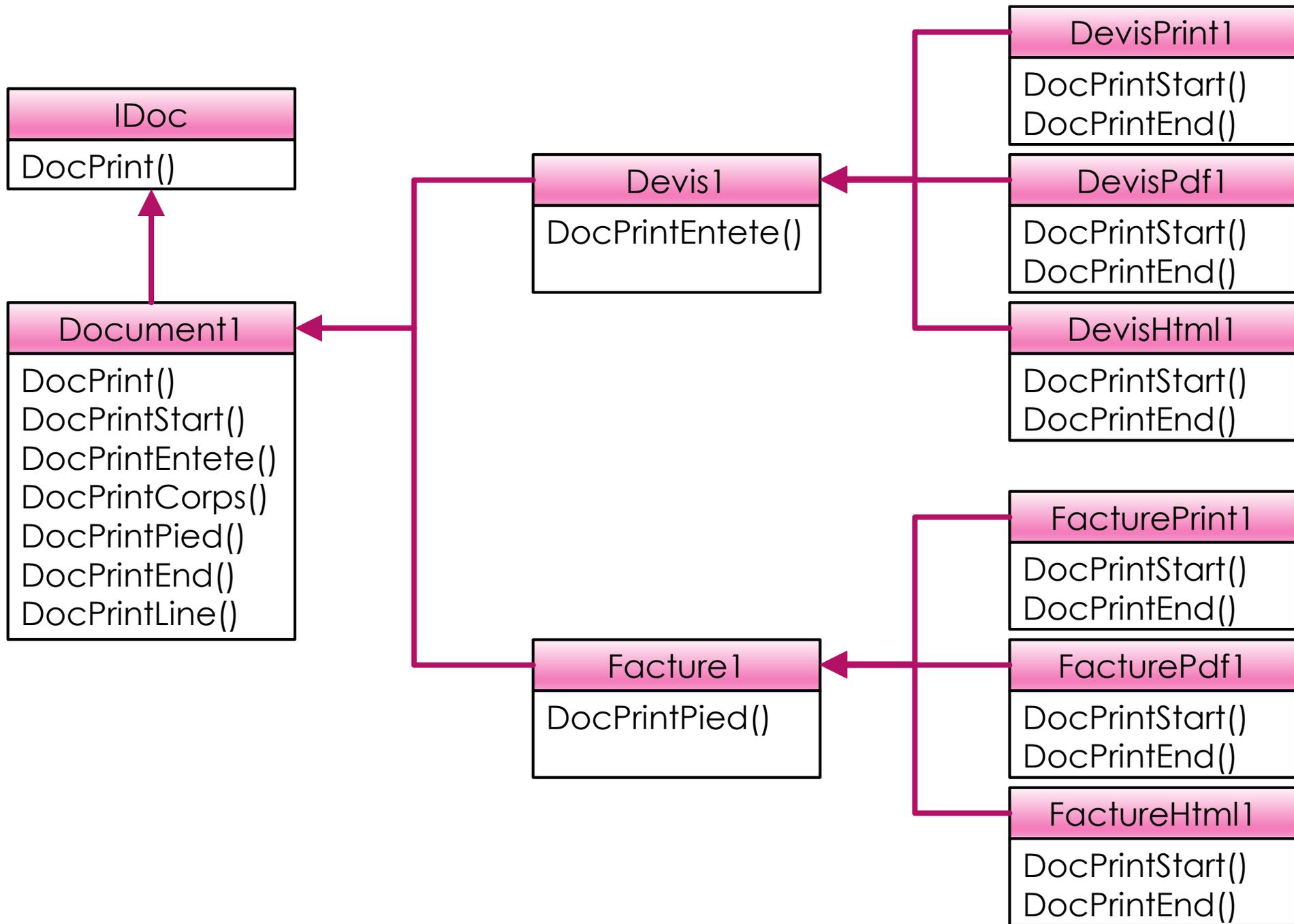


« Pont » (exemple – la tradition - deux)

- ▶ La deuxième piste :
 - ▶ Un premier sous-classement pour les types de document
 - ▶ Un second sous-classement pour les types de sortie
- ▶ Attention à la prolifération des classes

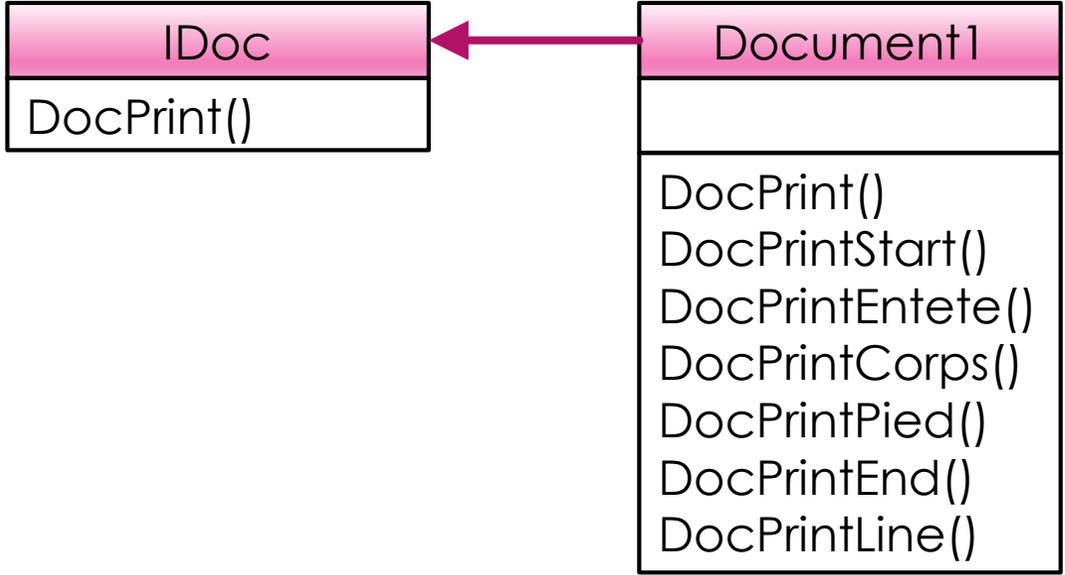


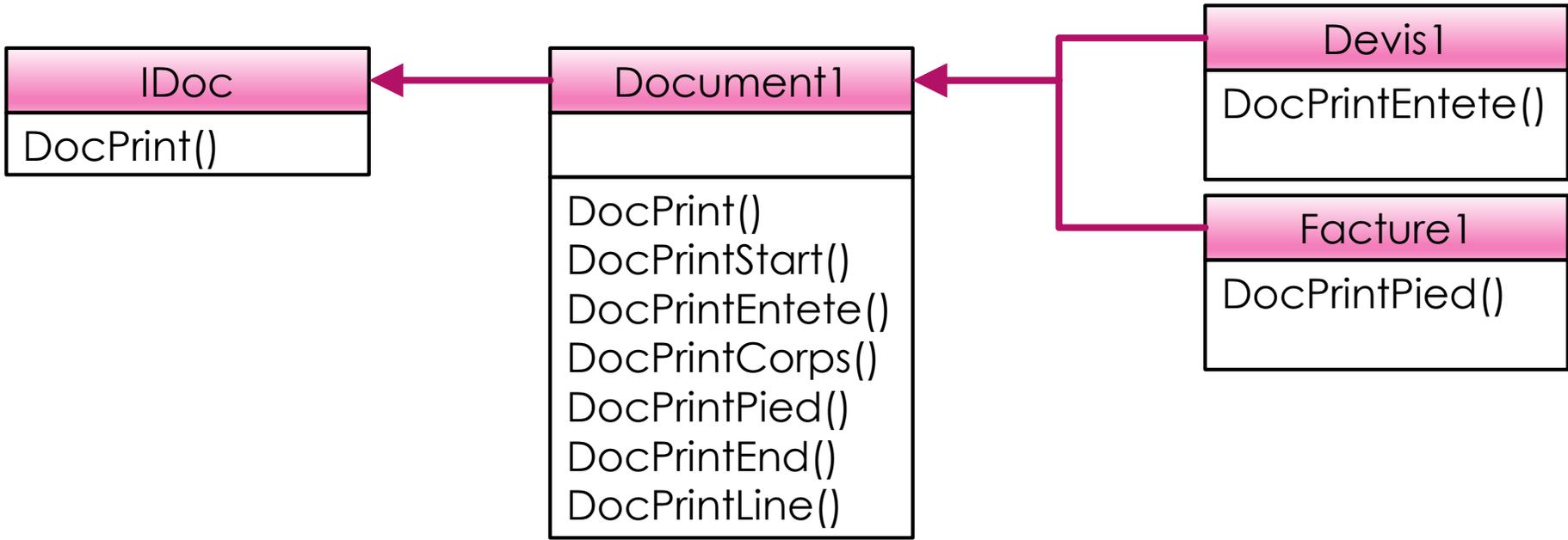


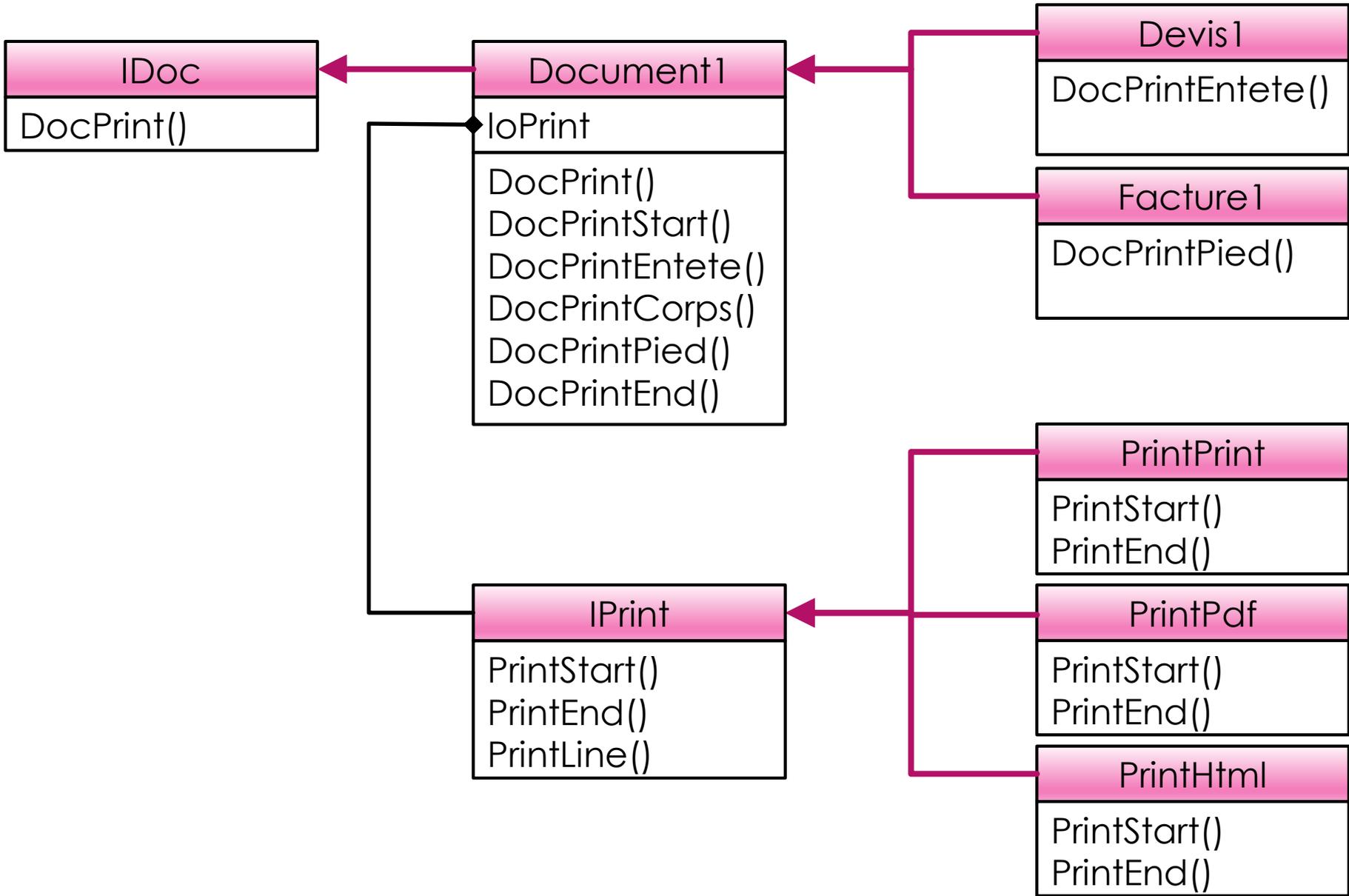


« Pont » (exemple – le bon pont)

- ▶ Deux interfaces : IDoc et IPrint
- ▶ Une instance de IPrint dans les classes implémentant IDoc
- ▶ Les méthodes de IDoc (chemin « abstrait ») appellent les méthodes de IPrint (implémentation « concrète » de primitives)
- ▶ Les deux interfaces peuvent être spécialisées de manière indépendante
- ▶ Une seule contrainte : les méthodes de IDoc ne doivent utiliser que les primitives proposées par IPrint







En guise de conclusion

- ▶ Trouver les bons objets
- ▶ Bien choisir la granularité
- ▶ Penser interface
- ▶ Encapsuler l'implémentation
- ▶ Favoriser la réutilisation